TCP Connection Management Mechanisms for Improving Internet Server Performance

by

Amol Shukla

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2005

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This thesis 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 connection establishment mechanisms in the Linux TCP stack and evaluate their performance in a variety of environments. We also 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. Our evaluation demonstrates that connection establishment mechanisms that eliminate the TCP-level retransmission of connection attempts by clients increase server throughput by up to 40% and reduce client response times by two orders of magnitude. Connection termination mechanisms that preclude support for half-closed connections additionally improve server throughput by up to 18%.

Acknowledgements

First and foremost, I would like to thank my advisor, Tim Brecht, for his guidance, understanding, and support. Tim gave me a free hand with my research, always had time for discussions, and asked all the right questions that provided valuable focus for this thesis. I have learnt a lot from Tim about computer systems and research.

I would like to thank the members of my committee, Martin Karsten and Srinivasan Keshav for their time and perceptive comments, which have improved the quality of this thesis.

I would also like to thank the members of the Networks and Distributed Systems Group for their help, encouragement, and stimulating interactions. In particular, I would like to thank Jay Black, Evan Jones, David Pariag, David Taylor, and Paul Ward for insightful discussions about this thesis.

My research was supported by a postgraduate scholarship from the National Science and Engineering Research Council. I am also grateful for the funding provided by Tim Brecht, the University of Waterloo, the Ontario Research and Development Challenge Fund, and Hewlett Packard.

Finally, I would like to thank my parents and family for their support and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Thesis Statement

The goal of this thesis is to understand the behaviour of overloaded Internet servers, and to improve their throughput and reduce client response times by examining implementation choices for TCP connection establishment and termination.

1.2 Motivation

The demand for Internet-based services has exploded over the last decade. Many organizations use the Internet and particularly the World Wide Web (often referred to as "the web") as their primary medium for communication and business. This phenomenal growth has dramatically increased the performance requirements for Internet servers.

The ubiquitous nature of web browsers has given rise to the occurrence of *flash crowds*[1], 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 arriving at a web site, prolonged periods of overload (i.e., load in excess of the server's capacity), and are often triggered without advance warning. In the hours following the September 11th

[1] The term "flash crowd" was coined by Larry Niven in a science fiction short story, where huge crowds would materialize in places of interesting events with the availability of cheap teleportation [5].

terrorist attacks, many media web 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 [38, 25, 68]. A previously unpopular web site can see a huge influx of requests after being mentioned in well-known newsfeeds or discussion sites, resulting in saturation and unavailability – this is popularly known as the *Slashdot effect* [2].

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. For example, an earthquake monitoring web site will often see significant user traffic only in the aftermath of an earthquake [87]. 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 [5]. Bhatti and Friedrich summarize the case against over-provisioning to protect against flash crowds, "brute force server resource provisioning is not fiscally prudent since no reasonable amount of hardware can guarantee predictable performance for flash crowds" [15].

Web servers thus 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. The client-server interaction on the web is based on HTTP, which uses TCP connections to transfer data between the end-points. Past work [47] 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 has to handle.

In this thesis, we study several different implementation choices for TCP connection management in order to understand their impact on server throughput and client response times under overload. In particular, we examine different approaches to implementing TCP connection establishment (i.e., the three-way handshake) at the server. We also investigate alternatives to the standard four-way TCP connection termination at both end-points. 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 server throughput. Some of the connection management mechanisms examined in this thesis not only increase the server throughput significantly, but they also reduce the client response time[2] by more than two orders of magnitude under overload. While we use web servers to evaluate different connection management mechanisms, the results presented in this thesis also apply to other TCP-based, connection-oriented Internet servers that can get overloaded.

1.3 Contributions

This thesis is concerned with servers that are subjected to overload conditions and makes the following contributions:

1. We provide a better understanding of server behaviour during overload. We describe some of the causes of the drop in server throughput and the increase in client response times.

2. TCP stack developers in Linux, various flavours of UNIX, and Windows have taken different approaches toward implementing connection establishment. It is not clear to a systems programmer, an administrator, or even a TCP stack developer, which of these approaches provide better server performance. We examine different implementation choices for TCP connection establishment, reviewing their advantages and disadvantages. In the process, we demonstrate that the default implementation of connection establishment in the Linux kernel can result in a disconnect between the TCP states at the client and the server, causing unnecessary traffic in the network and at the server.

3. We propose two new connection establishment mechanisms intended to alleviate server traffic under overload. Our mechanisms require modifications only to the TCP stack implementation at the server. No changes are required to the protocol specifications, client-side TCP stacks and applications, or server-side applications. We have implemented both of these alternatives in Linux, along with some of the

[2] Note that we use the term "client response time" in this thesis to denote the average response time measured at the clients.

connection establishment mechanisms currently in place in TCP stacks on UNIX and Windows. We evaluate the performance of these mechanisms in a variety of environments – using two different web servers (a traditional user-space server as well as a high-performance kernel-space server). We demonstrate that mechanisms that eliminate the retransmission of connection attempts by the client-side TCP stacks significantly improve server throughput and reduce client response times by more than two orders of magnitude.

4. We describe a mechanism that allows an early discard of client connection establishment packets under overload. This mechanism relies only on existing data structures available in the Linux TCP stack, does not require an external admission control technique, and ensures that the server is never under-utilized.

5. We evaluate the cost of supporting half-closed TCP connections (which occur when client applications initiate connection termination, for example, on a timeout) on server throughput. We show how supporting half-closed TCP connections can result in an imprudent use of resources at the server and describe a connection termination mechanism that disables the support for half-closed connections. Some web browsers, in particular Internet Explorer, terminate their TCP connections using an abortive release (i.e., by sending a reset (RST) to terminate connections). We examine whether the abortive release of connections by client applications has an impact on server throughput. Our evaluation indicates that disabling support for half-closed connections as well as an abortive release of connections by clients improves server throughput by more than 15%.

1.4 Outline

Chapter 2 provides background information for this thesis and reviews related research. Chapter 3 describes the experimental environment, workloads, and the methodology used in this thesis. Chapter 4 outlines the problems with the existing TCP connection establishment mechanism in Linux. It examines solutions to address these problems, reviews approaches implemented in other TCP stacks, and introduces two novel connection establishment mechanisms. Chapter 4 also examines the alternatives to the existing TCP connection termination mechanism. It includes the evaluation of different connection establishment and termination mechanisms on a workload representative of flash crowds. Chapter 5 presents additional evaluation of some of these mechanisms in different environments, namely, with different workloads, client timeouts, and server platforms, as well as under bursty traffic. It also presents a mechanism that allows an early discard of connection establishment packets from clients during overload. Chapter 6 contains a summary of our findings and outlines some ideas for future work.

Chapter 2

Background and Related Work

In this chapter, we review information required for the rest of this thesis and discuss previous related research. We first describe how TCP connection establishment and termination is currently implemented in most TCP stacks, examining the Linux implementation in detail. We then discuss previous research efforts to improve Internet server performance and address server overload during flash crowds.

2.1 Background – TCP Connection Management

Application-layer Internet protocols such as HTTP use TCP as a reliable transport for exchanging data between the client and the server. 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 connections available in HTTP 1.1. In the following sections, we describe how connection establishment and termination is currently implemented in the Linux TCP stack.

2.1.1 TCP Connection Establishment

TCP is a connection-oriented protocol that requires a connection to be established before end-points can exchange data. TCP connection establishment involves a three-way handshake between the end-points [73, 80]. The handshake has three steps to reduce the possibility of false connections [73][1]. Figure 2.1 illustrates the implementation of the three-way handshake in Linux. We use the implementation of TCP connection establishment in Linux as a representative example, other TCP stacks implement connection establishment in a similar fashion. Note that the TCP connection states at the end-points appear in bold letters in Figure 2.1.

A server-side application, for example, a web server, creates a socket and binds it to a well-known port. It then executes the `listen()` system call to notify the server TCP stack[2] of its willingness to accept incoming connections on the listening socket.

A client-side application, for example, a web browser, initiates a connection by creating a socket and issuing the `connect()` system call. This causes the client TCP stack to send a SYN segment to the specified server. 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). 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, which signifies that both sides have completed connection establishment. 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 by 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. Both sides can thereafter use `read()` and `write()` calls[3] to exchange data. Note that in most socket

[1] As described in RFC 793 [73], "The principle reason for the three-way handshake is to prevent old duplicate connection initiations from causing confusion" with new connection attempts.

[2] The terms "server TCP stack" and "client TCP stack" to used to imply the server-side and client-side functionality provided by the TCP stack in an operating system.

[3] We use `read()` and `write()` to refer to the entire family of system calls to read and write data to/from

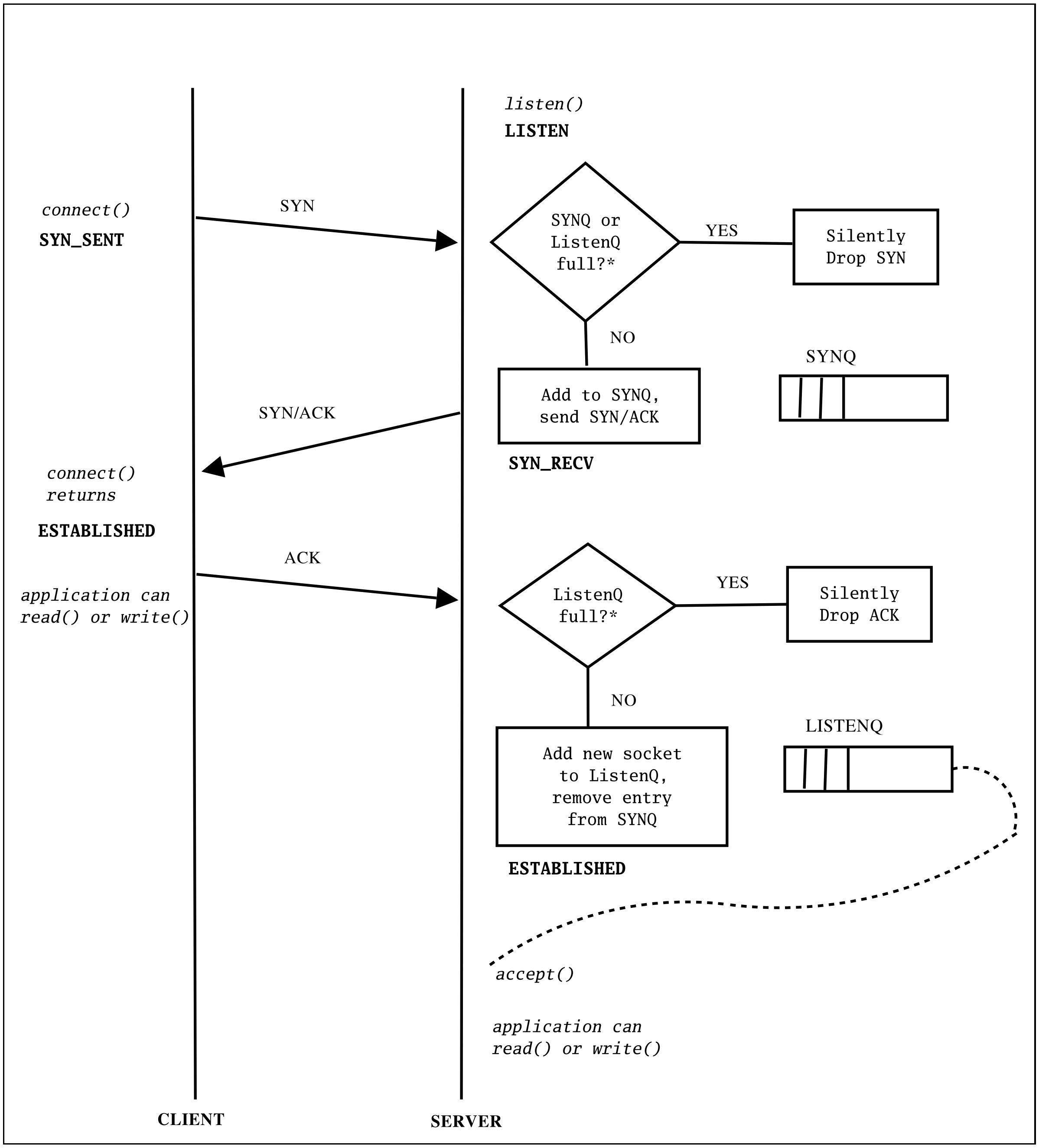


Figure 2.1: TCP connection establishment procedure in Linux

API implementations, the three-way connection establishment procedure is completed by the server TCP stack *before* the application issues an `accept()` call[4].

As shown in Figure 2.1, the server TCP stack uses two separate queues per listening socket to keep track of client segments used for connection establishment. The SYN queue[5] stores information (including the peer's IP address and port number, the TCP options used in the SYN, as well as the timestamp) about incomplete (half-open) connections, which are in the `SYN_RECV` TCP state.

The listen queue contains `ESTABLISHED` connections that are waiting for an `accept()` call from the server application. It is important to note that although we use terms such as "the SYN queue" and "the listen queue" in this thesis, these queues are in fact maintained on a *per-listening-socket* basis in the Linux TCP stack. The sizes of both of these queues are fixed and each entry consumes a portion of the available kernel memory.

Figure 2.2 summarizes the flow of TCP segments used for connection establishment. SYN segments arrive at the server based on the aggregate client connection rate. The server TCP stack creates entries for these connections in the SYN queue and responds with SYN/ACKs (provided there is space in the SYN and listen queues). An entry is normally retained in the SYN queue for the duration of one round trip time (RTT) to the client. That is, entries occupy space in the SYN queue while the server's SYN/ACK and the corresponding SYN/ACK ACK from the client are in transit in the network. Upon receiving a SYN/ACK ACK, the associated connection entry is removed from the SYN queue, and a new socket is created and added to the tail of the listen queue (provided there is space in the listen queue). When the server application issues an `accept()` system call (the frequency of which depends on the application's connection acceptance rate), the entry at the head of the listen queue is removed and a socket descriptor identifying the connection is returned to the application.

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

a socket.

[4] We will discuss the exceptions in Section 4.1.2.

[5] The SYN queue is actually implemented as a hash table to allow for an efficient lookup of the associated incomplete connection when a SYN/ACK arrives – the term "queue" is used for historical reasons.

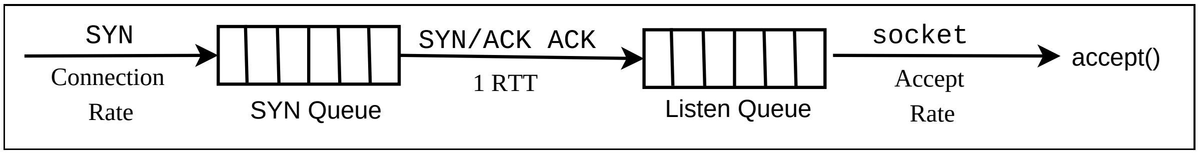


Figure 2.2: Flow of TCP segments used for connection establishment

than the rate at which the server application is able to accept and process new connections (i.e., the server is under overload). A SYN or SYN/ACK ACK segment that cannot be accommodated has to be dropped, we refer to this scenario as a *queue drop*. By our definition, queue drops occur only during the TCP connection establishment phase, either at the SYN stage (i.e., upon receiving a SYN) or the ACK stage (i.e., upon receiving a SYN/ACK ACK). Note the distinction between queue drops and generic packet drops, which may occur at any step during packet processing in the networking stack or even before the packet reaches the server. We now detail the causes of queue drops in the Linux TCP stack.

When a well-formed SYN segment is received (i.e., excluding segments which do not satisfy the TCP Protect Against Wrapped Sequences (PAWS) check, or those with incorrectly set flags such as IP broadcast), queue drops may occur only for one of the following reasons.

1. The SYN queue is full.

2. The listen queue is full and there are at least two entries in the SYN queue whose SYN/ACK timers have not expired.

3. The kernel cannot allocate memory to create an entry in the SYN queue.

4. The SYN queue is three-quarters full and TCP SYN cookies[6] are not enabled.

5. The TCP stack is unable to transmit a SYN/ACK because it cannot find a route to the destination.

We have found that in practice in our environment, SYN segments are dropped under high loads only because of two reasons – the SYN queue is three-quarters full (we refer to

[6] We discuss SYN cookies in Section 4.1.6.

this rule as *SynQ3/4Full*) or the listen queue is full (we refer to this rule as *ListenQFullAtSyn*).

