Appears in the Proceedings of HotWeb 2006, November, 2006

TCP Connection Management Mechanisms for Improving Internet Server Performance

Amol Shukla[†] and Tim Brecht
David R. Cheriton School of Computer Science, University of Waterloo
{ashukla, brecht}@cs.uwaterloo.ca

Abstract— **This paper investigates TCP connection management mechanisms in order to understand the behaviour and improve the performance of Internet servers during overload conditions such as flash crowds. We study several alternatives for implementing TCP connection establishment, reviewing approaches taken by existing TCP stacks as well as proposing new mechanisms to improve server throughput and reduce client response times under overload. We implement some of these mechanisms in Linux and evaluate their performance. Our evaluation demonstrates that connection establishment mechanisms that eliminate the TCP-level retransmission of connection attempts by clients can increase server throughput by up to 40% and reduce client response times by two orders of magnitude. Additionally we evaluate the cost of supporting half-closed connections at the server and assess the impact of an abortive release of connections by clients on the throughput of an overloaded server. We observe that mechanisms that do not support half-closed connections additionally improve server throughput by more than 15%.**

I. Introduction

The demand for Internet-based services has exploded over the last decade. The ubiquitous nature of web browsers has given rise to the occurrence of *flash crowds* where a large number of users simultaneously access a particular web site. Flash crowds are characterized by a rapid and dramatic surge in the volume of requests, prolonged periods of overload, and are often triggered without advance warning. In the hours following the September 11th terrorist attacks, many media sites such as CNN and MSNBC were overwhelmed with more than an order of magnitude increase in traffic, pushing their availability to 0% and their response time to over 47 seconds [1], [2]. A previously unpopular web site can see a huge influx of requests after being mentioned in well-known news feeds or discussion sites, resulting in saturation and unavailability – this is popularly known as the *Slashdot effect* [3].

In many web systems, once client demand exceeds the server's capacity, the server throughput drops sharply and the client response time increases significantly. Ironically, it is precisely during these periods of high demand that a web site's quality of service matters the most. Over-provisioning the capacity in web systems is often inadequate. Server capacity needs to be increased by at least 4-5 times to deal with even moderate flash crowds and the added capacity tends to be more than 82% idle during normal loads [4], most often making this approach fiscally imprudent. Web servers form a critical part of the Internet infrastructure and it is imperative to ensure that they provide reasonable performance during overload conditions such as flash crowds.

[†]Now at RealNetworks, Inc.

Past work [5] has reported that the escalation in traffic during flash crowds occurs largely because of an increase in the number of clients, resulting in an increase in the number of TCP connections that a server must handle. TCP stack developers in various flavours of UNIX and Windows have taken different approaches toward implementing TCP connection establishment (i.e., the three-way handshake). We review several of these approaches, implement them in Linux, and evaluate their performance. Additionally, we explore alternatives to standard TCP connection termination at both end-points. The main contributions of this paper are:

- We provide a better understanding of Internet server behaviour during overload conditions such as flash crowds.
- We demonstrate that the connection establishment mechanisms used in many existing TCP stacks result in degraded server throughput and increased client response times during overload.
- We present an alternative server-kernel mechanism that eliminates the retransmission of connection attempts by client-side TCP stacks during high loads, thereby improving server throughput by up to 40% and reducing client response times by more than two orders of magnitude. This mechanism does not require any changes to protocol specifications, client TCP stacks or applications, or server applications.
- Our performance evaluation indicates that disabling support for half-closed connections as well as an abortive release of connections by clients can improve server throughput by up to 15%.

II. Background

A typical HTTP interaction between a client and a server consists of the client establishing a TCP connection with the server, sending an HTTP request, receiving the server response, and terminating the connection. Multiple rounds of request-response transactions can take place over a single TCP connection if both of the end-points use persistent HTTP 1.1 connections. In the following paragraphs, we briefly describe how connection establishment and termination is typically implemented in a client-server environment, using the Linux TCP stack as a representative example.

A. TCP Connection Establishment

TCP connection establishment involves a three-way handshake between the end-points [6]. Figure 1 illustrates the handshake implementation in Linux. We now briefly discuss the portions of this process most relevant to our paper.

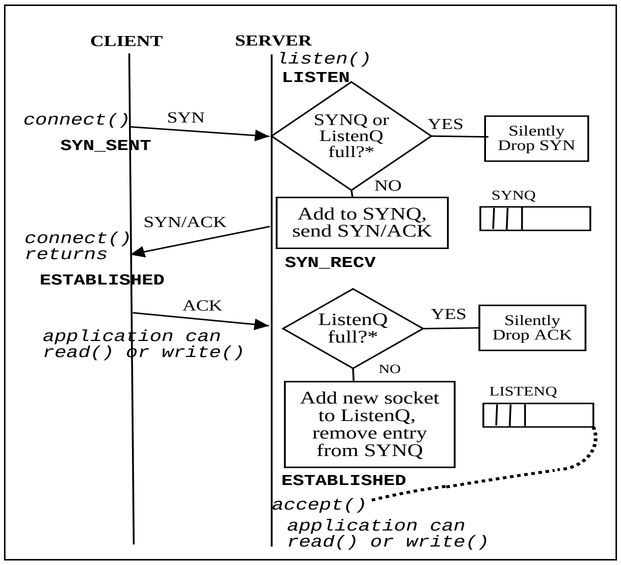


Fig. 1. TCP connection establishment in Linux

Upon receiving a SYN, the server TCP stack creates an entry identifying the client's connection request in the listening socket's SYN queue (sometimes called the SYN backlog and often implemented as a hash table). It then acknowledges the SYN by sending a SYN/ACK segment. The handshake is complete when the client TCP stack acknowledges the SYN/ACK with an ACK. We will refer to the ACK sent by the client in response to a SYN/ACK as SYN/ACK ACK to distinguish it from other ACK segments used in TCP. Upon receiving the SYN/ACK ACK, the server TCP stack creates a new socket, adds it to the listening socket's listen queue (sometimes called the accept queue), and removes the associated entry from the SYN queue. In order to communicate with the client, the server application has to issue the `accept()` system call, which removes the socket from the listen queue and returns an associated socket descriptor to the application. Note that in most socket API implementations, the three-way connection establishment procedure is completed by the server TCP stack *before* the application issues an `accept()` call.

The server TCP stack might not always be in a position to accommodate a SYN or a SYN/ACK ACK, this happens primarily when the SYN or the listen queue is full. The queues may become full because the rate of incoming client connection attempts is higher than the rate at which the server application is able to accept and process new connections[1]. A SYN or SYN/ACK ACK segment that cannot be accommodated has to be dropped, we refer to this scenario as a *queue drop*. The conservative approach of dropping connection establishment attempts earlier (at the SYN stage rather than the ACK stage) when the listen queue is full is also implemented in other TCP stacks such as FreeBSD. Similarly, in most cases, a SYN/ACK ACK is dropped when the listen queue is full upon arrival at the server.

[1]Increasing queue lengths does not improve the situation when the server is overloaded because the queues quickly become full.

Most TCP stacks implement connection establishment as described above. In order to try to protect against SYN flood denial of service attacks, some stacks use techniques such as SYN cookies [7] or SYN cache [8] to reduce the state the server is required to store to track incomplete connection requests. However, implementations differ significantly in how they react to queue drops. In Linux, SYN segments as well as SYN/ACK ACK segments are dropped silently when they trigger a queue drop. That is, no notification is sent to clients about these dropped segments. Most 4.2 BSD-derived TCP stacks, such as those in FreeBSD, HP-UX, and Solaris, only drop SYN segments silently [9]. Whenever a SYN/ACK ACK is dropped due to listen queue overflow, a TCP reset (RST) segment is sent to the client notifying it of the server's inability to continue with the connection. Some Windows TCP stacks do not drop either of these connection establishment segments silently, sending a RST to the client every time there is a queue drop. Note that in TCP, segments (except RSTs) that are not acknowledged by the server within a particular amount of time are retransmitted by the client.