When a well-formed SYN/ACK ACK segment arrives, queue drops may occur only for one of the following reasons.

1. The lookup for a network route to the destination fails.

2. The listen queue is full.

3. The kernel cannot allocate memory while attempting to create a new socket.

In our environment, SYN/ACK ACK segments are dropped under high loads only because the listen queue is full (we refer to this rule as *ListenQOverflow*). Note the distinction between ListenQFullAtSyn and ListenQOverflow – the former rule is enforced at the SYN stage while the later rule is invoked at the ACK stage. The kernel TCP code does not have special names for any of the queue drops. We chose the names for these rules for clarity and we will use them throughout this thesis. The Linux TCP stack takes a conservative approach when the listen queue is full, and tries to drop connection establishment attempts earlier (at the SYN stage through the ListenQFullAtSyn rule) rather than later (at the ACK stage through the ListenQOverflow rule). A similar approach can be found in other open-source TCP stacks such as FreeBSD. The early drop of SYN segments when the SYN queue is only three-quarters full, as opposed to waiting for it to be completely full, is a simple measure against SYN flood attacks in the absence of SYN cookies. While the effectiveness of dropping SYN segments when the SYN queue is just three-quarters full is unclear, the only way to disable it is by modifying the kernel TCP stack code. Since many existing production Linux systems operate without any custom TCP stack modifications, we run all of our experiments without any modifications to the SynQ3/4Full rule. Note that in our discussion of TCP connection establishment, we do not consider the possibility of simultaneous opening of connections by the end-points. In this thesis, we are primarily concerned with a typical client-server environment, such as a web browser interacting with a web server, there are no simultaneous connection openings in this environment.

TCP stack developers in different operating systems have taken different approaches to implementing TCP connection establishment. Most TCP stacks use per-socket SYN and listen queues, and implement rules similar to the ones used in Linux to effect queue drops. Others utilize techniques such as SYN cookies [14] or SYN cache [54] to reduce the state information about incomplete connections stored at the server. However, implementations differ most significantly in how they handle queue drops at the server [45]. 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 [45]. That is, whenever a SYN/ACK ACK is dropped due to ListenQOverflow, a TCP reset (RST) segment is sent to the client notifying it of the server's inability to continue with the connection. In contrast, 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 this thesis, we describe and critique several different approaches to handling queue drops. We are interested in answering the following question – *TCP stack developers in Linux, various flavours of UNIX*[7] *and Windows have taken different approaches to implementing connection establishment. Which of these approaches, if any, result in better Internet server 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.

2.1.2 TCP Connection Termination

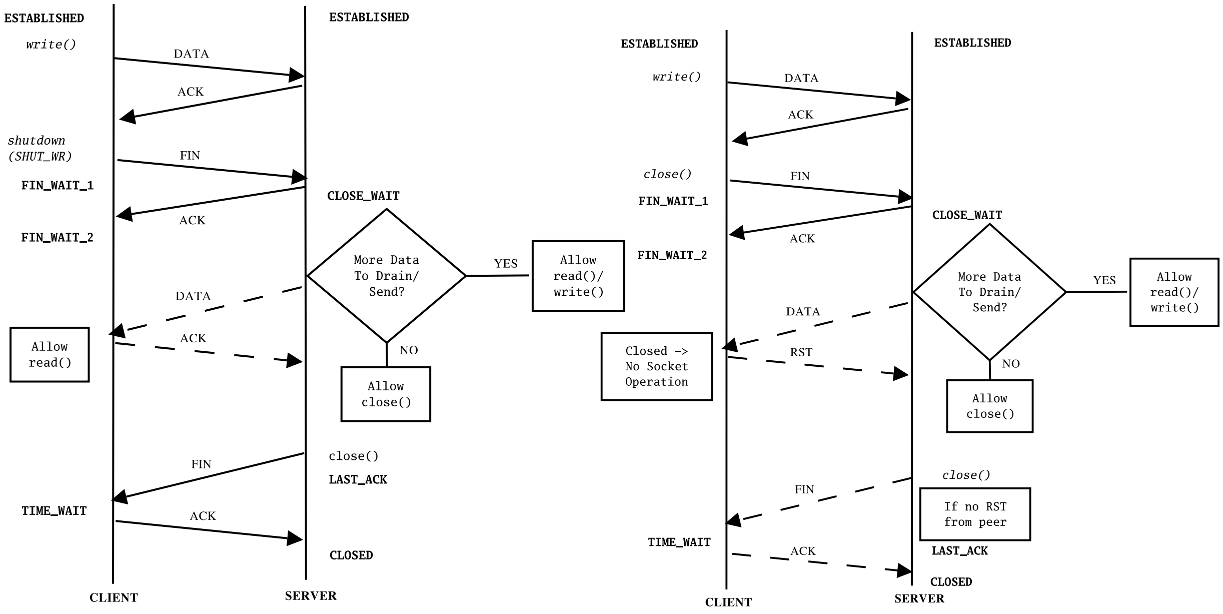
A TCP connection is full-duplex and both sides can terminate their end of the connection independently through a "FIN-ACK" handshake after they finish sending data [73]. That is, an end-point transmits a FIN to indicate that it is not going to *send* any more data on a connection. The other end-point sends an ACK confirming the FIN. This method of connection termination is called "graceful close".

[7] Unless otherwise specified, we refer to the TCP stacks in various commercial as well as open-source UNIX variants as UNIX TCP stacks.

Graceful connection closure is implemented with either half-closed (also called full-duplex) or half-duplex termination semantics. The terms “half-closed” and “half-duplex” are described in RFC 1122 [17]. The CLOSE operation outlined in RFC 793 allows for a connection to be “half closed”, that is, closed in only one direction, allowing an end-point that sends a FIN to *receive* data from its peer in the other direction. 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 to obtain data sent by the peer. Figure 3.1(a) illustrates the half-closed connection termination process.

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 [80]. RFC 1122, which “amends, corrects and supplements” RFC 793, 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.” [17]. Figure 3.1(b) illustrates half-duplex connection termination. A `close()` call typically returns immediately [1] (unless certain SO_LINGER options have been set) and destroys the socket descriptor (assuming no other process holds a reference to it), so the application loses its reference to the TCP connection. Thus, the half-duplex semantics entailed by the `close()` call imply that any data received from a peer after sending a FIN will not be read by the application, but will result in a RST instead.

As shown in Figure 2.3, 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, including Linux, assume that the client uses half-closed semantics. That is, they support half-closed client connections by default. However, most HTTP clients (e.g., web browsers) do not terminate connections using half-closed connection semantics. Instead, they use the `close()` system call to terminate both ends of a connection. Thus, server TCP stacks assume that client applications are using the `shutdown()` system call to terminate connections, however, most client applications in fact use the `close()` system call. In this thesis, we demonstrate how sup-



(a) Half-Closed Termination

(b) Half-Duplex Termination

Figure 2.3: TCP connection termination procedures

porting half-closed connections can result in an imprudent use of resources at the server, especially during overload. The effort spent by the server to generate and write replies after receiving a FIN from a client is wasted when a client application does not read those replies. We present an alternative connection termination mechanism that disables support for half-closed connections. This allows us to evaluate the cost of supporting half-closed connections when the server is overloaded.

Instead of graceful closure, an application can 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 [32] 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 [8]. The reason why Internet Explorer uses an abortive release to terminate connections is not clear. In this thesis, we examine whether the abortive release of connections can improve server throughput under high loads.

2.2 Related Work – Improving Server Performance under Overload

Research in Internet server performance over the years has included application and operating system interface improvements, networking subsystem enhancements, as well as techniques to address server overload during flash crowds. In this section, we review past work in this area as it relates to this thesis.

2.2.1 Application and Operating System Improvements

Modern Internet servers have to handle thousands of simultaneous connections [13]. Researchers have proposed different server architectures for efficiently handling this high level of concurrency. These[8] include – (i) event-driven architectures [70, 19], where connections are multiplexed over a few event-driven server processes (often just a single process), which use an event notification mechanism such as the `select()` system call to work on multiple connections without blocking for I/O, (ii) multi-threaded or multi-process architectures [70, 86], in which the server associates each connection with a thread or a process, and relies on the operating system or a library to continue processing connections that are not blocked, and (iii) hybrid architectures such as the Staged Event Driven Architecture [88]. 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 [13, 22, 53, 29, 67]. Past research has also examined techniques to improve server performance by reducing the data copied between the kernel and user-space applications [69], implementing the transmission of data with zero copying [82, 66], and reducing the number of kernel boundary crossings [66, 67]. In light of the considerable demand placed on operating systems, some researchers have sought to improve the performance of Internet servers by migrating part or all of their functionality into the kernel [84, 39, 46].

[8] This taxonomy of server architectures is provided by Pai et al. [70].

The connection management mechanisms examined in this thesis operate at a lower level (i.e., at the networking stack implementation level) and are complementary to these application and operating system interface improvements. These application and operating system improvements take effect only *after* a connection is established. They also have no impact on connection termination. In fact, we use many of these improvements, which constitute current “best practices”, in our evaluation of different connection management mechanisms.

Event Scheduling in Internet Servers

Recent research has identified the scheduling of events as an important issue in the design of Internet servers. Note the distinction between connection scheduling and event scheduling. Connection scheduling mechanisms seek to minimize the average response time for client connections by applying techniques such as Shortest-Remaining-Processing-Time-first (SRPT) to connections in a user-space server [28] and size-based scheduling in network transmit queues [76]. Ruan and Pai [74] have challenged the effectiveness of these mechanisms, suggesting that connection scheduling opportunities are an artifact of locking and blocking implementations in the operating system. Since connection scheduling takes effect only after a connection has been established and operates at a lower priority than network packet processing, we do not expect it to have any impact on the mechanisms considered in this thesis.

Event scheduling seeks to improve server throughput by maintaining a balance between accepting new connections and making forward progress on existing connections [22, 19, 72, 86]. Such mechanisms typically modify the accept strategy used by Internet servers to drain the listen queue aggressively, whenever the server accepts new connections. By accepting new connections aggressively, servers that are under overload can reduce (but not eliminate) queue drops arising from ListenQFullAtSyn or ListenQOverflow [19, 72]. In our evaluation, we use two web servers configured to aggressively accept new connections. We expect that the connection establishment mechanisms reviewed in this thesis will have an even greater impact on the throughput and the response time provided by servers such as Apache, which do not drain the listen queue aggressively, and consequently have more queue drops (in Linux as well as other UNIX TCP stacks).

It is important to maintain a balance between accepting new connections and completing work on existing connections – a mechanism that accepts new connections too aggressively can result in degraded server throughput. Ostrowski [67] describes a mechanism called *auto-accept*, which operates in the framework of a special kernel subsystem called Scalable Event Notification and Delivery (SEND). In this mechanism, the networking subsystem in the kernel automatically accepts a connection on behalf of an application after the three-way handshake has been completed, without requiring an explicit `accept()` system call. Unfortunately, a strategy such as auto-accept accepts connections too aggressively under overload and causes the kernel to spend most of its time completing the three-way TCP handshake and accepting connections, thereby preventing the server application from making forward progress on accepted connections. The use of auto-accept also results in a virtually unbounded listen queue, ensuring that there are no queue drops due to ListenQOverflow or ListenQFullAtSyn[9]. An unbounded listen queue can create memory pressure at the server because the kernel has to allocate socket entries to track the state of each accepted connection. We believe that the auto-accept mechanism is responsible for the lack of good performance in the SEND subsystem [18]. The size of the listen queue is bounded in most TCP stacks for good reason – as indicated by queueing theory, limiting the queue size acts as an implicit admission control mechanism that ensures that server applications have access to resources to complete work on accepted connections under high loads.

2.2.2 Networking Subsystem Improvements

In this section we review research proposing modifications to the TCP/IP networking subsystem in order to improve Internet server performance. We discuss both implementation changes as well as protocol modifications proposed in past work.

[9] The auto-accept mechanism can thus eliminate all queue drops when used in conjunction with techniques such as SYN cookies, which provide an unbounded SYN queue.

Implementation Improvements

In traditional networking stacks, whenever a network card in the server receives a packet, it signals an (hardware) interrupt, and the operating system runs an interrupt service routine, which is typically part of the device driver. The device driver then creates a buffer (called `skbuff` in Linux and `mbuf` in BSD) encapsulating the packet, places it on the IP queue, and posts a software interrupt. Later on, a kernel thread is run (in the software interrupt context) to pull packets from the IP queue, process them (e.g., reassemble fragmented packets), and pass them on to the transport layer, which performs further processing (e.g., places the data in a TCP segment in an appropriate socket queue). Thus, the protocol processing for an incoming packet occurs at a higher priority than user-space applications (e.g., web servers). This can result in receive livelock [63] under high load, where the server spends all its time processing interrupts, without giving user-space applications any opportunity to run. Receive livelock drives the overall system throughput to zero because no progress can be made on received packets. Various solutions have been proposed to address the receive livelock problem, these include interrupt throttling (also known as interrupt batching or interrupt coalescence), resorting to interrupt-initiated polling during overload [63], and early packet demultiplexing accompanied with protocol packet processing at the priority of the receiving process [30].

Linux implements interrupt-initiated polling through the NAPI (New API) framework [75]. With NAPI, a driver registers a device (e.g., the network card) for work following a hardware interrupt, and disables further interrupts from that device. A kernel thread later polls all registered devices for packets, and enables interrupts only when the device does not have more packets to send[10]. NAPI implements many of the ideas proposed by Mogul and Ramakrishnan [63] including the elimination of the IP queue and an early discard of packets on the network card under overload. NAPI also ensures that other tasks in the system (e.g., the web server) get an opportunity to run by making the polling thread preemptible. We use NAPI in the evaluation in this thesis because it is a well-known solution to prevent receive livelock at an overloaded server. While NAPI obviates the need for

[10] For fairness, each device is allocated a quota on how many packets it can send, however, polling is still used as long as the device has more packets to send.

techniques such as interrupt throttling, our preliminary results using interrupt throttling[11] were qualitatively similar to those using NAPI.

Clark et al. [24] provide insight into why TCP processing can be slow and describe optimizations to make it fast. The improvements outlined include "fast path" processing to optimize the common case of data transfer while a connection is established, TCP input and output header prediction, and low-overhead timer management. Clark et al. suggest that the protocol aspects of TCP do not cause a significant packet-processing overhead. That is, it is not necessary to revise TCP, it is only necessary to implement TCP efficiently to support high performance. Many of the implementation improvements outlined by Clark et al. have been since implemented in production TCP stacks. Kay and Pasquale [50] measure processing overhead of the TCP/IP implementation in Ultrix. They report that the processing time for long TCP messages is dominated by data touching operations (e.g., copying and checksum computation), and propose a checksum redundancy avoidance algorithm to disable checksum computation whenever the link layer provides cyclic redundancy check in LAN environments. They also show that it is difficult to significantly reduce the processing costs for short TCP messages (which are common in HTTP). Processing time for such messages is spread over many non-data-touching operations, whose overhead costs are relatively constant. Their work suggests that it is important to design application-layer protocols to reduce the number of short TCP messages. While Clark et al. and Kay and Pasquale assess the overhead of common operations on *established* connections in TCP implementations, in this thesis, we evaluate the overhead of TCP connection establishment and termination during periods of heavy load such as flash crowds.

Early networking stacks did not handle large number of TCP connections efficiently. McKenney and Dove [57] demonstrated the superiority of hash table based approaches over linked-list implementations for connection state (Protocol Control Block) lookup. The Linux TCP stack uses hash tables to demultiplex incoming TCP segments in a similar fashion. Previous work [31, 9] has reported a degradation in throughput due to the memory and CPU overheads arising from the management of a large number of connections in the `TIME_WAIT` state at the server TCP stack. Faber et al. [31] propose protocol modifications to TCP as well as HTTP to shift the `TIME_WAIT` state to the clients. Aron and Druschel [9]

[11] We used the e1000 device driver. Note that not all drivers support interrupt throttling.

propose TCP implementation modifications to reduce the overhead from connections in the TIME_WAIT state. Note that the server TCP stack has to enter the TIME_WAIT state only for those connections whose termination is initiated by the server application. While the server-initiated termination is typical for HTTP 1.0 connections, HTTP 1.1 connections are typically terminated by the client[12]. In this thesis, we focus on client-initiated connection termination in order to highlight the issues related to supporting half-closed TCP connections at the server and to evaluate the performance impact of an abortive release of connections by client applications. Consequently, there are *no* connections in TIME_WAIT state at the server.