In Section III, we critique the different approaches to handling queue drops; we are particularly interested in answering the following question – TCP stack developers in Linux, various flavours of UNIX, and Windows have taken different approaches to implementing connection establishment. Which of these approaches, if any, result in better performance under overload? We also present two novel mechanisms designed to eliminate the retransmission of TCP connection establishment segments in order to increase server throughput and reduce client response times.

B. TCP Connection Termination

A TCP connection is full-duplex and both sides can terminate their end of the connection independently through a "FIN-ACK" handshake after they finish sending data [6]. That is, each end-point transmits a FIN to indicate that it is not going to *send* any more data on a connection. This method of connection termination is called "graceful close".

Graceful connection closure is implemented with either half-closed (also called full-duplex) or half-duplex termination semantics. The CLOSE operation outlined in RFC 793 allows for a connection to be "half closed", allowing an end-point that sends a FIN to *receive* data from its peer. Some socket APIs provide the `shutdown()` system call to provide half-closed connection semantics, enabling applications to shutdown the sending side of their connection, while allowing activity on the receiving side through subsequent `read()` calls.

Most applications, however, use the `close()` system call to terminate both the sending as well as the receiving directions of the connection by treating the connection as if it is half-duplex [10]. RFC 1122 specifies – "A host may implement a ‘half-duplex’ TCP close sequence, so that an application that has called CLOSE cannot continue to read data from the connection. If such a host issues a CLOSE call while received data is still pending in TCP, or if new data is received after CLOSE is called, its TCP should send a RST to show that data was lost.” [11]. A `close()` call typically returns immediately and destroys the socket descriptor so the application loses its reference to the TCP connection.

Any client-initiated graceful connection termination, either with half-closed or half-duplex connection semantics, results in a FIN being sent to the server. Upon receiving a FIN segment, most server TCP stacks assume that the client uses half-closed semantics (i.e., the `shutdown()` system call). That is, they support half-closed client connections by default. However, most web clients do not terminate connections using half-closed connection semantics, instead they use the `close()` system call. In Section IV-A, we demonstrate how supporting half-closed connections can result in an imprudent use of resources at the server, especially during overload. We describe an alternative connection termination mechanism that disables support for half-closed connections allowing us to evaluate the cost of supporting half-closed connections when the server is overloaded.

Instead of graceful closure, an application can also force a connection to be terminated through an abortive release, which causes the TCP stack to send a reset (RST) segment to its peer. RFC 793 and RFC 2616 [12] strongly discourage the use of an abortive release to terminate connections as a part of normal operations. However, some client applications, in particular, Internet Explorer 5 and 6, which are currently the most popular web browsers, terminate *all* of their connections by forcing the client TCP stack to send a RST [13]. The reason why Internet Explorer uses an abortive release to terminate connections is not clear. In this paper, we examine the impact that abortive release has on server throughput under high loads. We are aware of the potential problems that can arise from not supporting half-closed connections or from an abortive release of connections; our approach toward these mechanisms is exploratory, not prescriptive.

III. Connection Establishment Alternatives

In this section, we outline some problems with existing implementations of TCP connection establishment using the Linux stack for illustration. We divide our discussion in Section III-A into problems arising at the ACK stage when the listen queue overflows and at the SYN stage when SYN segments are silently dropped. In Section III-B we review existing solutions and present novel mechanisms to address both of these problems.

A. Problems with Existing Mechanisms

1) Listen Queue Overflow: Under overload, we have observed that the server application’s listen queue is nearly always full because the rate of attempted connections exceeds the rate at which the application is able to accept them. The server receives a burst of SYNs and responds to all of them with SYN/ACKs, as long as there is space for at least one entry in the listen queue (and the SYN queue is not full). This invariably results in the server receiving more SYN/ACK ACKs than there is space for in the listen queue, leading to a high number of queue drops due to listen queue overflow.

While most TCP stacks are prone to listen queue overflow, its effect in Linux is particularly problematic. Recall that the Linux TCP stack silently drops a SYN/ACK ACK upon a listen queue overflow. It is instructive to study the effect that a silent SYN/ACK ACK drop by the server has on a client. Figure 2 provides `tcpdump` [14] output to illustrate the flow of TCP segments from a client (clnt) whose SYN/ACK ACK triggers listen queue overflow at the server (srvr). Note that we only display the time at which TCP segments were received or transmitted at the server, the end-point identifiers, the TCP flags field (i.e., SYN (S), PSH (P), or FIN (F)), the ACK field, the relative sequence and acknowledgment numbers, and the advertised window sizes.

Upon receiving a SYN/ACK, the client TCP stack sends a SYN/ACK ACK, transitions the connection to the `ESTABLISHED` state, and completes the `connect()` call signifying a successful connection. The client application is now free to start transmitting data on that connection, and therefore sends its request to the server (line 4). However, the client’s SYN/ACK ACK (line 3) results in a listen queue overflow at the server, and is is silently dropped in Linux.

At this point, from the server’s point of view, the connection is still in `SYN_RECV` state awaiting a SYN/ACK ACK, and all subsequent client TCP segments are handled in the `SYN_RECV` code path. Even subsequent data and FIN segments are treated as an implied SYN/ACK ACK, and can result in additional queue drops if the listen queue is full when they arrive at the server. Hence, there is a disconnect between TCP connection state at the client (`ESTABLISHED`) and the server (`SYN_RECV`). The client keeps retransmitting the first segment of its request (lines 5, 6). The retransmission timeout used by the client for data requests tends to be more aggressive than the exponential-backoff style retransmission timeout used for SYN retransmissions because the client has an estimate of the round-trip time after receiving the SYN/ACK. Eventually, the client-side application times out and terminates the connection (line 7).

The Linux TCP stack aggravates the problem by retransmitting a SYN/ACK even when the listen queue is full (lines 10, 13, and 16), thus creating additional load for itself. Moreover, the incomplete client connection continues to occupy space in the SYN queue thereby increasing the probability of additional SYN drops. If the listen queue is not full when a subsequent client segment arrives, the server stack creates a socket and places it on the listen queue. The server application can then accept, read, process, and respond to the client request (lines 19 and 20). However, the client application might have already closed its end of the connection, and hence the client TCP stack sends a reset (RST) in response to every server reply (lines 21 and 22).

Thus, the implementation of connection establishment in

```
(0) 11:32:00.926222 clnt.1024 > srvr.8080: S 1427884078:1427884078(0) win 5840
(1) 11:32:03.688005 clnt.1024 > srvr.8080: S 1427884078:1427884078(0) win 5840
(2) 11:32:03.688026 srvr.8080 > clnt.1024: S 956286498:956286498(0)
    ack 1427884079 win 5792
(3) 11:32:03.688254 clnt.1024 > srvr.8080: . ack 1 win 5840
(4) 11:32:03.688254 clnt.1024 > srvr.8080: P 1:81(80) ack 1 win 5840
(5) 11:32:03.892148 clnt.1024 > srvr.8080: P 1:81(80) ack 1 win 5840
(6) 11:32:04.312178 clnt.1024 > srvr.8080: P 1:81(80) ack 1 win 5840
(7) 11:32:04.688606 clnt.1024 > srvr.8080: F 81:81(0) ack 1 win 5840
(8) 11:32:05.152238 clnt.1024 > srvr.8080: FP 1:81(80) ack 1 win 5840
(9) 11:32:06.832233 clnt.1024 > srvr.8080: FP 1:81(80) ack 1 win 5840
(10) 11:32:07.686446 srvr.8080 > clnt.1024: S 956286498:956286498(0)
     ack 1427884079 win 5792
(11) 11:32:07.686533 clnt.1024 > srvr.8080: . ack 1 win 5840
(12) 11:32:10.192219 clnt.1024 > srvr.8080: FP 1:81(80) ack 1 win 5840
(13) 11:32:13.685533 srvr.8080 > clnt.1024: S 956286498:956286498(0)
     ack 1427884079 win 5792
(14) 11:32:13.685637 clnt.1024 > srvr.8080: . ack 1 win 5840
(15) 11:32:16.912070 clnt.1024 > srvr.8080: FP 1:81(80) ack 1 win 5840
(16) 11:32:25.683711 srvr.8080 > clnt.1024: S 956286498:956286498(0)
     ack 1427884079 win 5792
(17) 11:32:25.683969 clnt.1024 > srvr.8080: . ack 1 win 5840
(18) 11:32:30.352020 clnt.1024 > srvr.8080: FP 1:81(80) ack 1 win 5840
(19) 11:32:30.352122 srvr.8080 > clnt.1024: P 1:1013(1012) ack 82 win 46
(20) 11:32:30.352142 srvr.8080 > clnt.1024: F 1013:1013(0) ack 82 win 46
(21) 11:32:30.352394 clnt.1024 > srvr.8080: R 1427884160:1427884160(0) win 0
(22) 11:32:30.352396 clnt.1024 > srvr.8080: R 1427884160:1427884160(0) win 0
```

Fig. 2. `tcpdump` output to illustrate the problems with the TCP connection management implementation in Linux

many TCP stacks can result in listen queue overflow. The default Linux connection management mechanism, which we will hereby call *`default`*, reacts poorly to listen queue overflow, causing a disconnect between the TCP states at the client and the server as well as unnecessary network traffic.

2) Silent SYN Drop: As shown in Figure 1, most server TCP stacks, including Linux, also drop SYN segments silently if either the SYN or the listen queue is full. Like any other TCP transmission, the client TCP stack sets up a retransmission timeout whenever it sends a SYN segment to the server. If the SYN retransmission timeout expires before a SYN/ACK is received, the client TCP stack retransmits the SYN and sets up another timeout. This process continues until the server responds with a SYN/ACK, or until the threshold for the maximum number of SYN retransmissions is reached. Table I shows the SYN retransmission timeouts used in popular TCP stacks.

Operating System	**SYN retransmission timeout (sec)**
Linux 2.4/2.6	3, 6, 12, 24, 48
FreeBSD 5.3	3, 3.2, 3.2, 3.2, 3.2, 6.2, 12.2, 24.2
Mac OS X 10.3	3, 3, 3, 3, 3, 6, 12, 24
Windows 9x, NT	3, 6, 12
Windows 2000/XP	3, 6

TABLE I

SYN RETRANSMISSIONS TIMEOUTS IN TCP STACKS

An implicit assumption in silent SYN drops is that the connection queues overflowed because of a momentary burst in traffic that will subside in time to allow the retransmitted SYN segments to be processed. However, under persistent overload conditions, such as during flash crowds, the server TCP stack has to handle a large number of connection attempts for an extended period of time. The duration of overload at the server is much longer than the client application timeout, as well as the maximum SYN retransmission timeout used in most client TCP stacks. Hence, the majority of clients retransmit their SYNs while the server is *still* overloaded. As a result the server TCP stack is forced to process retransmitted SYN segments from "old" (i.e., client TCP stacks that are retrying) connections, in addition to a steady rate of incoming SYN requests arising from "new" connections.

We profiled an overloaded web server using a system-wide profiler. Our results indicated that TCP processing dominates CPU utilization compared to other tasks (e.g., driver or IP level packet processing) as the load exerted increases [15]. To illustrate the cost of TCP processing under overload, Table II shows the breakdown of TCP connection related statistics at a web server at different request rates. Note that the server reaches saturation after 22,000 requests/sec.

TCP Statistic	**Request Rate**		
	22,000	**28,000**	**34,000**
Expected SYN segments	2,640,000	3,360,000	4,080,000
Actual SYN segments	2,640,011	4,156,497	6,385,607
Actual SYN/ACK ACK segments	2,640,011	3,323,742	3,828,060
Connection establishment segments	5,280,022	7,480,239	10,213,667
Established connections	2,640,011	2,372,030	2,236,980
Ratio of established connections to actual SYN segments	1.00	0.57	0.35
Total segments	15,840,231	16,968,566	19,161,742

TABLE II

TCP CONNECTION STATISTICS AT A WEB SERVER

Under high loads, the number of SYN and SYN/ACK ACK segments received and processed at the server increases rapidly compared to the number of segments destined for already established connections. The increase in connection establishment segments is primarily due to retransmitted SYN segments. As the server TCP stack spends more time processing SYN segments, the number of connections that it is able to actually establish and process decreases.

Note that the high cost of handling retransmitted SYN segments is not because of a poor implementation of SYN processing in the Linux networking stack. On the contrary, the SYN-processing implementation in most modern TCP stacks, including Linux, is meant to be efficient to minimize the damage from SYN flood attacks [8]. Techniques such as SYN cookies do not alleviate the cost of processing retransmitted connection attempts. Furthermore, dropping SYN segments at an earlier stage in the networking stack does not mitigate the negative performance impact of retransmitted SYN segments [15].

B. Alternative Solutions

1) Solutions for Listen Queue Overflow: Before considering proactive solutions to prevent the listen queue from overflowing, we first review reactive approaches to avoid the disconnect between the TCP states at the end-points arising from listen queue overflow in a Linux server.

Instead of silently dropping a SYN/ACK ACK upon a listen queue overflow, the server TCP stack can send a reset (RST) to a client to notify it of its inability to continue with the connection. A RST ensures that both end-points throw away all information associated with that connection and subsequently there is no TCP traffic on that connection. The client application is notified of a reset connection on its next system call, which will fail with an error such as `ECONNRESET`. The implicit assumption in aborting a connection on queue drops due to listen queue overflow is that the server application is not able to drain the listen queue fast enough to keep up with the rate of incoming SYN/ACK ACKs. This approach to listen queue overflow is used by default in TCP stacks in FreeBSD, HP-UX 11, Solaris 2.7. Although not enabled by default, this behaviour is available as a configurable option in Linux. We will refer to this mechanism as *`abort`* (an abbreviation of the `tcp_abort_on_overflow` parameter that achieves this behaviour in Linux).

Whenever a SYN/ACK ACK triggers a listen queue overflow, the TCP stack can also temporarily grow the listen queue to accommodate it [9]. However, this mechanism disregards the listen queue size specified by the application, thereby overriding any listen queue based admission control policies. The additional connections in the listen queue can also deplete the available kernel memory. Under overload, the server might not be able to accept and process these connections before the client times out, resulting in an inappropriate use of resources. For these reasons, we do not consider growing the listen queue in response to an overflow in this paper.

Proactive approaches that avoid queue drops due to listen queue overflow ensure that the listen queue is never full when a SYN/ACK ACK arrives. We are aware of one such approach, namely, lazy `accept()` [10]. In a lazy `accept()`, the server TCP stack does not send a SYN/ACK in response to a SYN until the server application issues an `accept()` system call. This ensures that the server TCP stack can always accommodate a subsequent SYN/ACK ACK in the listen queue. Although the Solaris 2.2 TCP stack provided a `tcp_eager_listeners` parameter to enable lazy `accept()`, support for this parameter appears to have been discontinued in later Solaris versions [10]. The Windows Sockets API (version 2) provides similar functionality through the `SO_CONDITIONAL_ACCEPT` socket option that can be used with the `WSAAccept()` system call [16]. Note that a lazy `accept()` can result in poor performance because the applications will tend to block waiting for the three-way handshake to be completed by the TCP stack. More importantly, a lazy `accept()` makes the server vulnerable to SYN flood denial of service attacks, where malicious clients do not send a SYN/ACK ACK in response to a SYN/ACK.