Researchers have argued for offloading part or all of the functionality in a TCP stack on to a network card [4, 62, 36]. Modern network cards include support for the offloading of checksum computation and segmentation for large packet sends. We use checksum offloading in our evaluation and segmentation is not required for our workloads. Freimuth et al. [36] describe an architecture that offloads the entire TCP stack on to the network card, providing the server operating system with a high level interface (i.e., sockets) to the network card. This facilitates large data transfers, eliminates interrupts and bus crossings resulting from "uninteresting" transport-level events (e.g., TCP window updates), and improves memory reference behaviour in the operating system. Turner et al. [83] describe a networking stack architecture to improve the throughput and latency of TCP-based server applications. In their architecture, one or more processor cores in the server are dedicated to packet processing and communication data structures are exposed to applications through an asynchronous interface, allowing them to bypass the operating system for most I/O operations. Kaashoek et al. [48] introduce the notion of server operating systems, which are a set of abstractions and runtime support for specialized, high-performance server applications. Although these alternatives for the networking stack implementation at the server are interesting, the connection management mechanisms studied in this thesis operate at the protocol level and can be easily integrated with an offloaded or a specialized implementation of the TCP stack. We believe that the implementation choices for TCP connection management will continue to have an impact on server throughput and client

[12] However, the server can also terminate connections in HTTP 1.1, for example, when they are idle for a long time.

response times when implemented in dedicated or offloaded packet-processing stacks. As indicated by Mogul [36], connection-management costs are either unsolved or worsened by TCP offloading.

Researchers have proposed modifications to the conventional networking stack and socket API implementations to support differentiated quality of service. These include novel operating system abstractions [12], new operating system architectures [78], networking subsystem improvements [85], and new server architectures [15]. Voigt et al. [85] describe a prioritized listen queue mechanism that supports different service classes by reordering the listen queue based on the priorities assigned to incoming connections. While the aggressive connection acceptance strategy used by our user-space server (which drains the listen queue completely on every `accept()` call) obviates the effect of any reordering, TCP stack or web server implementations could split the listen queue into multiple priority queues [12, 15]. We do not take into account differentiated quality of service in this thesis, treating all client TCP segments uniformly. That is, we assume that the overload at the server is caused by clients who have the same priority. Since the connection management mechanisms discussed in this thesis operate at the protocol level, they can be incorporated into other networking stack implementations that provide better support for differentiated service under overload.

Protocol Improvements

Nahum et al. [66] describe optimizations to reduce the per-TCP-connection overhead in small HTTP exchanges. They propose a modification to system calls that write data to notify the TCP stack of an impending connection termination to allow the piggybacking of a FIN on the last data segment. The client TCP stack can also delay sending a SYN/ACK ACK for 200 ms – within this time window, most HTTP client applications send a data request, on which the SYN/ACK ACK can then be piggybacked. Similarly, a TCP stack can delay the acknowledgement of the remote peer's FIN for 200 ms to allow the piggybacking of its own FIN if the application issues a `close()` call within that time. All of these optimizations reduce the number of TCP segments required for an HTTP transaction, and improve server throughput by up to 18%. However, these optimizations do not alleviate server load resulting from the retransmission of TCP connection establishment segments or preclude server support for half-closed connections. Hence, they complement the implementation choices for TCP connection establishment and termination studied in this thesis. We do note that both abortive connection release as well as our mechanism for disabling support for half-closed connections at the server reduce the number of TCP segments required for HTTP transactions.

Balakrishnan et al. [10] present a detailed analysis of TCP behaviour from traces obtained from a busy web server. They conclude that existing TCP loss recovery mechanisms are not effective in dealing with packet losses arising from short web transfers, and report that web clients use parallel connections aggressively because throughput is positively correlated with the number of connections used. They provide server-side TCP modifications to address these problems, including an enhanced loss recovery mechanism, and an integrated approach to congestion control that treats simultaneous connections from a client as a single unit to improve bandwidth sharing across all clients. Congestion control for established TCP connections is still an area of active research. Congestion control and loss recovery mechanisms are in effect only while a TCP connection is established. The mechanisms studied in this thesis take effect before a TCP connection is established and during its termination.

Following work by Mogul [61], the HTTP 1.1 specification [32] advocates the use of persistent connections in order to alleviate server load as well as reduce network congestion arising from the use of a separate TCP connection for every web request. It recommends that a client application should maintain no more than 2 persistent connections with any server. Unfortunately, as reported by Balakrishnan et al. [10] and Jamjoom and Shin [45], current web clients open multiple simultaneous TCP connections in parallel with a server in order to improve their overall throughput (by obtaining more than their "fair share" of bandwidth), and reduce the client response time (since many web servers do not currently support pipelining). The average number of requests per connection ranges between 1.2 to 2.7 in popular browsers [45], which indicates that current browsers do not issue too many requests on a single HTTP 1.1 connection. Thus, servers have to handle significantly more TCP connection establishment and termination attempts than envisioned when HTTP 1.1 was introduced.

TCP for Transactions (known as T/TCP) [16, 81] is a backwards-compatible extension of TCP for efficient transaction (request-response) oriented service. By using a monotonically increasing connection count (CC) value in the options field of a TCP segment and caching the last CC used at both the end-points, T/TCP allows the three-way handshake mechanism to be bypassed while establishing connections. That is, a minimal T/TCP request-response sequence requires of just 3 TCP segments – (i) a single segment from the client containing the SYN, FIN, CC and the data request, (ii) the server acknowledgement of the SYN, FIN and CC, which also includes its data response and its CC, and (iii) a final segment from the client acknowledging that it has received the server's data and FIN. Thus, T/TCP allows application-layer protocols such as HTTP to connect to an end-point, send data and close the connection using a TCP single segment, without resorting to HTTP 1.1-style persistent connections. The CC option in T/TCP also allows for the truncation of the `TIME_WAIT` TCP state. Unfortunately, T/TCP requires modifications to the TCP stack at both the end-points and has not seen widespread deployment in the Internet. The introduction of persistent connections in HTTP 1.1 as well as vulnerability to SYN flood denial of service attacks due to a lack of compatibility with SYN cookies have inhibited the widespread deployment of T/TCP. The lack of deployment of T/TCP implies that the overhead of TCP connection establishment and termination is still high in current Internet servers.

2.2.3 Addressing Server Overload during Flash Crowds

Many research efforts and commercial products implement admission control to improve server performance during overload conditions such as flash crowds. Cherkasova et al. [23] present a session-based admission control mechanism to ensure longer sessions are completed during overload. Welsh et al. [89] describe adaptive admission control techniques to improve client response time in the Staged Event Driven Architecture [88]. Many web servers return a "server busy" message if their application-level queues get full [40]. Bhatti and Friedrich [15] describe a quality of service aware admission control mechanism that allows an overloaded web server to focus on premium requests by dropping all other requests. Unfortunately, all of these techniques perform admission control at the application-level. That is, a client connection attempt requires copying and processing overhead in the driver, the networking stack, and in some cases in the application, before it is dropped. Hence, these mechanisms often fail to improve the performance of overload servers [85, 40].

More sophisticated web systems use a server farm with a node at the front that performs admission control. Such a front-end node[13] can be a (hardware or software based) load balancer, Layer-4 or Layer-7 switch, or traffic shaper. A number of web sites that face overload due to flash crowds do not use a sophisticated web-farm architecture because their normal load can be easily handled by a single web server. Additionally, most web-farm architectures simply *shift* the burden of addressing overload to the front-end node. The TCP connection management mechanisms examined in this thesis can be deployed at either an overloaded front-end node or an individual web server to enhance system throughput and reduce client response times.

Voigt et al. [85] describe a kernel-level mechanism called TCP SYN policing to allow admission control and service differentiation when the server is overloaded. SYN policing controls the acceptance rate of new connections (i.e., SYN segments) as well as the maximum number of concurrent connections based on connection attributes (e.g., the client IP address) by using a token bucket based policer. SYN segments are dropped silently by the policer. In fact, the authors rule out the possibility of sending a RST when a SYN is dropped because it "incurs unnecessary extra overhead" and because some client TCP stacks (particularly, those on Windows) do not follow the recommendations of RFC 793 and immediately send a new SYN after receiving a RST for a previous SYN. In this thesis, we examine this assertion and evaluate the impact of both silent and explicit (i.e., accompanied with a RST) SYN drops on server throughput and response time with RFC 793-compliant as well as Windows-like client TCP stacks. We demonstrate that sending a RST whenever a SYN is dropped reduces client response times by two orders of magnitude. While it increases server throughput with RFC 793-compliant TCP stacks, sending a RST in response to a SYN fails to improve throughput with Windows-like client TCP stacks, corroborating the hypothesis of Voigt et al. To counter this problem, we describe a new connection establishment mechanism in this thesis that prevents the retransmission of connection attempts, even in Windows client TCP stacks. This mechanism can be used to enhance the effectiveness of SYN policing.

[13] While it is possible to use multiple front-ends with techniques such as round-robin DNS, we discuss a single front-end scenario without loss of generality.

Iyer et al. [40] describe an overload control mechanism that drops incoming requests at the network card. They use a user-space monitoring application that instructs an intelligent network card to drop TCP connection attempts from clients upon detecting overload. In this thesis, we describe an early-discard mechanism that allows connection attempts from clients to be dropped at *any* stage of the networking stack in response to queue drops, without requiring additional user-space monitoring. Unfortunately, we do not have access to a network card that can selectively drop packets. We do believe that our early-discard mechanism can be used to drop packets directly on such an intelligent network card. Iyer et al. also point out that it is currently unclear as to what protocol changes are required in the implementation of TCP and HTTP to allow efficient overload control in Internet servers. In this thesis, we explore this design space at the TCP connection establishment and termination stages.

End-systems or (intermediate network nodes) can apply techniques such as Active Queue Management (e.g., Random Early Drop) [33] or adaptive control [43] to implement admission control during TCP connection establishment. Some of the connection establishment mechanisms proposed and studied in this thesis can provide better system throughput and response time by explicitly notifying clients who are not admitted at an overloaded server. These mechanisms can be used in conjunction with techniques that actually *effect* client admission. Voigt et al. [85] point out a common shortcoming in *predictive* admission control techniques – it is difficult to arrive at optimal control parameters (e.g., load-shedding threshold) that do not under-utilize the server. In contrast to predictive admission control, the mechanisms studied in this thesis rely on queue drops when SYN or SYN/ACK ACK segments are received to implement admission control. That is, they use the SYN and the listen queues in the networking stack to perform *reactive* admission control only when the server is overloaded.

Mahajan et al. [56] introduced the notion of an aggregate to describe the network congestion and high bandwidth utilization resulting from an increase in traffic due to denial of service attacks and flash crowds. They focus on protecting the *network* from congestion resulting from aggregates by proposing two mechanisms – a local aggregate identification and rate-control algorithm, and a push back mechanism to allow a router to request adjacent upstream routers to perform rate limiting on links responsible for the aggregate. In this thesis, we focus on alleviating *end-system* (server) load during flash crowds. Our work complements efforts to protect network links during flash crowds. In fact, some of the connection management mechanisms discussed in this thesis eliminate the introduction of unnecessary packets into the network during flash crowds.

Jamjoom and Shin [45] describe a mechanism called persistent dropping designed to reduce the client delay during a flash crowd. Persistent dropping randomly chooses a portion of SYN segments received from the clients (based on a target reduction rate), and systematically drops them on every retransmission. That is, under overload, a subset of new client SYNs received are dropped with a non-zero probability and all retransmitted client SYNs are *always* dropped. Persistent dropping relies on techniques to distinguish between new and retransmitted SYN segments and requires the server TCP stack to either maintain additional state information about clients[14] or compromise on fairness by discriminating against a fixed set of clients for a length of time. Persistent dropping can be deployed by either routers or end-systems. While it does not seek to improve server (end-system) throughput during flash crowds, persistent dropping does reduce the mean client response times by up to 60%. In this thesis, we examine alternatives to persistent dropping, which allow the server to *prevent* the retransmission of TCP connection establishment segments by clients, without requiring any server-side state and forgoing fairness. We describe a connection establishment mechanism that is not only as effective as persistent dropping in reducing the client response time, it also improves the throughput of an overloaded server.

Jamjoom and Shin propose an ICMP extension called Reject message [44] to improve a server's ability to control the rate of incoming TCP connections. An ICMP Reject message can be sent to clients whose SYN segments are dropped. It contains information to notify a client to abort a connection attempt or to modify its next SYN retransmission timeout. Jamjoom and Shin do not provide an evaluation of this mechanism, in particular, its implication on the throughput and the response time of an overloaded server is unclear. Additionally, before it can take effect, the Reject message extension requires modifications to the ICMP implementation at client networking stacks. To our knowledge, the Reject message extension is not deployed in any production networking stacks. In this thesis, we

[14] Note that storing client state for incoming SYN segments increases server vulnerability to SYN flood denial of service attacks.

study TCP connection establishment mechanisms that can be implemented on server TCP stacks without requiring *any* extensions to existing Internet protocols, modifications to currently deployed TCP stacks at the clients, or changes to client-side or server-side applications. We also provide a detailed evaluation of the impact of several different connection establishment mechanisms, some of which allow the server to notify clients to abort their connection attempts, on server throughput and client response times during overload. The results presented in this thesis do make a strong case for developing a standard mechanism at the client-side socket API to inform client applications of overload at the server (e.g., through an error code such as `ESERVEROVRLD`) upon receiving a notification from the server.

Solutions such as HTTP proxy caching often fail to alleviate load at the server during flash crowds because of the geographical and organizational dispersion of clients [68] and because current cache consistency mechanisms fail to significantly reduce the number of requests that a web server has to handle during periods of extreme user interest [6]. Content Distribution Networks (CDNs) like Akamai [3] provide the infrastructure to ensure good performance and high availability during flash crowds[15]. However, most web sites have enough capacity to handle their normal load and experience flash crowds infrequently. As reported by Ari et al. [5], capacity added to handle flash crowds is more than 82% under-utilized during normal loads, hence, over-provisioning is not a cost-effective solution. It is also unlikely that all web sites can afford the services of a commercial CDN. This thesis studies mechanisms to improve server throughput at such web sites during flash crowds.

Padmanabhan and Sripanidkulchai [68] propose Cooperative Networking (CoopNet), a peer-to-peer caching solution, where end-hosts cooperate to improve performance perceived by all. As with the pseudo-serving approach introduced by Kong and Ghosal [51], CoopNet addresses flash crowds by utilizing clients that have already downloaded content, to serve that content to other clients, thereby alleviating load at the server. This cooperation among clients is invoked only for the duration of a flash crowd, and thus complements the traditional client-server communication. Backslash [79] and Coral cache [35] refine the peer-to-peer content distribution approach by using a distributed hash table overlay for

[15] Incidentally, in the aftermath of the September 11th terrorist attacks, Akamai saw traffic jump 350% above its normal load [25], suggesting that flash crowds can pose a threat to even well-provisioned distribution networks.

automatically replicating content of volunteer sites to alleviate load at the origin server. While pseudo-serving and CoopNet require modifications to existing client applications, Backslash and Coral leverage DNS redirection to work with existing web browsers. Note that all of these peer-to-peer systems require web sites to participate *before* a flash crowd occurs. Many web sites may not use peer-to-peer solutions for economic reasons (e.g., loss of ad revenue) or due to lack of awareness. Unauthorized mirroring of content may constitute a breach of copyright and might inhibit the widespread deployment of these solutions. In this thesis, we focus on trying to improve server throughput and response time in the traditional client-server communication model that is currently pervasive in the Internet. The connection management mechanisms discussed in this thesis can also improve the performance of overloaded content distribution networks.