Another proactive method of ensuring that there is no listen queue overflow, while still allowing the three-way handshake to be completed before connections are placed on the listen queue, is to implement listen queue reservations. Whenever the server TCP stack sends a SYN/ACK in response to a SYN, it reserves room for that connection in the listen queue (in addition to adding an entry to the SYN queue). Room is freed for subsequent reservations every time an entry is removed from the listen queue through an `accept()` system call. Room can also be created in the listen queue whenever an entry is removed from the SYN queue due to an error or a timeout. The server TCP stack sends a SYN/ACK only if there is room for a new reservation in the listen queue. We refer to the listen queue reservation mechanism as *`reserv`*.

To our knowledge, the *`reserv`* mechanism described above is novel. However, we arrived at it at the same time as the `Dusi_Accept()` call implementation described by Turner et al. for the Direct User Socket Interface within the ETA packet-processing architecture [17]. This call allows applications to post asynchronous connection acceptance requests that reserve room in the listen queue. In contrast to work by Turner et al., our listen queue reservation mechanism is designed to work with the BSD sockets API and existing TCP stack implementations. This *`reserv`* mechanism can be generalized to allow for "overbooking" to account for clients whose SYN/ACK ACK is delayed in WAN environments, and can be combined with a probabilistic SYN drop, SYN cookies, and a periodic clean up of reservations to counter SYN flood denial of service attacks.

2) Alternatives to Silent SYN Drop: When under overload, instead of silently dropping SYN segments, the server TCP stack can explicitly notify clients to stop the further retransmission of SYNs. One way of achieving this is to send a RST whenever a SYN is dropped. Usually, a RST is sent in response to a SYN to indicate that there is no listening socket at the specified port on the server. This is also equivalent to an ICMP "Port Unreachable" message [10]. As per RFC 793, the client TCP stack should then give up on the connection and indicate failure to the client application. Typically upon receiving a RST, a `connect()` call by the application would fail with an error such as `ECONNREFUSED`. Note that a client application is free to continue to retry a connection even after it receives an `ECONNREFUSED` error from its TCP stack, but most well-written applications would give up attempting the connection. Sending a RST when a SYN is dropped, eliminates TCP-level retransmissions, which are transparent to the client application and the user. In order to notify a client when the server is overloaded so that it can prevent further connection attempts (i.e., avoid SYN retransmissions), we modified the server TCP stack in Linux to send a RST whenever a SYN is dropped. We refer to this mechanism as *rst-syn*. (an abbreviation of “RST in response to a SYN on a SYN drop”)

Note that *rst-syn* only affects client behaviour at the SYN stage and is typically used in conjunction with ACK stage mechanisms described in Section III-B.1. That is, we can use *rst-syn* in conjunction with the listen queue reservation mechanism to send RST to clients whenever a reservation is not available, we refer to this mechanism as *reserv rst-syn*. Similarly, when *rst-syn* is used in conjunction with the reactive aborting of connections on listen queue overflow, we call the resulting mechanism *abort rst-syn*. After developing *rst-syn* and discovering the high overhead of processing retransmitted SYNs, we discovered that some TCP stacks, including those on some Windows operating systems, already implement such a mechanism [18]. As indicated earlier, most UNIX TCP stacks drop SYN segments silently. In this context, the results presented in this paper can be viewed as a comparison of the different connection establishment mechanisms implemented in current TCP stacks.

We should point out the shortcomings of *rst-syn*. Sending a RST in response to a SYN when there is a listening socket at the client-specified port implies overloading the semantics of a RST [19]. While it can be argued that from the server’s point-of-view the RST achieves the desired effect of preventing further SYN retransmissions from clients, a client TCP stack which receives a RST from the server is unable to determine whether the RST is because of an incorrectly specified port, or due to overload at the server. Unfortunately, no other alternative seems to be available in current TCP implementations to allow an overloaded server to notify clients to stop retransmitting SYN segments. An approach that could provide such functionality is suggested in RFC 1122 – “A RST segment could contain ASCII text that encoded and explained the cause of the RST” [11]. Unfortunately, this approach would require modifications to existing client TCP stacks.

Another problem with *rst-syn* is that it relies on client cooperation. Some client TCP stacks, notably those on Microsoft Windows, immediately resend a SYN upon getting a RST for a previous SYN[2]. Note that this behaviour does not follow the specifications of RFC 793, which explicitly specifies that a client TCP stack should abort connection attempts upon receiving a RST in response to a SYN. In addition to counteracting *rst-syn*, by retransmitting a SYN upon receiving a RST, Windows TCP stacks introduce unnecessary SYN segments into the network when the client is attempting to connect to a server at a non-existent port. The number of times SYNs are retried in this fashion is a tunable system parameter, which defaults to 2 retries in Windows 2000/XP and 3 retries in Windows NT [20]. Note that in contrast to SYN retransmissions when no SYN/ACKs are received from the server, SYNs are retried immediately by the Windows client TCP stack – no exponential backoff mechanism is used.

[2]Microsoft client TCP stacks ignore “ICMP port unreachable” messages in a similar fashion, retrying a SYN instead [9].

To our knowledge, there are no results that quantify the effects of the Windows-like clients with server TCP stacks which do send a RST to reject a SYN. On the other hand, some past work has cited this behaviour of Windows clients as a justification for dropping SYN segments silently [9], [21]. In this paper, we test this hypothesis by evaluating the effectiveness of *rst-syn* with RFC 793-compliant as well as Windows-like TCP stacks. Unfortunately, our workload generator (`httperf`) is UNIX-specific and non-trivial to port to the Windows API. Hence, we modified the Linux TCP stack in our clients to mimic the behaviour of Windows clients upon receiving a RST in response to a SYN. We will refer to this emulation in Linux as *win-emu*.

Silently dropping SYN segments results in SYN retransmissions in all client TCP stacks. Notifying the client that is should abort SYN retransmissions with a RST is not effective in stopping SYN retries with Windows-like TCP stacks. To workaround this problem, we developed a mechanism which avoids retransmissions of TCP connection establishment segments, albeit in an ungainly fashion. The key idea of this mechanism is to drop no SYN segments, but to instead notify the client of a failure to establish a connection at the ACK stage upon listen queue overflow. Thus, a SYN/ACK is sent in response to every SYN, irrespective of whether there is space available in the SYN queue or the listen queue (SYN cookies are used to simulate an unbounded SYN queue). The *abort* mechanism is later used to reset those connections whose SYN/ACK ACKs cannot be accommodated in the listen queue. We refer to this mechanism as *no-syn-drop*. In order to ensure that there are no SYN drops, we eliminate the check for space availability in the listen queue upon SYN arrival, and use SYN cookies [7] to effectively simulate an unbounded SYN queue.

One limitation of *no-syn-drop*, shared with *abort*, is that we give some clients the false impression that the server can establish a connection. In reality, under overload, the server might be able to accommodate only a portion of the SYN/ACK ACKs received from the clients in its listen queue, and would have to abort those connections that result in listen queue overflow. Upon receiving a RST on an established connection, all client TCP stacks that we are aware of (including Microsoft Windows) immediately cease further transmissions. An `ECONNRESET` error is reported to the application on its next system call on that connection. Most well-written client applications, including all browsers that we are aware of, already handle this error. We reiterate that the only reason for exploring *no-syn-drop* is due to a lack of an appropriate mechanism in current TCP implementations to notify clients (particularly, Windows-like clients) about an overload at the server in order to stop retransmission attempts.

IV. TCP Connection Termination Alternatives

A. Problems with Existing Mechanisms

Supporting half-closed connections can result in an imprudent use of resources at an overloaded server. Many browsers and web crawlers terminate connections, including those connections that timeout or are aborted by the user, by issuing a `close()` system call. That is, they do not use half-closed connection semantics. However, because most server TCP stacks, including Linux, support half-closed connections, they continue to make queued data available to the server application through `read()` calls even after receiving a FIN from a client. Only when the data queue is completely drained will `read()` return `EOF` notifying the server application that the client has terminated the connection. Any prior `read()` call is processed by the application and can result in subsequent writes. The effort spent generating and writing data at the server is wasteful because upon receiving the data the client TCP stack responds with a RST when the half-closed connection semantics are not used for connection termination.

B. Alternative Mechanisms

1) Disabling Support for Half-Closed Connections: We describe a mechanism that can help mitigate the impact of supporting half-closed connections on server throughput under overload. In particular, we are interested in answering the following question – If the server TCP stack were to disable support for half-closed connections and treat all FINs as an indication that the client application is no longer interested in either sending or receiving data on a connection, can we improve server throughput?

Note that we are aware of the potential problems that not supporting half-closed connections can cause. However, others have pointed out that assuming that most clients will not use half-closed connection semantics is not unreasonable [10] and although we are not aware of any modern web browsers that use these semantics, disabling support for all half-closed connections at the server breaks clients that do rely on half-closed semantics. Unfortunately, in current TCP implementations, clients do not provide any indication of whether they are using half-closed semantics in their FIN segments, hence, the server TCP stack cannot selectively disable half-closed connections from some clients. In this paper, we disable support for all half-closed connections in order to assess if the throughput of an overloaded server can be improved by avoiding the imprudent use of resources on connections that clients do not care about. We do not suggest that server TCP stacks should stop supporting half-closed connections entirely. However, support for half-closed connections could be disabled at the server on a per-application basis. For example, a server-side program could issue a `setsockopt()` system call to notify the TCP stack that it should not support half-closed connections on any of its sockets. Alternatively, TCP could be enhanced to allow an end-point to notify its peer that it is not using half-closed semantics while terminating a connection (e.g., through a special set of flags). We perform an experimental evaluation in this paper to determine whether such approaches are worth pursuing.

We refer to our connection termination policy that does not support half-closed connections as *`rst-fin`* (an abbreviation of "RST to FIN"). In this mechanism all FIN segments that do not have piggy-backed data are assumed to indicate that the client is interested in no further read or write activity on that connection, and the server TCP stack takes steps to immediately stop work on that connection. To achieve this, it treats a FIN from the client as if it were a RST, and throws away all information associated with that connection. The server TCP stack then transmits a RST to the client. Note that the transmission of RST in response to a FIN is done directly by the server's TCP stack without any intervention from the server application. The application gets notification of the terminated connection through a `ECONNRESET` error on a subsequent system call on that socket. The handling of RST at the client is completely transparent to the application if the socket descriptor has been destroyed using the `close()` call. Sending a RST in response to a FIN (instead of immediately sending a FIN/ACK) allows clients which are using half-closed connection semantics to be notified of an abnormal connection termination, instead of getting an `EOF` marker indicating that the server has no more data to send. In this way *`rst-fin`* can reduce the amount of time the server spends processing TCP segments on connections that clients do not care about.

2) Connection Termination with Abortive Release: Some browsers such as Internet Explorer 5 and 6 terminate connections through an abortive release [13]. An abortive release implies that the application notifies its TCP stack to abruptly terminate a connection by sending a RST instead of using the two-way FIN/ACK handshake. Applications typically use an abortive release in response to abnormal events (e.g., when a thread associated with the connection crashes). We have observed that Internet Explorer uses an abortive release to terminate all of its connections, whether they represent abnormal behaviour or a routine event. Note that using an abortive release as part of the normal connection termination process is against the recommendations of RFC 793 and RFC 2616. RFC 2616, which describes HTTP 1.1, states, – "When a client or server wishes to timeout it *should* issue a graceful close on the transport connection" [12].

The reasons why Internet Explorer terminates all connections in an abortive fashion are not entirely clear, especially because there are usually enough ephemeral ports at the client to transition connections into the `TIME_WAIT` state. A RST sent by the client does bypass the `CLOSE_WAIT` and `LAST_ACK` TCP states allowing the server TCP stack to transition directly to the `CLOSED` state. Moreover, it also ensures that the server application does not work on connections that have been given up on by the clients. Since Internet Explorer is currently the most popular web browser (used by more than 75% of clients according to some estimates [13]), we try to answer the following question – Does an abortive connection termination by client applications improve server throughput? We use the `SO_LINGER` socket option with `httperf` to emulate the abortive release behaviour of Internet Explorer. We believe that ours is the first (public) attempt to study the impact of abortive release on server throughput. We refer to the abortive release of a connection by a client application as *`close-rst`* (an abbreviation of "client connection close with RST").

It is important to note that both *rst-fin* and *close-rst* can result in lost data if data segments arrive (out of order) on a connection after a FIN or RST, or if there is data pending in the socket buffer that has not been read (or written) by the application when a FIN or a RST is received on a connection. Both of these mechanisms treat a connection termination notification from the client as an indication that the client does not care about the connection, including the potential loss of TCP-acknowledged data that might not have been delivered to the server application. Such an approach is acceptable given the semantics of HTTP GET requests, which are the primary cause of overload in web servers. It might not be appropriate in other application-level protocols or protocol primitives. We reiterate that we take an exploratory approach toward studying the impact of TCP connection termination mechanisms on server throughput, not a prescriptive one.

V. Experimental Methodology

In this paper, we present results obtained with a user-space μserver on a single representative workload. These results were qualitatively similar to those obtained with other servers and workloads we examined [15]. The results of the connection management mechanisms studied in this paper can also apply to other TCP-based, connection-oriented Internet servers which can get overloaded.

Our workload is motivated by a real-world flash crowd event [1], [2]. Sites such as CNN.com and MSNBC were subjected to crippling overload immediately after the September 11th terrorist attacks. The staff at CNN.com responded by replacing their main page with a text-only page sized fit into a single unfragmented IP packet. We devised our one-packet workload to mimic the resulting workload. Each client thus requests a single file using an HTTP 1.1 connection.

All experiments are conducted in an environment consisting of 8 client machines and 1 server machine. We use a 32-bit, 2-way 2.4 GHz Intel Xeon (x86) server, which contains 1 GB of RAM and two Intel PRO/1000 gigabit Ethernet cards. The client machines are identical to the server, and are connected to it through full-duplex, gigabit Ethernet switches. We partition the clients into 2 different subnets to communicate with the server on different network cards. In this paper, we focus on evaluating server performance under the assumption that the server's network bandwidth is not a bottleneck during overload and have configured our experimental setup accordingly. While we perform our evaluation in a LAN, we expect our results to hold in a WAN environment. Techniques such as SYN cookies obviate the large number of incomplete connections in the SYN queue, and WAN-induced delays and packet drops only serve to increase the probability of connection queue overflows in stacks that do not use SYN cookies. Moreover, the server-side overhead of processing retransmitted segments and continuing work on connections that have been terminated by clients persists in a WAN setting.

Our server runs a Linux 2.4.22 kernel in uni-processor mode. This kernel is a vanilla 2.4.22 kernel with its TCP stack modified to implement the different connection management mechanisms, and to collect and report fine-grained statistics during connection establishment and termination. Note that the TCP connection management code that we are concerned with has not changed between the 2.4.22 kernel and the latest 2.6 kernel, and the results of the different connection management mechanisms on the newer kernel are qualitatively similar to those presented in this paper (see [15] for details). We do not modify the default values for networking-related kernel parameters, including the size of the SYN (1024) and listen (128) queues, because a large number of existing production systems operate with default values. We reiterate that increasing queue lengths under persistent overload is not effective because any finite-sized queues will tend to get full.

We use the open-loop `httperf` [22] workload generator to create sustained overload at the server. By using an application-level client timeout, `httperf` ensures that the server receives a continuous rate of requests which is *independent* of its reply rate. The timeout is similar to the behaviour of some web users (or even browsers) who give up on a connection after waiting for some period of time. We use a 10 second timeout in all of our experiments as an approximation of how long a user might be willing to wait, but more importantly it allows our clients to mimic the behaviour of TCP stacks in Windows 2000 and XP (the TCP implementation used by the majority of web clients today [13]) by allowing at most two SYN retransmissions.