Both flash crowds and denial of service attacks are characterized by a large surge in traffic and result in overload at the server. While flash crowds are caused by legitimate clients, who seek "interesting" content at a web site, denial of service attacks are caused by malicious attackers, and represent "an explicit attempt by attackers to prevent legitimate users of a service from using that service" [21]. A server should handle as many requests as possible during flash crowds, but need not handle requests originating from denial of service attacks. Recent work has presented techniques to distinguish between flash crowds and denial of service attacks. Jung et al. [47] study the characteristics of flash crowds and denial of service attacks, and propose a network-aware clustering mechanism to identify and drop packets (including TCP SYN segments) from IP addresses that are likely to represent malicious clients. Kandula et al. [49] present a kernel extension called Kill-Bots to protect servers from denial of service attacks that masquerade as flash crowds. Kill-Bots uses graphical tests[16] to identify and block IP addresses of the attack machines. In addition to authentication, the Kill-Bots work also examines admission control in the context of malicious clients. Although Kill-Bots modifies the TCP stack to avoid storing any connection state related to unauthenticated clients, the authentication procedure is implemented *after* a TCP connection is established[17]. In this thesis, we demonstrate

[16] Graphical puzzles called CAPTCHAs are also used by free web mail providers for identifying human users while creating accounts.

[17] Note that two TCP connections need to be established for every new HTTP request from an unauthenticated legitimate client in Kill-Bots, thereby increasing the connection management overhead.

that some implementation choices for TCP connection establishment can improve server throughput during overload. While our evaluation focuses on flash crowds (i.e., legitimate requests), the mechanisms examined can be used along with techniques such as network-aware clustering or Kill-Bots for admission control, authentication, or for rejecting requests from malicious clients during denial of service attacks.

In the next chapter, we describe the experimental environment and the methodology used in this thesis.

Chapter 3

Experimental Environment, Workloads, and Methodology

In this chapter we describe the hardware and software environment, including the web servers, used in our evaluation. We also discuss the workloads, the overload model, and the methodology used in this thesis to evaluate different TCP connection management mechanisms.

3.1 Experimental Environment

3.1.1 Web Servers

We use web (HTTP) servers to illustrate the impact that implementation choices for TCP connection management can have on server throughput and response time under overload. Web servers form the front-end of many Internet applications and are likely to bear the brunt of heavy traffic during flash crowds. Note that the results of the connection management mechanisms studied in this thesis can also apply to other TCP-based, connection-oriented Internet servers which can get overloaded, such as LDAP servers.

We use two different web servers in our evaluation – the μserver, an event-driven, user-space web server, and TUX, a kernel-space web server.

The μserver

We use the μserver [52, 19] to represent current high-performance, user-space web servers. The μserver is a single process event-driven web server designed to facilitate research on the impact of operating system design and implementation on server performance and scalability.

While the μserver can support multiple event notification mechanisms, in our experiments we use the `select()` system call to obtain events from the operating system. We also use the *Accept-INF* [19] option while accepting new connections. That is, whenever `select()` indicates that an `accept()` call can complete without blocking, the μserver keeps issuing `accept()` calls in a tight loop until one of them fails, thereby draining the listen queue entirely. Recent work [19, 77, 37] has demonstrated that a `select()`-based μserver using the *Accept-INF* option can withstand sustained overload and yield throughput that rivals that provided by specialized, in-kernel web servers.

We use the μserver in our experiments because it provides better performance than existing thread or process-per-connection, user-space web servers such as Apache. In fact, our preliminary experiments showed that the throughput with Apache was more than five times lower than that obtained with the μserver[1]. Apache uses multiple threads or processes, each of which accepts a connection and processes it to completion before accepting the next connection. We expect that the TCP connection management mechanisms studied in this thesis will have an even greater impact on the performance of servers like Apache that are not as aggressive as the μserver in accepting new connections.

TUX

We use the in-kernel TUX [39, 55] web server (also called the Redhat Content Accelerator) in some of our experiments. TUX has been reported to be the fastest web server on Linux [46]. By running in the kernel, TUX eliminates the cost of event notification (or scheduling), kernel boundary crossings, and redundant data buffering, which traditional user-space servers incur. More importantly, TUX has direct access to kernel-level data structures such as the server socket's listen queue, and is closely integrated with the kernel's

[1] We used the one-packet workload for this evaluation.

networking stack and file system cache. This allows it to implement several optimizations such as zero-copy request parsing, zero-copy disk reads, and zero-copy network writes. Note that we use the *Accept-1* [19] option added to TUX by Brecht et al. [19] in order to maintain a balance between accepting new connections and completing work on existing connections. In this configuration, TUX accepts a single connection from the listen queue and works on it until it is completed or until it blocks. This is different than the default approach of draining the listen queue completely and then working on all the accepted connections. Pariag [72] and Brecht et al. [19] have demonstrated that with the *Accept-1* option, TUX can provide higher throughput and lower response time compared to its default connection acceptance mechanism.

We use TUX in our experiments to demonstrate that implementation choices for TCP connection management can have a significant impact on the performance of an overloaded server, even if the server has been streamlined and optimized to perform a single task efficiently. In particular, we believe that some of the TCP connection establishment mechanisms discussed in this thesis can be deployed in Layer 4 or 7 switches, which perform admission control for a web server farm, to improve their throughput and reduce client response times during flash crowds. TUX provides an approximation of the potential effectiveness of these mechanisms in improving the throughput and reducing the response time in such specialized switches.

Server Configuration

As described earlier, we use the `select`-based μserver (version 0.5.1-pre-03) with the *Accept-INF* option. While we use the zero-copy `sendfile()` system call for the SPECweb99-like workloads in the μserver, for the one-packet workload we use the `writev()` system call to transmit data as well as HTTP headers from an application-level cache because it results in better performance due to the absence of `setsockopt()` calls to cork and uncork a socket [71, 66]. We use TUX (kernel module version 2) with a single thread, as recommended in its users manual [39], enhanced with the *Accept-1* option as described earlier.

Both the μserver and TUX are configured to use a maximum of 15,000 simultaneous connections, however, we never reach this maximum limit. Logging is disabled on both the servers to prevent disk writes from confounding our results. Note that in this thesis we focus on using the μserver and TUX to evaluate several different implementation choices for TCP connection management. Unlike earlier work [19, 72, 77], we do not seek to present a direct comparison between the two web servers.

SYN and Listen Queue Sizes

The size of the SYN queue can be configured through the `net.ipv4.tcp_max_syn_backlog` sysctl (a sysctl is an administrator-configurable parameter in Linux). Its default size is 1024 for systems with more than 256 MB of memory, 128 for those systems with less than 32 MB of memory, and 256 otherwise. The size of the listen queue is determined by the value specified for the "backlog" parameter to the `listen` system call. Historically, Internet servers have used a backlog value between 5 and 8 [80]. Recent TCP stacks allow applications to use a larger backlog value to buffer more connections between `accept()` calls. The Linux 2.4 TCP stack silently limits the backlog value specified by the application to 128. The Linux 2.6 stack allows this maximum backlog value to be modified through the `net.core.somaxconn` sysctl, but continues to use a default maximum backlog of 128. Thus, the *de facto* listen queue size in Linux TCP stacks is at most 128[2]. The listen queue size for the in-kernel TUX web server can be set through the `net.tux.max_backlog` sysctl, which defaults to 2048.

For our experiments, we use the default values for the SYN queue (1024) and the listen queue (128 for μserver, 2048 for TUX) because of the large number of existing production systems that operate with these limits. While it is possible that using different values for these queues might influence server throughput during short bursts of high load, for the persistent overload scenarios (such as flash crowds) that we study in this thesis, it is sufficient to note that these queues are of a fixed size and will overflow during sustained periods of high load. Note that Arlitt and Williamson have reported that the size of the listen queue does not have a significant impact on server performance under high load [7]. We also conducted some preliminary experiments under persistent overload with different queue sizes and found no qualitative differences in server throughput obtained with the different connection establishment mechanisms studied in this thesis. As expected from

[2] Unless configured otherwise.

queueing theory, using a larger queue does increase the average client response time.

3.1.2 Machine and Network Setup

All experiments are conducted in an environment consisting of eight client machines and one server machine. We use a 32-bit, 2-way 2.4 GHz Intel Xeon (x86) server in most of our experiments. This machine contains 8 KB of L1 cache, 512 KB of L2 cache, 1 GB of RAM, and two Intel e1000 gigabit Ethernet cards. The client machines are identical to the server, and are connected to it through a full-duplex, gigabit Ethernet switch. We partition the clients into two different subnets to communicate with the server on different network cards. That is, the first four clients are linked to the server's first network card, while the remaining four clients communicate with the server using a different subnet on its second network card. In this thesis, we focus on evaluating server performance under the assumption that the upstream and downstream bandwidth of the server's network link is not the bottleneck during overload. Hence, we have configured our experimental setup to ensure that the network bandwidth is not the bottleneck in any of our experiments.

Our server runs a custom Linux 2.4.22 kernel in uni-processor mode. This kernel is a vanilla 2.4.22 kernel with its TCP stack modified to support the different connection establishment and termination mechanisms studied in this thesis. We use a 2.4 Linux kernel because of difficulties (i.e., kernel panics and server bugs) running TUX with the newer 2.6 kernel. Note that the TCP connection management code that we are concerned with in this thesis has not changed between the 2.4.22 kernel and the latest 2.6 kernel (2.6.11.6). The results of the different connection management mechanisms on a 2.6 kernel will be qualitatively similar to those presented in this thesis. To demonstrate this, we show some experimental results using a 2.6.11 kernel in Section 5.5. All of our clients run a Redhat 9.0 distribution with its 2.4.20-8 Linux kernel.

For our experiments in Section 5.5, we use an HP rx2600 server with a different processor architecture – a 64-bit, 2-way 900 MHz Itanium 2 (ia-64). This machine has 32 KB of L1 cache, 256 KB of L2 cache, 1.5 MB of L3 cache, 4 GB of RAM, and two Intel e1000 gigabit Ethernet cards. The rest of the client and network setup used for the Itanium 2 experiments remains unchanged.

We do not modify the default values for other networking-related kernel parameters such as send and receive socket buffer sizes, TCP maximum retransmission limits, transmit queue lengths, or driver-specific parameters on our servers (or our clients). A large number of production server machines run “out of the box” without modifying the default configuration. Running our experiments in the same fashion increases the applicability of our results. More importantly, changing the default values (without having a good reason to do so) unnecessarily increases the parameter space that must be explored in the experiments.

We have instrumented the TCP stack in our server kernels to report detailed statistics during connection establishment and termination. The Linux kernel only tracks the number of failed connection attempts (total queue drops), and the number of listen queue overflows (ListenQOverflow-induced queue drops). We added fine-grained counters to track all the causes of queue drops, including those due to SynQ3/4Full and ListenQFullAtSyn[3]. Although these counters are not tracked by the Linux kernel by default, the overhead added by our additions is minimal. These detailed statistics are available through the standard kernel SNMP interface and can be obtained through programs such as `netstat`.

3.2 Workloads and Overload Model

We use two different workloads to evaluate the impact of TCP connection management mechanisms on server performance. Note that both of our workloads involve requests for static files that can be served completely from the in-memory file system cache. During a flash crowd, clients are typically interested in a very small subset of a web site [25, 87], which is cached by the file system after the first few requests. To alleviate server load during flash crowds, web administrators often replace pages that require the generation of dynamic content with static pages [25, 68]. Our first workload is inspired by a real-world flash crowd, while our second workload is based on the popular SPECweb99 benchmark [26]. In this section, in addition to discussing both these workloads in detail, we will also describe our workload generator (which is able to generate overload conditions at the server using a few client machines), and our overload model.

[3] All rules that cause queue drops are described in Section 2.1.1

3.2.1 One-packet Workload

Our first workload is motivated by real-world flash crowd events [25, 68]. Media sites such as CNN.com and MSNBC were subjected to crippling overload immediately after the September 11th terrorist attacks [25], pushing site availability down to 0% and response time to over 47 seconds [38]. The staff at CNN.com responded by replacing their main page with a small, text-only page, sized to allow the server reply (including the headers) to fit into a single unfragmented IP packet. We devised our one-packet workload to mimic the load experienced by CNN.com on September 11th. Each client requests the same one-packet file using an HTTP 1.1 connection. Note that every client connection consists of a *single* request.

We believe that the one-packet workload also highlights other aspects of load experienced by real-world web servers that are typically ignored by benchmarks such as SPECweb99. Nahum [65] compares the characteristics of SPECweb99 with data traces gathered from several real-world web server logs. His analysis points out some important shortcomings of the SPECweb99 benchmark. In particular, SPECweb99 does not take into account conditional GET requests, it underestimates the number of HTTP 1.0 requests, and overestimates the average file transfer size.

A conditional GET request contains an "If-Modified-Since" header. If the requested file has not been modified since the value specified in the header, the server returns an `HTTP 304 Not Modified` header with zero bytes of (file) data. This response can easily fit into a single IP packet. Nahum reports that up to 28% of all requests in some server traces consist of single packet transfers due to conditional GET requests. He also reports that the proportion of HTTP 1.0 connections used in SPECweb99, namely, 30%, is much smaller than that seen in his traces. HTTP 1.1, standardized in 1997, added support for persistent connections, where multiple requests can be made over a single TCP connection. However, even in some of Nahum's traces obtained in 2000 and 2001, up to 50% of the requests originated from a HTTP 1.0 connection. Finally, the median transfer size in these traces is much smaller than that used in SPECweb99 and is small enough to fit in a single IP packet.

Thus, our one-packet workload is derived from measures taken by CNN.com to address a flash crowd. Having the server deliver a single small file from its file system cache is perhaps the best way to serve the most clients under extreme overload. It also incorporates a number of characteristics reported by Nahum in real-world web traces, which are ignored by popular web benchmarks. Past work characterizing flash crowds [47] has reported that the increase in server traffic occurs largely because of an increase in the number of clients, which causes the number of TCP connections that a server has to handle to escalate. The connection-oriented nature of the one-packet workload, which arises from using a single request per HTTP connection, stresses the server's networking subsystem, thereby highlighting the impact of the implementation choices for TCP connection management.

3.2.2 SPECweb99-like Workload

The SPECweb99 benchmark [26] has become the *de facto* standard for evaluating web server performance [65]. Developed by the Standard Performance Evaluation Corporation (SPEC), its workload model is based on server logs taken from several popular Internet servers and some smaller web sites. The benchmark provides a workload generator that can generate static as well as dynamic requests. The later constitute 30% of all requests [27].

As described in Section 3.2.1, during flash crowds, web servers have to handle primarily a static, in-memory workload. To ensure that our workload remains representative of real-world overload scenarios, we carefully constructed an HTTP trace that mimics the static portion of the SPECweb99 workload. We use one directory[4] from the SPECweb99 file set, which contains 36 files and occupies 4.8 MB. The entire file set can be easily cached and no disk I/O is required once the cache has been populated. Our trace contains HTTP 1.1 connections with one or more requests per connection, where each connection represents an HTTP session. The trace recreates the file classes and the Zipf-distribution access patterns that are required by the SPECweb99 specifications. We refer to the resulting workload as SPECweb99-like.

30% of connections used in SPECweb99 are HTTP 1.0, and the number of requests per connection for the remaining 70% of HTTP 1.1 connections range from 5 to 15, with an average of 7.2. As described in Section 2.2.2, many popular web browsers use parallel connections for their requests in order to improve client throughput and minimize the

[4] Note that all directories in the SPECweb99 file set are identical with respect to the structure and sizes of their files.

response time. This is in contrast to using a single sequential persistent connection to fetch embedded objects modelled in SPECweb99. Jamjoom and Shin [45] report that the average number of requests per connection issued in current browsers is between 1.2 and 2.7. In particular, Internet Explorer, by far the most popular web browser today [8], uses an average of 2.7 requests per connection. This is significantly lower than the average of 7.2 requests per connection used in SPECweb99.

Thus, SPECweb99 underestimates the rate of new connection attempts seen at busy web servers. To address this problem, we use three different values for average requests per connection with our SPECweb99-like workload, namely, 1.0, 2.62[5], and 7.2. That is, we use three different SPECweb99-like workloads each of which follow the file classes and access patterns specified in SPECweb99, but use sessions that on average have a single request per connection, 2.62 requests per connection and 7.2 requests per connection. Like the one-packet workload, the single-request-per-connection SPECweb99-like workload is connection-oriented, and can be seen as an extension of the one-packet workload that uses different file sizes in its requests. The mean size of files requested in all of our SPECweb99-like workloads is around 15 KB. The 2.62-requests-per-connection SPECweb99-like workload seeks to be representative of the behaviour of current web browsers, while the 7.2-requests-per-connection SPECweb99-like workload meets most[6] of the SPECweb99 specifications for static content.

We use the one-packet workload for most of our experiments in Chapters 4 and 5 because its connection-oriented nature highlights the performance impact of the different TCP connection management mechanisms. We also believe that it is more representative of the loads handled by web servers during flash crowds, where most of the clients are interested in a single "hot" page (e.g., a breaking news story or a live game score), and the server administrators have taken steps to ensure that all the "hot" content is delivered as quickly and efficiently as possible. We examine the impact of the SPECweb99-like workloads on the implementation choices for TCP connection management in Section 5.1.

[5] With our SPECweb99-like trace generator, an average of 2.62 requests per connection was the closest that we could come to the average of 2.7 requests per connection used in Internet Explorer.

[6] We use HTTP 1.1 for connections consisting of a single request instead of HTTP 1.0 as specified in SPECweb99 in order to ensure that all connection terminations are initiated by the clients.

3.2.3 Workload Generator

The workload generator provided with the SPECweb99 benchmark operates in a closed loop. That is, each client modelled in the generator will only send a new request once the server has replied to its previous request. Banga and Druschel [11] demonstrate that such a naive load generation scheme allows the rate of requests made by the generator to be throttled by the server. Even though more than one client can be run on a machine, many such clients that are blocked waiting for a response from the server can cause that machine to run out of resources such as available ephemeral TCP ports or file descriptors. Without resources at the client machine, additional connections cannot be attempted, hence, a closed-loop workload generator is unable to maintain sustained load that can exceed the capacity of the server. While it is possible to use a large number of client machines to create overload conditions, Banga and Druschel show that several thousand closed-loop generators (and hence client machines) would be needed to generate sustained overload.

We address the inability of the SPECweb99 generator to produce overload at the server by using `httperf` [64], an open-loop workload generator in conjunction with our SPECweb99-like traces. Open-loop generators such as `httperf` induce overload by implementing connection timeouts. Every time a connection to the server is attempted a timer is started. If the timer expires before a connection can be established, the connection is aborted so that a new one can be attempted in its place. If the connection does get established, the timer is restarted for every request issued and the connection is closed on a timeout. A connection is deemed successful if all of its requests are served without a timeout. By using an application-level timeout, `httperf` is able to bound the resources required by a client. This ensures that the server receives a continuous rate of requests which is *independent* of its reply rate. The timeout used in `httperf` is similar to the behaviour of some web users (or even browsers) who give up on a connection if they do not receive a response from the server within a reasonable amount of time.

By running in an event-driven fashion, only a single copy of `httperf` is needed per client CPU to generate sustained overload. In addition to the throughput and response time statistics for successful clients, `httperf` tracks the number of timed out and server-aborted[7] connections as errors. While it is possible that there are other causes for errors

[7] Connections for which a RST is received from the server, either prior to or after connection establish-

measured at the client (e.g., running out of ephemeral ports) we have carefully monitored our tests to ensure that all `httperf` errors are due to timeouts or connections aborted by the server.

We use a 10 second timeout in all of our experiments. With our Linux clients, this allows a SYN segment to be retransmitted at most twice when no response is received from the server, before a connection is aborted by the client. A 10 second timeout is chosen 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[8], which also retransmit a SYN at most twice. We examine the impact of using different timeouts in Section 5.3.

3.2.4 Overload Model

We use a *persistent* overload[9] model that simulates flash crowds in our evaluation of different connection management mechanisms. Under persistent overload, a server has to try to handle loads well in excess of its capacity for an extended period of time. That is, the duration of sustained overload at the server is much longer than that of any interim periods of light load. In contrast, a *bursty traffic* model assumes that overload conditions are transient, with short-lived bursts of heavy traffic (overload bursts) interrupting protracted periods of light load at the server. Figure 3.1 illustrates the difference between these overload models.

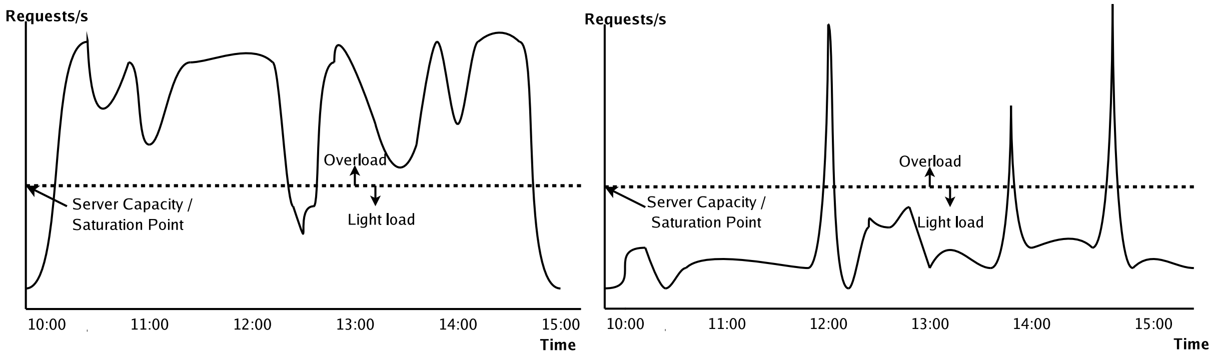
Our persistent overload model assumes that a server has to handle sustained overload for a length of time measured in minutes or hours. This is not an unreasonable assumption – past flash crowds have reported prolonged periods of overload at the server, which last for many hours, sometimes even for a day [25, 68, 87, 47]. Accordingly, flash crowd simulation models [5] often assume a rapid spike in load at the server at the beginning of the flash crowd, a continuous stream of incoming requests that saturate the server for a protracted period, followed by a sudden drop in load that marks the end of the flash crowd.

Persistent overload does not imply that there is only a single long-lasting period of

ment.

[8] The TCP implementation used by a majority of web clients today [8].

[9] The term “persistent overload” is due to Mahajan et al. [56].



(a) Persistent Overload

(b) Bursty Traffic

Figure 3.1: Comparison of Overload Models

overload at the server. It is possible to have multiple periods of overload – similar to the multiple shock waves model described by Ari et al. [5] – as long as their duration is much longer than the duration of the interim periods of light load. Note that persistent overload represents flash crowds, it does not imply that the server is perpetually under overload. An "always-overloaded" web site indicates that it is time for the administrators to upgrade server capacity. Our persistent overload model applies to those web sites which normally have enough capacity to handle the load received, but end up operating near or past their saturation point due to a temporary surge in the popularity of the content that they are serving.

The overload model can influence the results of some TCP connection establishment mechanisms. Consider a mechanism which notifies client TCP stacks to stop the retransmission of connection attempts whenever queue drops occur. Such a mechanism can lead to poor aggregate server throughput if the duration of overload is short compared to the duration of light load because most of the client retransmissions might be serviced during the periods of light load[10]. On the other hand, as we will demonstrate in this thesis, if overload conditions last long enough so that most of the client TCP retransmission attempts occur while the server is still under overload, such a mechanism can actually increase server throughput. Note the distinction between retransmissions at the TCP layer and the application layer. The former are due to the semantics of TCP, where segments that are not

[10] Note however, that as discussed in Section 5.2, it might be possible to avoid queue drops during short bursts altogether by sizing the SYN and listen queues appropriately.

acknowledged by the peer are retransmitted. Higher layer protocols are not aware of these retransmissions. Application-level retries are specifically initiated by the client-side application or by the end user. As described in Section 3.3, we treat application-level retries in the same way as new connection establishment attempts.

Thus, we use the persistent overload model in most of our experiments because it is representative of the load handled by servers during flash crowds. Note that unless otherwise qualified, we use the term "overload" to mean persistent overload. In Section 5.2, we discuss the impact of bursty traffic on server performance with different connection establishment mechanisms.

3.3 Methodology

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. The actual rate achieved by `httperf` depends on how quickly it can recycle resources such as ports and file descriptors to generate new requests. While we always report the target request rate (denoted requests/sec) specified to `httperf` in our evaluation, at very high loads there can be a slight (less than 5%) difference in the target rate and the request rate actually achieved by `httperf`, especially for those connection establishment mechanisms that require a large number of connection attempts to be active for the duration of the timeout (e.g., ***default***). For the one-packet (as well as the single request-per-connection SPECweb99-like) workload, the request rate equals the rate of new connection attempts.

Note that `httperf` does not enforce application-level retries directly. The load generation model in `httperf` assumes that no immediate application-level retries are initiated upon a connection failure, either on an explicit notification from the server or on a (application) timeout. This assumption ensures that the rate of requests generated is independent of the connection establishment mechanism used at the server. We account for application-level retries (e.g., those inititated by a user hitting the "Reload" button in a web browser) by using a higher request rate. That is, the requests generated at a particular rate represent not only maiden requests from new clients, but also retried attempts from those clients that did not get a response for their initial request.

Two minutes provide sufficient time for the server to achieve steady state execution. We have observed that running a data point for longer than two minutes did not alter the measured results, but only prolonged the duration of the experiment. We use a two minute idle period between consecutive data points, which allows TCP sockets to clear the TIME_WAIT state before the next data point is run. Before running an experiment, all non-essential services and daemons such as cron and sendmail are terminated on the server. We also ensure that we have exclusive access to the client and server machines as well as the network during the experiments to prevent external factors from confounding the results.

We primarily use the following metrics in our evaluation of server performance.

1. Server throughput (replies/sec) – the mean number of replies per second delivered by the server, measured at the clients.

2. Client response time – the mean response time measured at the clients for successful connections. It includes the connection establishment time as well as the data transfer time.

Additionally, we use the following metrics to support our analysis of throughput and response time results.

1. Queue drop rate (qdrops/sec) – the number of connection establishment attempts that are dropped at the server TCP stack, reported on a per-second basis.

2. Error rate (errors/sec) – the number of failed (i.e., aborted by the server or timed out by the client application) connections, reported on a per-second basis.

For our primary metrics, server throughput and client response time, we compute and report the 95% confidence intervals based on the statistics provided by `httperf`. We do not compute confidence intervals for our supporting metrics, the error and queue drop rates, which are simple counts obtained for each request rate. Obtaining the confidence intervals for these metrics would require each experiment to be run multiple times, this significantly increases the duration of our evaluation but does not insure additional insight. Note that for workloads using a single request per connection (e.g., the one-packet workload), the sum of the server throughput (reply rate) and the error rate equals the request rate achieved by `httperf`.

The results of each experiment are depicted on a graph where the x-axis represents the request rate and the y-axis displays one of the performance metrics. While discussing the results of our experiments, we focus our attention on server behaviour after it has reached its peak capacity (i.e., the saturation point). All of the connection management mechanisms discussed in this thesis come into play only when the server is overloaded, in particular, only when queue drops occur at the server. They do not affect server throughput or client response times during periods of light load.

In the next chapter, we study the impact of TCP connection establishment and termination mechanisms on server throughput and client response time using the one-packet workload.

Chapter 4

Impact of TCP Connection Management on Overload Server Performance

In this chapter, we show that implementation choices for TCP connection management, in particular TCP connection establishment and TCP connection termination, can have a significant impact on an overloaded server's throughput and response time. Figure 4.1 outlines the road map of the different connection management mechanisms investigated in this chapter[1]. In Section 4.1, we point out the shortcomings in the existing connection establishment mechanism implemented in Linux (as well as some UNIX TCP stacks), discuss alternative solutions, and evaluate them. The connection establishment mechanisms studied can be categorized based on the stage at which they operate in the three-way handshake. We examine two alternatives at the SYN stage and two alternatives at the ACK stage, as shown in Figure 4.1.

We also evaluate two alternative connection termination mechanisms in Section 4.2. We present a connection termination mechanism that allows us to assess the cost of supporting half-closed TCP connections. We also evaluate the impact of an abortive release of connections by clients on server throughput. Note that some of these connection estab-

[1] Note that the existing connection management mechanism in Linux is left out of Figure 4.1 for clarity.

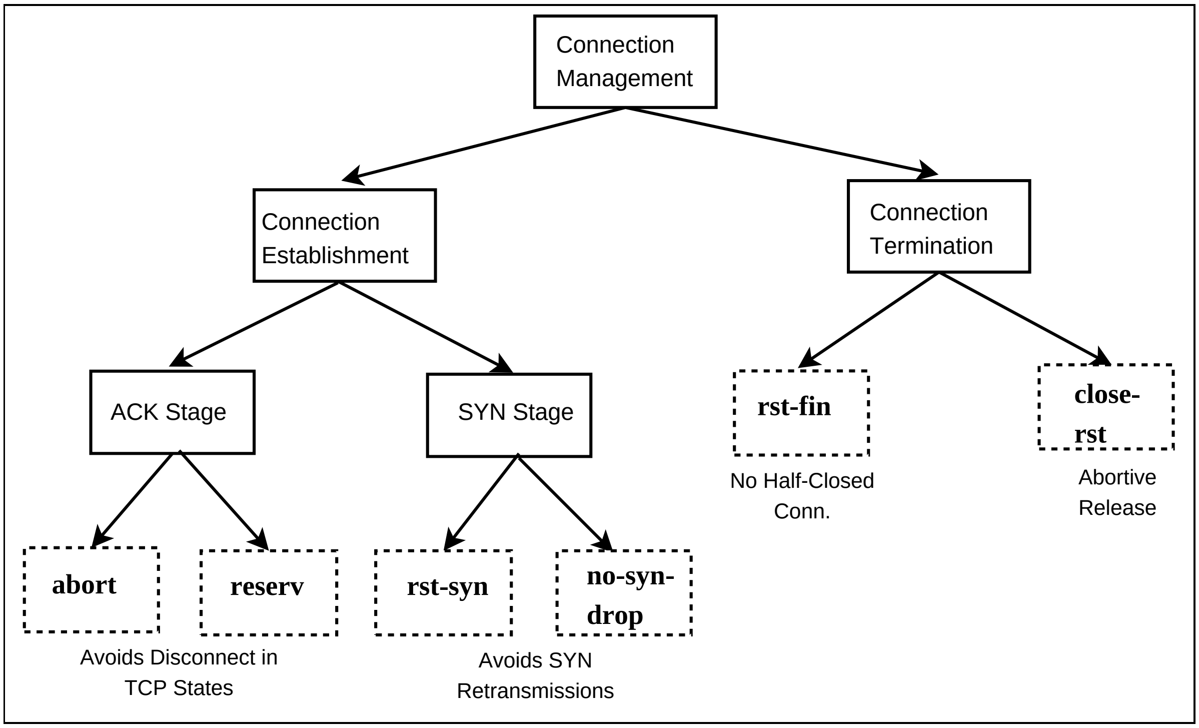


Figure 4.1: Road map of TCP connection management mechanisms investigated in this chapter

lishment as well as termination mechanisms can be used together. In particular, different mechanisms (leaf nodes) that do not have a common parent in the tree shown in Figure 4.1 can be combined together. As explained in Section 3.2, we generate overload using the one-packet workload with a 10 second timeout for all of our experiments in this chapter.

4.1 Impact of TCP Connection Establishment on Server Performance

In the following sections, we describe the shortcomings in the Linux server TCP stack when queue drops occur while processing client connection establishment attempts, and review alternative implementation choices. Some of these alternative mechanisms are already implemented in UNIX or Windows TCP stacks, while some of them are novel. All of these implementation choices affect only the server TCP stack – no protocol changes to TCP, or modifications to the client TCP stacks and applications, or server applications are required.

We refer to the existing connection establishment mechanism in Linux as ***default***. As explained in Section 2.1.1, under overload, queue drops can occur in the Linux TCP stack when the server receives a SYN and either the SynQ3/4Full or the ListenQFullAtSyn rule is triggered, or when the ListenQOverflow rule is triggered upon receiving a SYN/ACK ACK. As indicated earlier, we divide our discussion into problems arising from queue drops occurring at the ACK stage and those occurring at the SYN stage.

4.1.1 Problem – Listen Queue Overflow upon Receiving SYN/ACK ACKs

As shown in Figure 2.1, the Linux TCP stack does not respond with a SYN/ACK to a SYN when the listen queue is full. This prevents the client from immediately sending a SYN/ACK ACK while the listen queue is still likely to be full and thus reduces the number of queue drops due to ListenQOverflow.