In each of our experiments we evaluate server performance with a particular connection establishment and/or termination mechanism. An experiment consists of a series of data points, where each data point corresponds to a particular request rate. For each data point, we run `httperf` for two minutes during which it generates requests on a steady basis at the specified rate. Two minutes proved to be sufficient to allow steady state evaluation of the server.

We evaluate server performance using two metrics, server throughput and client response time. Server throughput reports the mean number of replies per second delivered by the server and measured at the clients. Client response time reports the mean response time measured at the clients for successful connections. It includes the connection establishment time as well as the data transfer time. For both of these statistics, we compute and graph the 95% confidence intervals (which in most cases are small enough that they cannot be seen).

VI. Evaluation

Table III provides a brief summary of the TCP connection establishment and termination mechanisms evaluated in this section. Recall that *reserv*, *no-syn-drop*, and *rst-fin* are novel mechanisms not yet implemented in production TCP stacks. We evaluate connection establishment alternatives with two different types of clients. Clients which follow RFC 793 and abort a connection attempt upon receiving a RST in response to a SYN (as in Linux and most UNIX stacks), we call these *regular* clients. Clients which retransmit a SYN upon receiving a RST in response to a SYN (as in Windows

Type	Name	Description	Implemented in
Establishment	`abort`	Send RST when listen queue overflows upon SYN/ACK ACK arrival	FreeBSD, Solaris, HP-UX, Linux (opt)
	`reserv`	Proactive reservation to avoid listen queue overflows	None
	`abort rst-syn`	Send RST while dropping SYNs in conjunction with `abort`	Windows
	`reserv rst-syn`	Send RST while dropping SYNs in conjunction with `reserv`	None
	`no-syn-drop`	Do not drop valid SYNs, use `abort` upon listen queue overflow for explicit connection rejection	None
Termination	`rst-fin`	Disable support for half-closed connections at server	None
	`close-rst`	Client terminates connections using abortive release	Internet Explorer 5/6
Both	`default`	Default connection management mechanism	Linux

TABLE III

SUMMARY OF TCP CONNECTION MANAGEMENT MECHANISMS EVALUATED

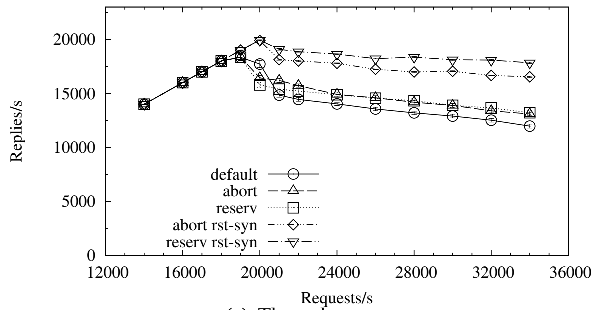
systems), we call these `win-emu` clients because we have modified a Linux stack to emulate the Windows behaviour.

A. Connection Establishment Mechanisms

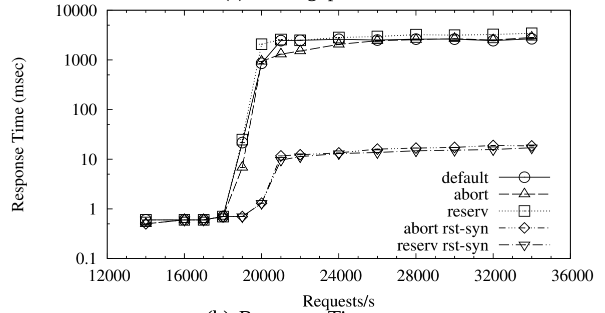
1) Regular clients: Figure 3 shows the results of connection establishment mechanisms with `regular` clients. Figure 3(a) indicates that neither `abort` nor `reserv` are able to provide significant improvements in throughput over `default`. The peak throughput is the same with all three mechanisms. After the peak, `abort` and `reserv` provide less than 10% improvement in throughput over `default`. It is also evident from Figure 3(b) that the response times obtained with `abort` and `reserv` are fairly close to those obtained with `default`. Note that we use a log scale for all the response time results in this paper. The sharp spike in response times after 19,000 requests/sec corresponds to the server's peak (saturation point). After the peak (e.g., at 20,000 requests/sec), many clients have to retransmit a SYN using an exponential backoff before they get a response from the server. Hence, the average connection establishment time and consequently the response time rises sharply. Similarly, the lack of improvement in throughput with `abort` and `reserv` is due to the overhead of processing retransmitted SYN segments at the server.

As shown in Figure 3(a), `abort rst-syn` and `reserv rst-syn` increase the peak throughput by 11% compared to `abort` and `reserv` respectively, and offer around 20-24% improvement after the peak, which increases as the request rate is increased. When compared to `default`, mechanisms using `rst-syn` result in more than 30% improvement in throughput after the peak. Throughput improves with `rst-syn` because it eliminates the cost of processing retransmitted SYN segments, which allows more time to be devoted to completing work on established connections. The gains obtained from `rst-syn` can be explained by the general principle of dropping a connection as early as possible under overload [23].

Moreover, as seen in Figure 3(a), the response time is reduced by two orders of magnitude with `rst-syn`. Recall that response time refers to the average response time measured at clients for successful connections, and it represents the sum of connection establishment time and data transfer time. When the server is overloaded, a substantial amount of time is spent establishing connections. SYNs might have to be retransmitted multiple times before a SYN/ACK is received and the `connect()` call returns on the client. We have observed that under high loads, less than half of all



(a) Throughput



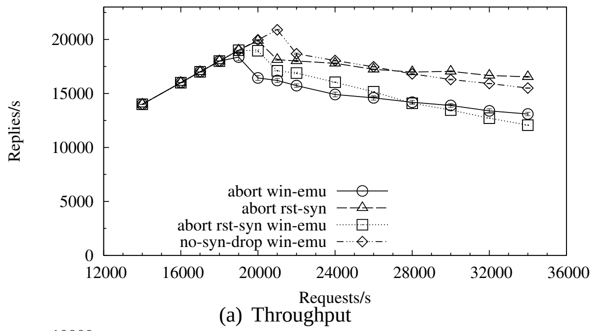
(b) Response Time

Fig. 3. Connection establishment mechanisms with `regular` clients

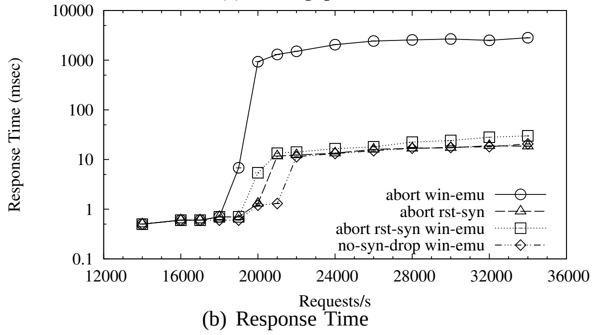
attempted connections receive a SYN/ACK for their first SYN retransmission. Most connection attempts require the transmission of multiple SYNs before they receive a SYN/ACK or time out [15]. On the other hand, with `rst-syn`, a client connection attempt fails on the first SYN transmission. Clients which do get a response for a SYN, receive it without having to resort to SYN retransmissions. As a result, the average connection establishment time, and hence the average client response time tends to be very short. `rst-syn` yields higher throughput as well as lower response time because the server is under persistent overload during flash crowds. While it rejects some clients immediately, the server always has a sustained rate of new clients to handle, so its throughput does not decrease. We believe that under high load, it is better to provide good service to some clients along with an immediate notification of failure to the rest of the clients, rather than giving poor service to all the clients.