However, while responding to a SYN, the TCP stack code only checks if the listen queue is *completely* full. It is possible that the number of SYN/ACKs sent by the server at a given point in time exceeds the amount of free space available in the listen queue. Consider a listen queue of size 128, which is entirely empty. The TCP stack then receives 1000 SYNs. Since the listen queue is not full, it sends back 1000 SYN/ACKs. If the round-trip time (RTT) to each client is roughly the same, the server TCP stack will receive 1000 SYN/ACK ACKs around the same time. This is a problem because only 128 SYN/ACK ACKs can be accommodated in the listen queue, the rest trigger the ListenQOverflow rule.

The Linux TCP stack *silently* drops a SYN/ACK ACK that triggers ListenQOverflow. No notification is sent to the client and the client's incomplete connection continues to occupy space in the SYN queue. Under overload, we have observed that the server application's listen queue is nearly always full because the rate of new connections exceeds the rate at which the application is able to accept them. The TCP stack 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 three-quarters 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 ListenQOverflow.

It is instructive to study the effect of a silent SYN/ACK ACK drop at the server on the client. Figure 4.2 provides `tcpdump` [42] output to illustrate the flow of TCP segments from a client (s01) whose SYN/ACK ACK triggered ListenQOverflow at the server (suzuka). In our `tcpdump` output 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 acknowledgement 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 returns the `connect()` call signifying a successful connection. The client application is now free to start transmitting data on that connection, and therefore sends an 80-byte HTTP request to the server (line 4). However, the client's SYN/ACK ACK (line 3) has been silently dropped by the server due to ListenQOverflow.

From the server's point of view, the connection is still in `SYN_RECV` state awaiting a SYN/ACK ACK, hence, all subsequent client TCP segments (except RSTs) are handled in the `SYN_RECV` code path. Even subsequent data (PSH[2]) and termination notification (FIN) segments from the client are handled in the code path for connections in the `SYN_RECV` state. The data and FIN segments, which are treated as an implied SYN/ACK ACK, can result in additional queue drops if the listen queue is full when they arrive at the server.

Thus 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). Note that a typical HTTP request fits within a single TCP segment. The retransmission timeout (RTO) 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 (RTT) after receiving the SYN/ACK. Eventually, the client-side application times out (after one second in the example in Figure 4.2) and terminates the connection (line 7). Unfortunately, the FIN segment also triggers ListenQOverflow at the server, so it has to be retransmitted (lines 8,

[2] A PSH segment usually corresponds to an application write in BSD-derived TCP stacks [80].

```
(1) 11:32:03.688005 s01.1024 > suzuka.55000: S 1427884078:1427884078(0) win 5840
(2) 11:32:03.688026 suzuka.55000 > s01.1024: S 956286498:956286498(0)
    ack 1427884079 win 5792
(3) 11:32:03.688254 s01.1024 > suzuka.55000: . ack 1 win 5840
(4) 11:32:03.688254 s01.1024 > suzuka.55000: P 1:81(80) ack 1 win 5840
(5) 11:32:03.892148 s01.1024 > suzuka.55000: P 1:81(80) ack 1 win 5840
(6) 11:32:04.312178 s01.1024 > suzuka.55000: P 1:81(80) ack 1 win 5840
(7) 11:32:04.688606 s01.1024 > suzuka.55000: F 81:81(0) ack 1 win 5840
(8) 11:32:05.152238 s01.1024 > suzuka.55000: FP 1:81(80) ack 1 win 5840
(9) 11:32:06.832233 s01.1024 > suzuka.55000: FP 1:81(80) ack 1 win 5840
(10) 11:32:07.686446 suzuka.55000 > s01.1024: S 956286498:956286498(0)
     ack 1427884079 win 5792
(11) 11:32:07.686533 s01.1024 > suzuka.55000: . ack 1 win 5840
     (FreeBSD clients do not send this extra ACK, instead sending
      the FIN directly)
(12) 11:32:10.192219 s01.1024 > suzuka.55000: FP 1:81(80) ack 1 win 5840
(13) 11:32:13.685533 suzuka.55000 > s01.1024: S 956286498:956286498(0)
     ack 1427884079 win 5792
(14) 11:32:13.685637 s01.1024 > suzuka.55000: . ack 1 win 5840
(15) 11:32:16.912070 s01.1024 > suzuka.55000: FP 1:81(80) ack 1 win 5840
(16) 11:32:25.683711 suzuka.55000 > s01.1024: S 956286498:956286498(0)
     ack 1427884079 win 5792
(17) 11:32:25.683969 s01.1024 > suzuka.55000: . ack 1 win 5840
(18) 11:32:30.352020 s01.1024 > suzuka.55000: FP 1:81(80) ack 1 win 5840
     (The following TCP sequence occurs only if the server is able
      to promote the incomplete connection to the listen queue)
(19) 11:32:30.352122 suzuka.55000 > s01.1024: P 1:1013(1012) ack 82 win 46
(20) 11:32:30.352142 suzuka.55000 > s01.1024: F 1013:1013(0) ack 82 win 46
(21) 11:32:30.352394 s01.1024 > suzuka.55000: R 1427884160:1427884160(0) win 0
(22) 11:32:30.352396 s01.1024 > suzuka.55000: R 1427884160:1427884160(0) win 0
```

Figure 4.2: `tcpdump` output of a client whose SYN/ACK ACK is silently dropped due to ListenQOverflow while establishing a connection

9, 12, 15, and 18).

If the server is unable to accommodate any subsequent client segments in its listen queue, its SYN/ACK retransmission timeout expires, and it resends the SYN/ACK (lines 10, 13, and 16). There are some obvious problems with this SYN/ACK retransmission. In the existing Linux implementation, a SYN/ACK is retransmitted even if the listen queue is full. This can easily cause subsequent client responses to be dropped if the listen queue continues to be full (lines 11, 12, 14, and 15), resulting in unnecessary network traffic. By retransmitting the SYN/ACK, the server TCP stack also creates additional load for itself, which is imprudent during overload. More importantly, the incomplete client connection continues to occupy space in the SYN queue (to allow SYN/ACK retransmissions) as long as a segment from that client cannot trigger a new entry to be added to the listen queue. This reduces the amount of free space in the SYN queue and might cause subsequent SYNs to be dropped due to the SynQ3/4Full rule.

If the listen queue is not full when a subsequent client segment arrives, the server TCP 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 might not care about responses on that connection from the server. As a result, it sends a reset (RST) in response to every server reply (lines 21 and 22).

A subtle side-effect of handling all client segments subsequent to a SYN/ACK ACK drop in the `SYN_RECV` code path is an inflation (or over-counting) of queue drops measured at the server. The count of ListenQOverflow-induced queue drops can be higher than expected and can exceed the total number of new connections attempted by the client. Table 4.1 shows a breakdown of all queue drops due to ListenQOverflow ("Total ListenQOverflow-Induced Qdrops") counted by the server TCP stack for the one-packet workload with the μserver at different request rates. Recall that we did not add this statistic, it is tracked and reported by the vanilla Linux kernel. The "SYN/ACK ACKS Dropped" column indicates the number of SYN/ACK ACKs dropped and the "Other Segments Dropped" column indicates the number of all other segments (i.e., data and FIN) dropped in the `SYN_RECV` state code path. We have instrumented these two counters to obtain a fine-grained count of ListenQOverflow-induced queue drops. Table 4.1 shows that at a high rate of incoming

Conn./sec	SYN/ACK ACKs Dropped	Other Segments Dropped	Total ListenQOverflow-Induced QDrops
20,000	62,084	72,235	134,319
26,000	171,641	181,867	353,508
32,000	165,935	167,620	333,555

Table 4.1: Breakdown demonstrating inflation of ListenQOverflow-induced queue drops reported by server TCP stack due to silent drop of SYN/ACK ACKs

connection attempts, the server TCP stack inflates the number of queue drops due to ListenQOverflow, and hence, the number of failed connection attempts by a factor of two. That is, while we expect the number of ListenQOverflow-induced queue drops to reflect only the number of SYN/ACK ACKs dropped, this count in the kernel is inflated because subsequent client segments are handled by the same `SYN_RECV` state code path. The inflated ListenQOverflow-induced queue drop count skews the total number of queue drops in *all* the results with ***default*** reported in this thesis.

To summarize, the default mechanism to handle queue drops arising from ListenQOverflow in the Linux TCP stack (***default***) is problematic because:

1. It can result in a disconnect between the TCP states at the client and the server.
2. It can result in unnecessary TCP traffic (i.e., the transmission of data and FIN segments following a silent SYN/ACK ACK drop) at the server as well as in the network.

4.1.2 Solutions for the Listen Queue Overflow Problem

The disconnect between the TCP states at the end-points resulting from queue drops due to ListenQOverflow and the subsequent redundant TCP traffic can be avoided in two different ways – reactively, or by proactively preventing the listen queue from overflowing.

Reactive Solutions

Instead of silently dropping a SYN/ACK ACK when ListenQOverflow is triggered, 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 implicit assumption in aborting a connection on queue drops due to ListenQOverflow 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. We will refer to this mechanism as ***abort***[3].

Unfortunately, any RST sent after a SYN/ACK is not transparent to the client application. The application will be notified of a reset connection by the client TCP stack on its next system call on that socket, which will fail with an error such as `ECONNRESET`. Thus, ***abort*** results in the client getting a false impression of an established connection, which is then immediately aborted by the server. Recall that without ***abort***, when ListenQOverflow is triggered, the client has an impression that a connection has been established, when it is in fact in `SYN_RECV` state at the server.

By default, ***abort*** is used when a SYN/ACK ACK is dropped in TCP stacks in FreeBSD, HP-UX 11, Solaris 2.7 and Windows [45]. Although it is not enabled by default, administrators on Linux can achieve similar behaviour by setting the `net.ipv4.-tcp_abort_on_overflow` sysctl.

Jamjoom and Shin [45] describe another reactive mechanism to handle queue drops arising from ListenQOverflow. Whenever a SYN/ACK ACK triggers ListenQOverflow, the TCP stack can temporarily grow the listen queue to accommodate it, but continue to drop SYN segments by enforcing the ListenQFullAtSyn rule. Growing the listen queue to accommodate the additional SYN/ACK ACK allows the server to transition the associated connection to the `ESTABLISHED` TCP state. Note that no resizing of kernel data structures is required because the listen queue is implemented as a linked list. 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 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

[3] An abbreviation of the `abort_on_overflow` sysctl that achieves this behaviour in Linux.

of resources. For these reasons, in this thesis, we do not consider growing the listen queue to address queue drops due to ListenQOverflow.

Proactive Solutions

Proactive approaches that avoid queue drops due to ListenQOverflow ensure that the listen queue is never full when a SYN/ACK ACK arrives. We are aware of one such proactive approach, namely, lazy `accept()` [80]. We also describe a new proactive approach, which implements listen queue reservation.

Lazy `accept()`

In the lazy `accept()` approach, 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. In fact, there is no reason to use separate SYN and listen queues to implement lazy `accept()`. The server TCP stack can track information about connections on which a SYN is received in a single queue until there is an `accept()` system call from the application. It can subsequently complete the three-way handshake for the entry at the head of that queue and make its associated socket descriptor available to the application[4].

Such a scheme provides the server application with fine-grained admission control functionality because the TCP stack does not complete the three-way handshake *before* the application notifies it to do so. This is in contrast with most current TCP stack and socket API implementations, in which a server application has to reject a connection that has already been established by its TCP stack in order to effect admission control.

The Solaris 2.2 TCP stack provided a `tcp_eager_listeners` parameter to achieve the lazy `accept()` behaviour [80]. Support for this parameter seems to have been discontinued in later Solaris versions. The Windows Sockets API (Version 2) provides this functionality through the `SO_CONDITIONAL_ACCEPT` socket option which can be used with the `WSAAccept()` system call [58].

[4] Unfortunately, none of the open-source TCP stacks support lazy `accept()`, hence, we are unable to verify its implementation details.

Note that a lazy `accept()` approach can result in poor performance because the application thread will invariably block waiting for the three-way handshake to be completed by the TCP stack. While the server TCP stack can return an `EWOULDBLOCK` error for sockets in non-blocking mode if there is no pending connection (i.e., queued SYN request), a lazy `accept()` would have to otherwise block until the server TCP stack receives a SYN/ACK ACK, unless there is an external asynchronous notification mechanism to signal connection completion to the application. It is unclear whether the implementation of lazy `accept()` in Windows (which is currently the only major platform that supports this proactive mechanism) provides an asynchronous notification or lets the `WSAAccept()` system call on a socket block (even if the socket is in non-blocking mode) until the three-way handshake is completed. More importantly, a lazy `accept()` makes the application vulnerable to SYN flood denial of service (DoS) attacks, where malicious clients do not send a SYN/ACK ACK in response to a SYN/ACK. This can cause lazy `accept()` calls to stall. The queued connections from malicious clients not only deplete server resources, they can also stall the application by causing SYN segments from legitimate clients to be dropped.

Listen queue reservation

Another proactive method of ensuring that there are no listen queue overflows, 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 (e.g., when the client resets the connection or if the number of SYN/ACK retransmission attempts exceeds the maximum limit). The server TCP stack sends a SYN/ACK only if there is room for a new reservation in the listen queue. The ***reserv*** mechanism avoids listen queue overflows by ensuring only as many SYN/ACK ACKs will be received as have been reserved in the listen queue. We refer to the listen queue reservation mechanism as ***reserv***.

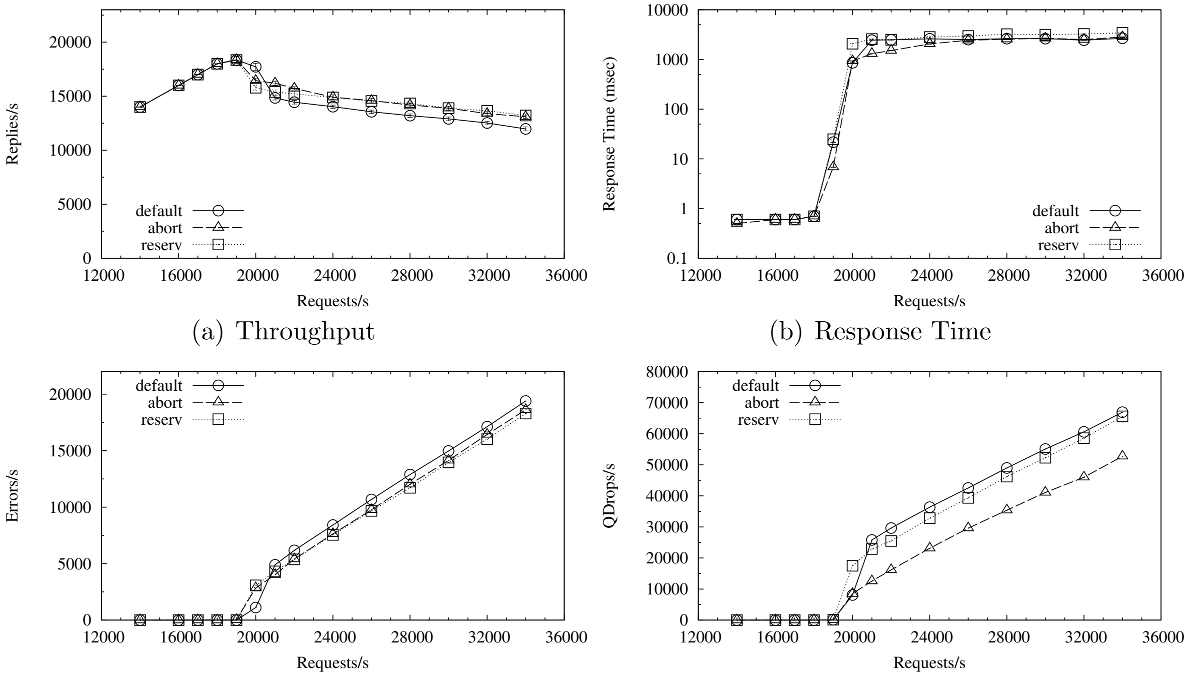
To our knowledge, the ***reserv*** mechanism described above is novel. However, we arrived at it at the same time as the `Dusi_Accept()` call described by Turner et al. [83] for the Direct User Socket Interface within the ETA architecture. This call allows applications to post asynchronous connection acceptance requests which reserve room in the listen queue. In contrast to work by Turner et al., our listen queue reservation mechanism works with the BSD sockets API and TCP stacks implemented in most existing operating systems.