Note that the improvements in response times are not because the server handles fewer clients. In fact, both `abort rst-syn` and `reserv rst-syn` actually *reduce* the number of clients which do not receive a server response compared to `default` [15]. Thus, while `abort` and `reserv` by themselves fail to significantly improve server performance under overload, when used in conjunction with `rst-syn`, they are effective in improving server throughput by up to 40% and reducing client response times by more than two orders of magnitude on `regular` clients.

2) Windows-like clients: In this section, we compare the results of `abort rst-syn` and `no-syn-drop` against `abort` on `win-emu` clients. We chose `abort` because it is available in most UNIX TCP stacks, noting that `default` has slightly lower throughput than `abort`. Recall that our `win-emu` clients emulate the behaviour of the Windows client TCP stack in Linux, and retransmit a SYN immediately after receiving a RST for a previous SYN. We also include the results of `abort rst-syn` on `regular` clients (denoted "`abort rst-syn`") for comparison. Figure 4 summarizes these results.



(a) Throughput



(b) Response Time

Fig. 4. Connection establishment mechanisms with `win-emu` clients

Figure 4(a) shows that, as expected, the throughput with `abort rst-syn win-emu` is lower than that with `abort rst-syn` because of the overhead of processing client-initiated SYN retries. However, as shown in Figure 4(b), the client response time is reduced by two orders of magnitude with `abort rst-syn` compared to `abort`, even on `win-emu` clients. Windows TCP stacks resend another SYN immediately upon receiving a RST for a previous SYN. An immediate retransmission ensures that the connection establishment times, and hence, the response times, remain low for clients that receive responses from the server on subsequent SYN transmissions.

By avoiding the retransmission of any connection establishment segments, `no-syn-drop` allows more time to be spent completing work on established connections. As a result, its throughput, at and after the peak, is more than 15% higher than that of `abort win-emu` and `abort rst-syn win-emu`. The throughput and response time resulting from `no-syn-drop` is comparable to that provided by `abort rst-syn` on `regular` clients.

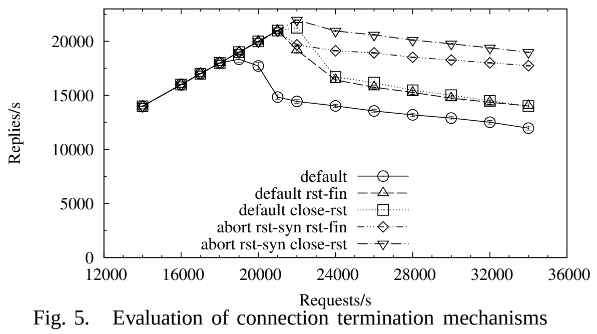


Fig. 5. Evaluation of connection termination mechanisms

To summarize these results in the context of current operating system implementations we find that:

- The server-side connection establishment approaches used in Linux, Solaris, FreeBSD, and most UNIX systems (i.e., `abort` and `default`) do not work well under persistent overload.
- Although the approach used in Windows servers (`abort rst-syn`) provides better response time and throughput with `regular` clients (e.g., those on Linux and most UNIX systems), it fails to improve server throughput with Windows-based (`win-emu`) clients.
- Our `no-syn-drop` mechanism can be used by overloaded servers to work around Windows clients, thereby providing higher throughput and low response times.

B. Connection Termination Mechanisms

Disabling support for half-closed connections only affects server throughput. The response time, which is determined by how long it takes to establish a connection and for the subsequent data transfers is not affected. Hence, we only include throughput results in our evaluation of connection termination mechanisms. Figure 5 shows the impact of `rst-fin` and `close-rst` when used in conjunction with `default`. To study if there is a relationship between connection termination and establishment mechanisms – in particular, to check if one of the connection termination mechanisms can obviate the need for better connection establishment mechanisms (or vice-versa) – we also present the results of `rst-fin` and `close-rst` when used with `abort rst-syn`.

Figure 5 demonstrates that when compared to `default`, `default rst-fin` results in a 17% improvement in the peak throughput. Using `rst-fin` prevents resources from being spent on processing connections that have been timed out by the client at the server. It also allows the TCP connection state to transition directly from `ESTABLISHED` to `CLOSED`. This frees the server from having to process additional ACK segments (in response to FINs), including those on connections that were serviced successfully. For the same reasons, `default close-rst` provides an improvement in throughput over `default`, which is comparable, at and after the peak, to that obtained with `default rst-fin`.

Connection termination mechanisms also complement connection establishment mechanisms. The throughput yielded by both `rst-fin` and `close-rst` increases when they are coupled with `abort rst-syn`. When compared to `default rst-fin`, `abort rst-syn rst-fin` provides more than 20% higher throughput (after peak), and `abort rst-syn close-rst` provides close to 30% higher throughput.

In addition to these results, we have evaluated TCP connection management mechanisms discussed in this paper under a variety of different environments, including SPECweb99 [24] static-content workloads, in-kernel web servers, and transient, bursty load [15]. Given current browser implementations, which issue only a few requests over a single TCP connection [9], we have observed that mechanisms that eliminate the retransmission of connection establishment attempts from clients yield a significant reduction in response times and an increase in server throughput during flash crowds, even on workloads with large transfer sizes. Moreover, even a high-performance kernel-space web server can benefit significantly from using better connection establishment and termination mechanisms. Finally, even under short-lived bursts of traffic, the overall degradation in throughput due to premature rejection of client connection attempts by mechanisms such as `rst-syn` and `no-syn-drop` is negligible. These mechanisms provide at least an order of magnitude reduction in client response times under both bursty load and persistent overload observed during flash crowds, thereby making a strong case for deployment in production TCP stacks.

VII. Related Work

Internet servers have to handle thousands of simultaneous connections. Researchers have proposed novel server architectures [25] for efficiently handling this high level of concurrency. Other researchers have suggested modifications in operating system interfaces and mechanisms for efficiently delivering information about the state of socket and file descriptors to user-space Internet servers [26], reducing the data copied between the kernel and user-space applications [27], and reducing the number of kernel boundary crossings [28]. The connection management mechanisms studied in this paper operate at a lower level and are thus complementary to these application and operating system interface improvements.

Researchers have also sought to improve networking stack implementations [29], [23] and many of the proposed optimizations have found their way into the mainstream operating systems. While research and commercial efforts to offload part or all of the functionality in a TCP stack on a specialized network card are underway, as indicated by Mogul [30], connection management costs are either unsolved or worsened by TCP offloading. Furthermore, the mechanisms discussed in this paper can be implemented in any specialized TCP stack. Protocol-level modifications to improve server performance has also been an area of active research. Nahum et al. [28] describe optimizations to reduce the per-TCP-connection overhead in small HTTP exchanges, including modifications to allow the piggybacking of a FIN on the last data segment, and delaying the acknowledgment of a remote peer's FIN. These optimizations do not alleviate server load resulting from the retransmission of TCP connection establishment segments or preclude server support for half-closed connections and are complementary to the mechanisms studied in this paper.

Following work by Mogul [31], the HTTP 1.1 specification [12] advocated the use of persistent connections to alleviate server load and reduce network congestion arising from the use of a separate TCP connection for every request. Unfortunately, current web clients open multiple simultaneous TCP connections in parallel with a server in order to improve their overall throughput, and reduce the client response time [32], [9]. The average number of requests per connection ranges between 1.2 to 2.7 in popular browsers [9], hence, servers have to handle significantly more TCP connection establishment and termination attempts than envisioned when HTTP 1.1 was introduced. TCP for Transactions (T/TCP) [33] is a backward-compatible extension of TCP for efficient transaction (request-response) oriented service. T/TCP allows protocols such as HTTP to connect to an end-point, send data and close the connection using a single TCP segment. The introduction of persistent connections in HTTP 1.1 as well as its vulnerability to SYN flood denial of service attacks has inhibited the widespread use of T/TCP.

As an alternative to application-level admission control mechanisms, which typically entail at least connection-establishment overhead, Voigt et al. [21] describe a kernel-level mechanism called TCP SYN policing, which uses a token bucket shaper to drop SYN segments silently under overload. The authors rule out the possibility of sending a RST when a SYN is dropped because of its extra overhead, especially with Windows clients. In this paper, we examine this assertion and describe a new connection establishment mechanism that prevents the retransmission of connection attempts, even with Windows clients. The `no-syn-drop` mechanism can be used to enhance the effectiveness of a SYN-policing type approach.

Jamjoom and Shin [9] describe a mechanism called persistent dropping designed to reduce the client-perceived response time during a flash crowd by systematically dropping SYNs (including retransmissions) from randomly chosen clients. In this paper, we examine alternatives to persistent dropping, which allow the server to eliminate client-side retransmission of connection establishment segments, without requiring any server-side state or forgoing fairness. Jamjoom et al. [34] indicate that explicitly rejecting client SYNs can improve client-perceived delays and describe a predictive mechanism for deciding when to drop packets and reject SYNs. Their work assumes the availability of a connection rejection mechanism and their evaluation is based on simulation. We review implementation choices for connection establishment at both SYN and SYN/ACK ACK processing stages, describe a rejection mechanism that requires modifications only to the server-side TCP stack, and present results of its Linux implementation that demonstrate that explicit connection rejection can not only reduce the client response time but also improve the throughput of an overloaded server.

VIII. Conclusion

In this paper, we provide a better understanding of the behaviour of an overloaded server. In particular, we describe some of the reasons why server throughput drops and client response times increase as the load at the server increases. After studying several different connection management mechanisms, we demonstrate that implementation choices for TCP connection establishment and termination can have a significant impact on server throughput and client response times during overload conditions such as flash crowds.

We show that mechanisms implemented at the server TCP stack, which eliminate the TCP-level retransmission of connection attempts by clients during overload, can improve server throughput by up to 40% and reduce client response times by two orders of magnitude. We also describe a new mechanism that prevents the TCP-level retransmission of connection attempts from clients without requiring any modifications to client-side TCP stacks and applications, or server-side applications.

Additionally, we demonstrate that supporting half-closed TCP connections can lead to an imprudent use of resources at an overloaded server when most client applications terminate connections using the half-duplex semantics offered by the `close()` system call. We also examine whether client applications that terminate connections using an abortive release (such as Internet Explorer) affect server throughput. Our results indicate that both an abortive release and precluding support for half-closed TCP connections improve server throughput by more than 15%.

IX. Acknowledgments

We gratefully acknowledge Hewlett-Packard (through the Gelato Federation), the Ontario Research and Development Challenge Fund, and the National Sciences and Engineering Research Council of Canada for financial support for portions of this project. This work has benefited substantially from discussions with David Pariag, Martin Karsten and Srinivasan Keshav.

References

[1] Computer Science and Telecommunications Board, *The Internet Under Crisis Conditions: Learning from September 11*. The National Academies Press, 2003.

[2] V. Padmanabhan and K. Sripanidkulchai, "The case for cooperative networking," in *First International Workshop on Peer-to-Peer Systems*, 2002.

[3] S. Adler, *The Slashdot Effect, An Analysis of Three Internet Publications*, `http://ssadler.phy.bnl.gov/adler/-adler/SAArticles.html`.

[4] I. Ari, B. Hong, E. Miller, S. Brandt, and D. Long, "Modeling, analysis and simulation of flash crowds on the Internet, Tech. Rep. UCSC-CRL-03-15, 2004.

[5] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites," in *Proceedings of the International World Wide Web Conference*, May 2002.

[6] J. Postel, "RFC 793: Transmission Control Protocol," September 1981.

[7] D. Bernstein, *TCP SYN cookies*, `http://cr.yp.to/-syncookies.html`.

[8] J. Lemon, "Resisting SYN flood DoS attacks with a SYN cache," in *BSDCON2002*, 2002.

[9] H. Jamjoom and K. Shin, "Persistent dropping: An efficient control of traffic aggregates," in *Proceedings of ACM SIGCOMM 2003*, August 2003.

[10] W. Stevens, *TCP/IP Illustrated, Volume 1*. Addison Wesley, 1994.

[11] R. Braden, "RFC 1122: Requirements for Internet hosts – communication layers," 1994.

[12] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC 2616: Hypertext Trasfer Protocol – HTTP/1.1," June 1999.

[13] M. Arlitt and C. Williamson, "An analysis of TCP Reset behaviour on the Internet," *SIGCOMM Computer Communication Review*, vol. 35, no. 1, 2005.

[14] V. Jacobson, C. Leres, and S. McCanne, *tcpdump manual*, available at `http://www.tcpdump.org/tcpdump_man.html`.

[15] A. Shukla, "TCP connection management mechanisms for improving internet server performance," Master's thesis, Department of Computer Science, University of Waterloo, July 2005.

[16] Microsoft, *Windows Sockets 2 API – WSAAccept function*, `http://msdn.microsoft.com/library/en-us/-winsock/winsock/wsaaccept_2.asp`.

[17] Y. Turner, T. Brecht, G. Regnier, V. Saletore, G. Janakiraman, and B. Lynn, "Scalable networking for next-generation computing platforms," in *Third Annual Workshop on System Area Networks (SAN-3)*, February 2004.

[18] Microsoft, *Knowledge Base Article 113576 – Winsock App's Reject Connection Requests with Reset Frames*, 2003, `http://support.-microsoft.com/kb/113576/EN-US`.

[19] S. Floyd, "RFC 3360: Inappropriate TCP Resets considered harmful," 2002.

[20] Microsoft, *Knowledge Base Article 175523 – Winsock TCP Connection Performance to Unused Ports*, 2003, `http://support.micro-soft.com/kb/175523/EN-US`.

[21] T. Voigt, R. Tewari, D. Freimuth, and A. Mehra, "Kernel mechanisms for service differentiation in overloaded web servers," in *Proceedings of the USENIX Annual Technical Conference*, June 2001.

[22] D. Mosberger and T. Jin, "httperf: A tool for measuring web server performance," in *First Workshop on Internet Server Performance*, June 1998.

[23] J. Mogul and K. Ramakrishnan, "Eliminating receiver livelock in an interrupt-driven kernel," in *Proceedings of the USENIX Annual Technical Conference*, 1996.

[24] S. P. E. Corporation, *SPECweb99 Benchmark*, `http://www.-specbench.org/web99`.

[25] V. Pai, P. Druschel, and W. Zwaenepoel, "Flash: An efficient and portable web server," in *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[26] G. Banga, J. Mogul, and P. Druschel, "A scalable and explicit event delivery mechanism for UNIX," in *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999.

[27] V. Pai, P. Druschel, and W. Zwaenepoel, "IO-Lite: A unified I/O buffering and caching system," *ACM Transactions on Computer Systems*, vol. 18, 2000.

[28] E. Nahum, T. Barzilai, and D. Kandlur, "Performance issues in WWW servers," *IEEE/ACM Transactions on Networking*, vol. 10, Febuary 2002.

[29] D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of TCP processing overhead," *IEEE Communications Magazine*, June 1989.

[30] J. Mogul, "TCP offload is a dumb idea whose time has come," in *Proceedings of HotOS'03: 9th Workshop on Hot Topics in Operating Systems*, May 2003.

[31] J. Mogul, "The case for persistent-connection HTTP," *SIGCOMM Computer Communication Review*, vol. 25, no. 4, 1995.

[32] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm, and R. H. Katz, "TCP behavior of a busy Internet server: Analysis and improvements," in *INFOCOM*, 1998.

[33] R. Braden, "RFC 1644 - T/TCP – TCP extensions for Transactions, functional specification," 1989.

[34] H. Jamjoom, P. Pillai, and K. Shin, "Resynchronization and controllability of bursty service requests," *IEEE/ACM Transactions on Networking*, vol. 12, no. 4, 2004.