Our current implementation of ***reserv*** is geared toward non-malicious clients in a LAN environment. It primarily seeks to address the disconnection in TCP states at the client and the server and to provide a better understanding of whether listen queue reservations can improve server throughput. Hence, we assume that every client for which there is a reservation in the listen queue will respond immediately with a SYN/ACK ACK. This implementation can be generalized to allow for "overbooking" (similar to the overbooking approach used for airline seat reservations) to account for clients whose SYN/ACK ACK is delayed. In particular, in WAN environments we might need to send more SYN/ACKs to high-latency clients than there are spaces for reservation, lest the server's listen queue becomes completely empty while the SYN/ACK ACKs from the clients are in transit[5]. In general, the number of reservations can be bounded by the sum of the SYN queue size, the listen queue size, and the number of client SYN/ACK ACKs in transit. A more important shortcoming of our current ***reserv*** implementation is its vulnerability to attacks from malicious clients. We defer the discussion of security implications of ***reserv*** until Section 4.1.7.

4.1.3 Evaluation of the Listen Queue Overflow Solutions

We now compare the performance of ***abort*** and ***reserv*** against ***default***. As described in Section 3.3, we measure the throughput, response time, error rate, and queue drops when these mechanisms are used with the user-space μserver and the kernel-space TUX web servers. Figures 4.3 and 4.4 show the results obtained with the μserver and TUX respectively. Note that for all the μserver results in this thesis (including those shown in Figure 4.3), the x-axis starts at 12,000 requests/sec. We leave out data points prior

[5] Note that a client SYN/ACK ACK lost in transit will result in a SYN/ACK retransmission by the server. A listen queue reservation is deleted when the threshold on maximum SYN/ACK retransmissions is reached.



(a) Throughput

(b) Response Time

(c) Errors

(d) Queue Drops

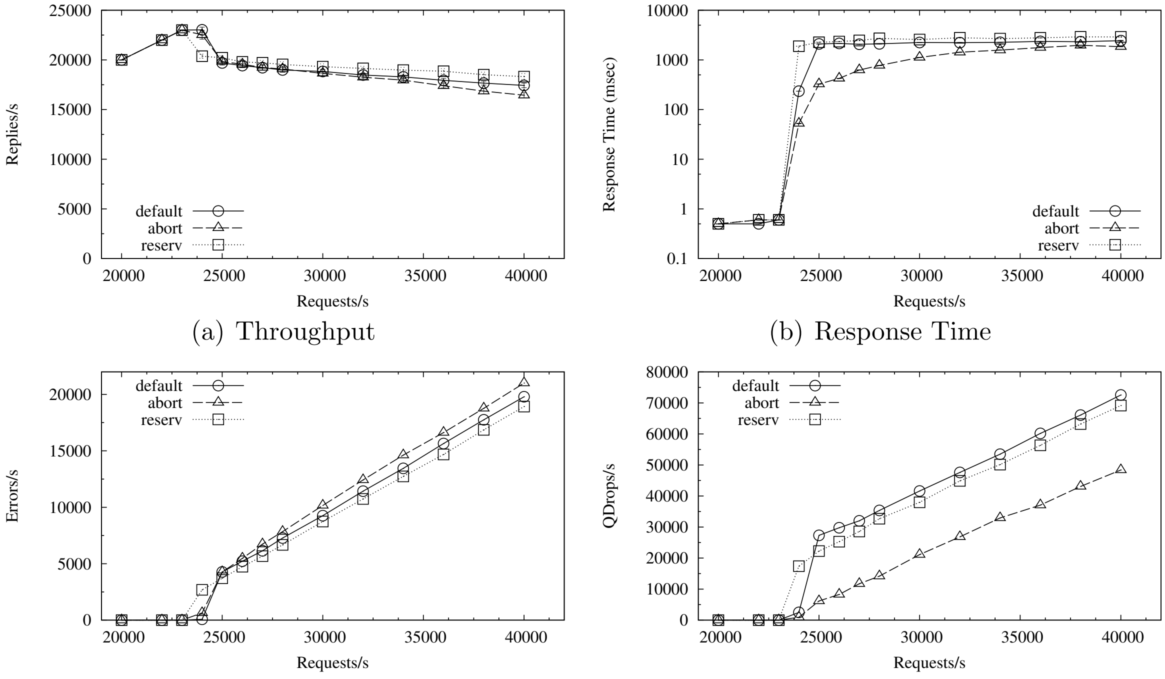
Figure 4.3: Evaluation of the μserver with listen queue overflow solutions

to 12,000 requests/sec out of the graphs because at these points the server is not under overload and its reply rate matches the client request rate. For the same reason, the x-axis in all the graphs for TUX starts at 20,000 requests/sec.

Figures 4.3(d) and 4.4(d) show that ***abort*** reduces the number of queue drops significantly by ensuring that there is only one queue drop due to ListenQOverflow per connection. Surprisingly, ***reserv*** has significantly higher queue drops compared to ***abort***. This is due to retransmitted SYN segments from the clients. The server TCP stack does not respond to a SYN with a SYN/ACK when there is no reservation available in the listen queue. That is, it silently drops the SYN[6]. The SYN retransmission timeout at the client's TCP stack expires if it does not receive a SYN/ACK within a specific amount of time, resulting in another SYN transmission[7]. This retransmitted SYN can also result in further queue drops if there is no room for a reservation in the listen queue when it arrives at the

[6] In most UNIX TCP stacks, including Linux, whenever a SYN is dropped, it is dropped silently. No notification is sent to the client [80].

[7] All TCP stacks have an upper bound on how many times a SYN is retransmitted.



(a) Throughput

(b) Response Time

(c) Errors

(d) Queue Drops

Figure 4.4: Evaluation of TUX with listen queue overflow solutions

server. Note that SYN retransmissions also influence the queue drops in ***abort***, but to a lesser degree because connection requests are mostly rejected at the ACK stage. Table 4.2 shows a per-second breakdown of connection establishment segments dropped at the SYN stage, namely, "SYNs Dropped", and either dropped or rejected at the ACK stage, namely, "ACKs Reset/Dropped", for three request rates with TUX.

The table demonstrates that with ***reserv***, all connection establishment attempts are dropped at the SYN stage, while with ***default***, most attempts are dropped at the SYN stage. The SYN drop rate is higher than the request rate in both of these mechanisms due to SYN retransmissions by the clients. With ***abort***, most attempts are rejected at the ACK stage with fewer SYN drops compared to ***reserv*** or ***default***. Note that with ***abort*** all of the queue drops at SYN stage are due to ListenQFullAtSyn and not SynQ3/4Full. In our low-latency LAN environment, a SYN/ACK ACK from a client arrives almost immediately after the server sends a SYN/ACK. This ensures that connections occupy space in the SYN queue for a very short time. The quick SYN/ACK ACK arrival does however put additional pressure on the listen queue.

Rate	*default*		*abort*		*reserv*	
	SYNs Dropped	ACKs Dropped	SYNs Dropped	ACKs Reset	SYNs Dropped	ACKs Dropped
26,000	26,828	5,135	2,992	5,698	27,195	0
30,000	39,966	4,734	12,341	9,818	40,782	0
34,000	53,216	4,283	21,312	13,259	53,883	0

Table 4.2: Breakdown of connection attempts dropped or rejected per second at the SYN and ACK stages with TUX

Figures 4.3(a) and 4.4(a) indicate that neither ***abort*** nor ***reserv*** are able to provide significant improvements in throughput over ***default***. The peak[8] throughput is the same with all the three mechanisms. After the peak, ***abort*** and ***reserv*** provide less than 10% improvement in throughput over ***default*** in the μserver and a negligible increase in throughput in TUX. It is also evident from Figures 4.3(b) and 4.4(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 thesis. The sharp spike in response times after 19,000 requests/sec in Figure 4.3(b) corresponds to the server's peak (saturation point). After the peak (i.e., 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. The lack of improvement in throughput with ***abort***, despite the reduction in the number of queue drops, is due to the overhead of processing retransmitted SYN segments at the server. The same problem limits the throughput obtained with ***reserv***. In Section 4.1.4, we will analyze the overhead of processing retransmitted SYN segments in detail.

Figure 4.4(c) shows that with TUX the error rate (i.e., the rate of connection failures seen by the client) resulting from ***abort*** is 5%-10% higher than ***default***. This is because TUX is able to drain its listen queue very quickly by running in the kernel. The RST sent when a SYN/ACK ACK is dropped due to ListenQOverflow is counter productive because it terminates the client connection prematurely. On the other hand, with ***default***

[8] The highest rate at which the server's reply rate matches the request rate.

subsequent segments transmitted by the client are treated as implied SYN/ACK ACKs and if one such segment does not trigger ListenQOverflow (before the client times out), the connection can be placed on the listen queue and is eventually serviced. Note that a user-space server such as the μserver might not be able to drain its listen queue quickly enough to keep up with the rate of incoming SYN/ACK ACKs. In such cases, by preventing subsequent client transmissions from being treated as implied SYN/ACK ACKs (thereby adding to the server's burden), ***abort*** can actually reduce the error rate by up to 10% as demonstrated in Figure 4.3(c).

Note that neither ***abort*** nor ***reserv*** affect the throughput, response time, queue drops, or the error rate with either web servers prior to the peak (saturation point). We reiterate that at loads below saturation, there are no queue drops, hence the implementation choices for connection establishment have no impact on server throughput and response time. Thus, ***abort*** and ***reserv*** succeed in eliminating the disconnect in the TCP states at the end-points and alleviate load by relieving the server from handling data and FIN segments transmitted following a silent SYN/ACK ACK drop. However, both these mechanisms fail to improve throughput or response times because they do not mitigate the overhead of processing retransmitted SYN segments. We will evaluate this overhead in the next section.

4.1.4 Problem – Silent SYN Drop

As seen in Section 4.1.3, SYN retransmissions can increase the number of queue drops. In this section, we analyze the impact of retransmitted SYN segments on server throughput and response time under persistent overload.

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 4.3 shows the SYN retransmission timeouts used in popular TCP stacks. Note that some of the timeouts in the table (in particular those for Linux and FreeBSD) differ from previously reported values by Jamjoom and Shin [45]. We obtained the timeouts

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 4.3: SYN retransmissions timeouts in current TCP stacks

shown in Table 4.3 by observing the actual `tcpdump` output on different operating systems.

Figure 4.5 shows the `tcpdump` output of a Linux 2.4 client whose SYNs are silently dropped by the server. It demonstrates that the client TCP stack will retransmit SYNs up to five times – 3, 9, 21, 45, and 93 seconds[9] after the original SYN transmission.

```
 (1) 19:32:35.926222 s01.1033 > suzuka.55000: S 3600646294:3600646294(0) win 5840
 (2) 19:32:38.923018 s01.1033 > suzuka.55000: S 3600646294:3600646294(0) win 5840
 (3) 19:32:44.922816 s01.1033 > suzuka.55000: S 3600646294:3600646294(0) win 5840
 (4) 19:32:56.922559 s01.1033 > suzuka.55000: S 3600646294:3600646294(0) win 5840
 (5) 19:33:20.922018 s01.1033 > suzuka.55000: S 3600646294:3600646294(0) win 5840
 (6) 19:34:08.912657 s01.1033 > suzuka.55000: S 3600646294:3600646294(0) win 5840
```

Figure 4.5: `tcpdump` output of a client whose SYNs are silently dropped while establishing a connection

A client application can also prevent further SYN retransmissions by issuing a `close()` system call on a socket descriptor that has not been connected. The SYN retransmission timeout in the TCP stack is different from a timeout used by the client application or by the end user, each of which might use a different timeout. For example, a user may stop trying to connect to a web site if he or she does not get a response within 5 seconds. Similarly, a web browser might close a connection after waiting for 15 seconds. In fact, the 10 second application-level timeout used in all of our experiments allows for the retransmission of at most two SYNs, which is lower than the maximum number of SYN retransmissions made by the Linux TCP stack. As explained in Section 3.3, we take into account retransmissions

[9] These retransmission timings are cumulative values of the differences in successive lines in Figure 4.5.

by the application or the end user by including them in the rate of incoming client requests.

As described in Section 2.1.1, if either the SYN or the listen queue is full, the server TCP stack does not respond with a SYN/ACK. By ignoring the SYN, it forces the client TCP stack to retransmit another SYN, hoping that the queues will have space for the connection request at a later time [80]. Thus, 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. That is, the majority of clients retransmit their SYNs while the server is *still* overloaded. Hence, 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.

Cost of Processing Retransmitted SYN Segments Under Overload

In order to understand why ***abort*** did not improve server performance in Section 4.1.3, we now evaluate the overhead of processing retransmitted SYN segments under overload.

We profiled our server using `oprofile` [41], a system-wide profiler, to determine the percentage of CPU time spent in kernel-space as well as user-space code. Note that as described in Section 3.2, our workloads do not require disk accesses and there are no network bottlenecks in our experimental environment. We then grouped the functions reported in the flat profile into different categories for a component-level analysis. Since we are interested in analyzing Internet servers, we used the following categories – driver (interrupt-handling and NAPI-based packet-processing), net-ip (IP packet-processing), net-tcp (TCP segment-processing), net-socket (socket API, including `skbuff` management), net-other (protocol-independent device support), mm (generic memory management), other-kernel (all other kernel routines such as process, and file-system management), and server application (TUX or the μserver).

For simplicity, we discuss profiling results obtained by running the in-kernel TUX server

Kernel Component	**%CPU Time**	
	Rate=17,500	**Rate=25,000**
driver	26.96	21.83
net-ip	3.45	3.62
net-tcp	16.59	21.56
net-socket	7.92	8.90
net-other	3.82	3.98
tux	13.09	13.47
mm	6.86	7.98
other-kernel	18.58	17.29
TOTAL	97.27	98.63

Table 4.4: Breakdown of CPU time spent in kernel components near peak load and under overload summarized from `oprofile` data for TUX with ***abort*** on the one-packet workload

with ***abort*** on the one-packet workload. The results with the μserver are qualitatively similar, except for the additional event-notification overhead that one would expect from a user-space, event-driven network server. Table 4.4 shows the component-level profiles for two data points, one just before the peak (17,500 requests/sec), and one under overload (25,000 requests/sec). Note that the total CPU time does not add up to 100% because it excludes time taken by the `oprofile` daemon. Note also that the peak rate is lower with profiling enabled – without profiling TUX peaks near 22,000 requests/sec with ***abort***, as shown in Figure 4.4(a).

As the load exerted on the server increases, the TCP component dominates the CPU usage. The CPU consumption of TCP segment processing goes up from about 17% to 22% as the request rate increases from 17,500 to 25,000. The CPU usage of all other components except "driver" (as explained below) remains approximately the same. Even with an increase in server load, the application (shown as "net-tux") does not get a proportionately higher share of CPU time to complete work on existing requests or to accept new connections.

The amount of time spent in the driver component decreases with an increase in load because as explained in Section 2.2.2, we have enabled NAPI in our experiments to avoid interrupt-driven receiver livelock. With NAPI, the networking subsystem resorts to periodic polling under high loads to prevent interrupt handling from hogging the CPU. NAPI also ensures that tasks other than packet-processing (e.g., the web server) get adequate CPU time under overload at the server. In other statistics collected during the experiments (but not shown here), we have observed that the amount of time spent polling is lower before the peak load (at 17,500 requests/sec) compared to under overload (at 25,000 requests/sec)[10]. Hence, under overload, with strategies to counter receiver livelock in place, TCP segment-processing code becomes one of the major consumers of CPU cycles.

It is instructive to determine the cause of the increase in CPU cycles consumed by the TCP component when the server is overloaded. Table 4.5 (which corresponds to TUX results for *abort* in Figure 4.4) shows the breakdown of incoming TCP segments as the load exerted increases. The expected number of SYN segments is computed by multiplying the request rate by the duration of the experiment (around 120 seconds). The difference between the actual and expected SYN segments received can be attributed to SYN retransmissions from the clients[11].

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 estab. conn. to actual SYN segments	1	0.57	0.35
Total segments	15,840,231	16,968,566	19,161,742

Table 4.5: Incoming TCP segment statistics near peak load and under overload for TUX with *abort* on the one-packet workload

[10] Consequently, the number of interrupts per second is higher at 17,500 requests/sec.

[11] More correctly, a difference greater than 50 is due to SYN retransmissions, a value less than that can be caused by our workload generator running slightly longer than the specified duration.

As the load exerted increases, the number of connection establishment segments (i.e., SYN plus SYN/ACK ACK segments) received and processed at the server increases rapidly compared to the number of segments destined for already established connections. For example, at 22,000 requests/sec, only a third of the total TCP segments received are for establishing connections (i.e., 5,280,029 out of 15,840,231), while at 34,000 request/sec, more than half of the total TCP segments processed constitute an attempt to establish a connection (i.e., 10,213,667 out of 19,161,742). The increase in connection establishment segments is primarily due to retransmitted SYN segments. As shown in Table 4.5 under overload, retransmitted SYN segments significantly increase the aggregate TCP traffic at the server. This corroborates results presented by Jamjoom and Shin [45], who report that SYN retransmissions result in close to two-folds increase in traffic at the server during flash crowds. As we will demonstrate in Figure 4.9, many client connection attempts require the server to process more than one SYN segments before the connection is either established, aborted, or timed out. 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. We will discuss the implications of retransmitted SYN segments on server throughput and response time in detail in Section 4.1.6. At this point, however, we can conclude that a significant portion of the server's CPU resources are devoted to processing retransmitted TCP SYN segments under overload.

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 [54]. As discussed in Section 5.4, dropping SYN segments at an earlier stage in the networking stack (e.g., at the IP layer as done by some firewalls such as `netfilter`) does not mitigate the negative performance impact of retransmitted SYN segments.

4.1.5 Alternatives to Silent SYN Drop

In this section, we describe approaches that attempt to alleviate TCP segment-processing load resulting from retransmitted SYN segments at the server. These solutions can be broadly classified as follows[12].

1. Sending a RST to a client whenever a SYN is dropped.
2. An approach such as persistent dropping [45] to filter out retransmitted SYN segments with lower overhead. Note that in persistent dropping, retransmitted SYN segments are still dropped silently.
3. Dropping no SYN segments, but rejecting connection establishment attempts at the ACK stage instead.

Persistent dropping was discussed in Section 2.2.3. We describe the other alternatives in detail in the following subsections.

Send RST on a 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. Note that the ***abort*** mechanism involves sending a RST whenever a SYN/ACK ACK 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[13]. As per RFC 793, the client TCP stack should then give up on the connection and indicate a 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. By sending a RST when a SYN is dropped[14], we eliminate TCP-level retransmissions, which are transparent to the client application. 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

[12] Note that this list is not exhaustive.

[13] A RST in response to a SYN in TCP is equivalent to an ICMP "Port Unreachable" message [80].

[14] Note that if a RST sent to a client is lost under wide-area network connections, it is no different than a silent SYN drop, a client will retransmit a SYN when its retransmission timeout expires.

to send a RST whenever a SYN is dropped. This behaviour can be chosen through the `net.ipv4.tcp_drop_syn_with_rst` sysctl. We refer to this mechanism as ***rst-syn***[15].

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 4.1.2. 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 aborting of connections on ListenQOverflow-induced SYN/ACK ACK drops, we call the resulting mechanism ***abort rst-syn***.

While we developed ***rst-syn*** independently after discovering the high overhead of processing retransmitted SYNs, we later discovered that some TCP stacks, including those on some Windows operating systems, already implement such a mechanism [59][16]. As indicated earlier, most UNIX TCP stacks drop SYN segments silently. In this context, the results presented in this thesis can be viewed as a comparison of the different connection establishment mechanisms implemented in current TCP stacks.

Before presenting the results of ***rst-syn***, we point out its shortcomings. 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 [34]. RFC 793 states: "As a general rule, reset (RST) must be sent whenever a segment arrives which apparently is not intended for the current connection. A reset must not be sent if it is not clear that this is the case" [73]. 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, such an approach violates the recommendations of RFC 793. In particular, 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 a functionality is suggested in RFC 1122 – "A RST segment could contain ASCII text that encoded and explained the cause of the RST" [17]. A server TCP stack would indicate that

[15] An abbreviation of "RST in response to a SYN on a SYN drop".

[16] Some firewalls also use ***rst-syn*** for congestion control [34].

it is overloaded in its RST segment. Client TCP stacks could then notify the applications about the overload at the server with an error such as `ESERVEROVRLD`. Unfortunately, such an approach would require modifications to existing TCP stack implementations at the clients. In this thesis, we explore approaches to allow the server to notify clients that it is overloaded with currently deployed client TCP stacks, and evaluate the effectiveness of such approaches.

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[17]. Note that this behaviour does not follow the specifications of RFC 793. As shown in Figure 4.6, RFC 793 explicitly specifies that a client TCP stack should abort connection attempts upon receiving a RST in response to a SYN.

```
If the state is SYN-SENT then
   [checking of ACK bit omitted]
   If the RST bit is set
      If the ACK was acceptable then signal the user
      "error: connection reset", drop the segment, enter CLOSED state,
      delete TCB, and return.
      Otherwise (no ACK) drop the segment and return.
```

Figure 4.6: Event-processing steps for handling RSTs received in `SYN_SENT` state specified in RFC 793 [73]

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[18]. 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 [60]. Microsoft provides the following rationale for their decision to retry SYNs – “The approach of Microsoft platforms is that the system administrator has the freedom to adjust TCP performance-related settings to their own tastes, namely the

[17] Microsoft client TCP stacks ignore “ICMP port unreachable” messages in a similar fashion, retrying a SYN instead [45].

[18] A port which does not have an associated listening socket on the server.

maximum retry ... The advantage of this is that the service you are trying to reach may have temporarily shut down and might resurface in between SYN attempts. In this case, it is convenient that the `connect()` waited long enough to obtain a connection since the service really was there" [60]. While a user familiar with RFC 793 can set this parameter to zero, we assume that most users run with the default values. Also 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.

We will refer to the behaviour of Windows-like TCP stacks that retry a SYN upon receiving a RST for a previous SYN as ***win-syn-retry***. To our knowledge, there are no (public) results that quantify the effects of the ***win-syn-retry*** behaviour with server TCP stacks which do send a RST to reject a SYN. On the other hand, some past work has cited the ***win-syn-retry*** behaviour as a justification for dropping SYN segments silently [45, 85]. In this thesis, we test this hypothesis by evaluating the effectiveness of ***rst-syn*** with RFC 793-compliant as well as ***win-syn-retry*** 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 implement the ***win-syn-retry*** behaviour. If a configurable option is set, our client stacks retransmit SYNs up to 3 times after receiving a RST for the original SYN. Note that we have chosen an aggressive retry value of 3 (default in Windows 98/NT) instead of 2 (default in Windows 2000/XP), which we suspect would be used in most current Windows client TCP stacks. We will refer to our emulation of the ***win-syn-retry*** behaviour in Linux as ***win-emu***.

No SYN Drops

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 ***win-syn-retry*** 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. Thus, we send a SYN/ACK in response to every SYN, irrespective of whether there is space available in the SYN queue or the listen queue. We later use ***abort*** to reset those connections whose SYN/ACK ACKs cannot be accommodated in the listen queue. We refer to this mechanism as ***no-syn-drop***.

An obvious limitation of ***no-syn-drop*** is that we give some clients false impression of the server's ability to establish a connection. In reality, under overload, we can only accommodate a portion of the SYN/ACK ACKs from the clients in the listen queue, so we have to abort some established client connections[19]. 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, ***win-syn-retry*** clients) about an overload at the server in order to stop retransmission attempts. Note that when a RST in sent to a client whose SYN/ACK ACK triggers ListenQOverflow, all state information about the client connection at the server TCP stack is destroyed (with both ***abort*** and ***no-syn-drop***). If this RST is lost in a WAN environment, subsequent (data or FIN) segments from the client will be answered with a RST by the server TCP stack. In this way, the client will eventually receive a notification of the server's inability to continue with the connection.

In order to ensure that there are no SYN drops with ***no-syn-drop***, we eliminated the ListenQFullAtSyn rule while processing SYN segments in the Linux TCP stack. Note that as a result of eliminating this rule, we expect many more listen queue overflows. We had to use another technique to ensure that SYN segments are not dropped when the SYN queue is full. While we have not observed any SYN drops because of a full SYN queue[20] with ***abort*** or ***reserv*** in our environment, the SYN queue could get close to full with high-latency clients in a WAN environment.

We use TCP SYN cookies [54, 14], a mechanism originally created to guard against SYN

[19] Note that the connections aborted are in the `ESTABLISHED` state only at the client.

[20] More precisely due to SynQ3/4Full.

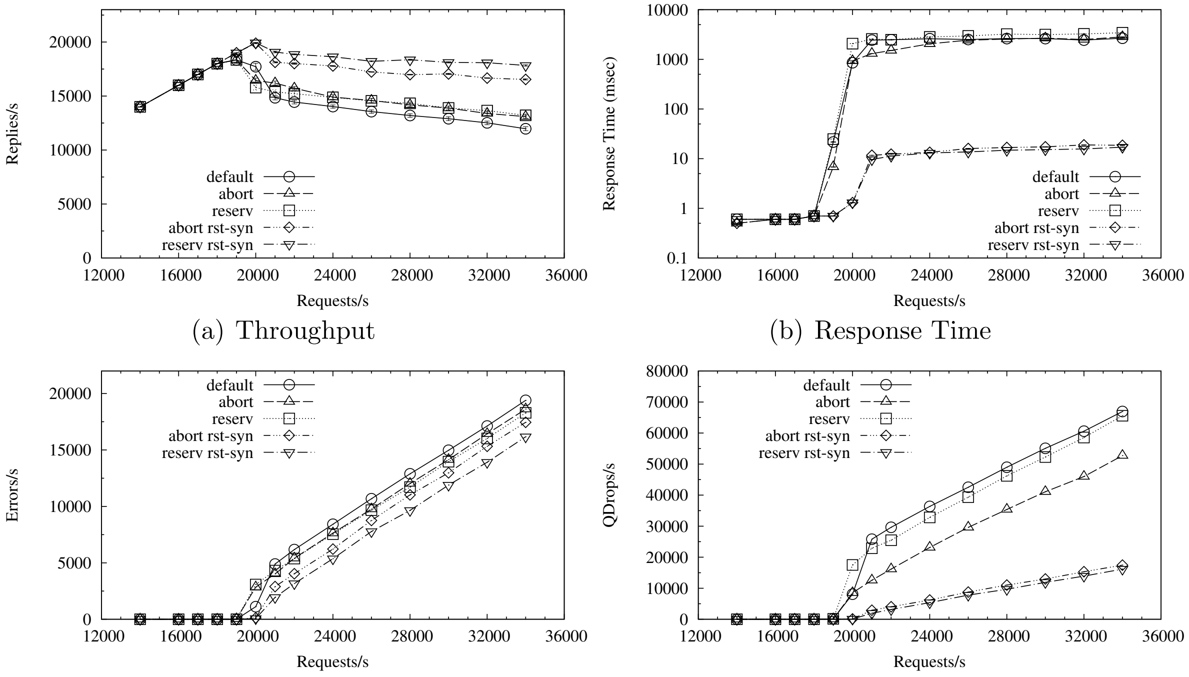
flood denial of service attacks, to ensure that SYNs are not dropped due to lack of space in the SYN queue. SYN cookies do not save any information about incomplete connections at the server, thereby giving the impression of an unbounded SYN queue. The server TCP stack sends a specially-crafted sequence number (the cookie) in its SYN/ACK that encodes information that would normally be saved in the SYN queue. When a SYN/ACK ACK arrives, the server recreates the incomplete request based on the sequence number acknowledged. Note that the SYN cookies approach has some limitations – in particular not all TCP options (e.g., window scaling) included in a SYN are encoded in a SYN/ACK, these cannot be reconstructed from the SYN/ACK ACK received, so this information is lost [54]. Hence, most UNIX TCP stacks use SYN cookies as a fall-back mechanism only after the SYN queue is completely full. We also had to fix the TCP SYN cookies implementation in Linux to properly implement *abort*. Several 2.4 and 2.6 kernel sources that we examined (including the latest 2.6.11.6) did not check if `abort_on_overflow` (which implements *abort* in Linux) is enabled when SYN cookies are used – a potential oversight on the part of the networking developers since neither SYN cookies nor `abort_on_overflow` are enabled by default. As discussed in Section 4.1.1 without *abort*, server throughput is degraded on a silent listen queue overflow, this problem is exacerbated by relaxing the ListenQFullAtSyn rule when responding to a SYN. In our evaluation of *no-syn-drop*, we use a modified Linux TCP stack, which enforces *abort* when used in conjunction with SYN cookies.

4.1.6 Evaluation of Silent SYN Drop Alternatives

In this section, we evaluate server performance with *rst-syn* on RFC 793-compliant clients, and with *rst-syn* and *no-syn-drop* on Windows-like (*win-emu*) clients.

Regular Clients

We first evaluate server throughput and response times with *rst-syn* on client TCP stacks which follow RFC 793, and abort a connection attempt upon receiving a RST in response to a SYN. We will refer to such client TCP stacks as *regular* clients. As indicated earlier, we expect that *rst-syn* will be used in conjunction with *abort* or *reserv*. Figures 4.7



(a) Throughput

(b) Response Time

(c) Errors

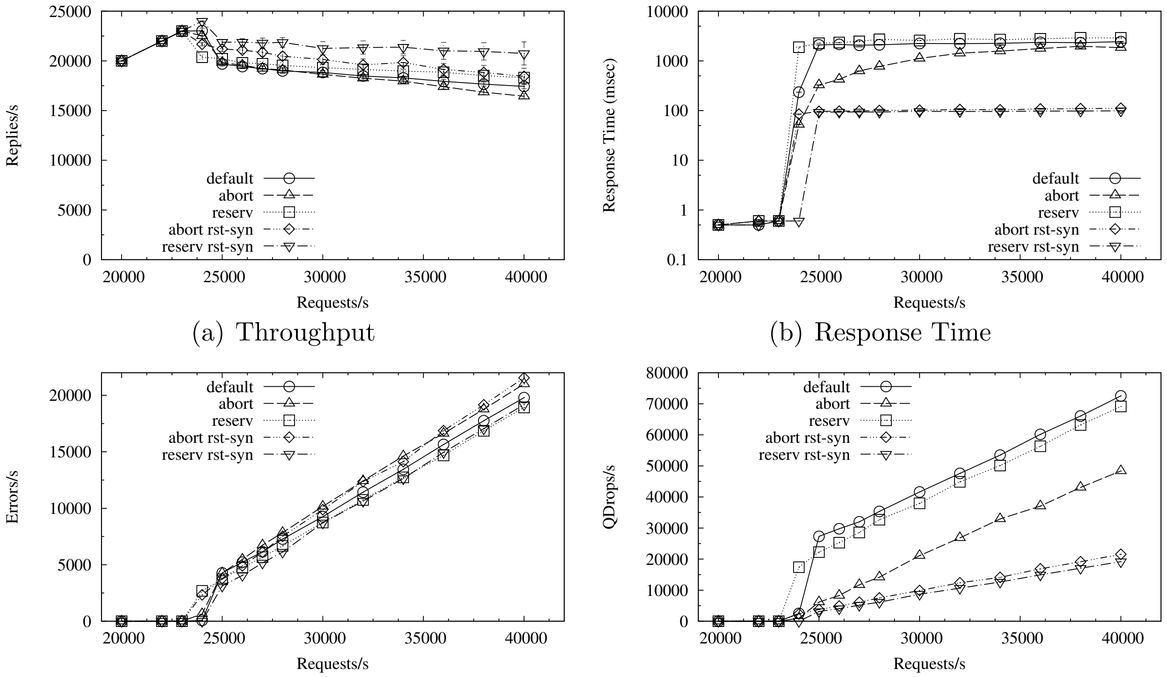
(d) Queue Drops

Figure 4.7: Evaluation of the μserver with alternatives to silent SYN drop

and 4.8 demonstrate that when *abort* and *reserv* are combined with *rst-syn*, they yield a significant improvements in throughput and client response times.

As shown in Figure, 4.7(a) *abort rst-syn* increases the peak throughput of the μserver by 11% compared to *abort*, and offers around 20% improvement after the peak, which increases as the request rate is increased. Similarly, *reserv rst-syn* increases the peak throughput of the μserver by 11% compared to *reserv*, and provides more than 24% improvement in throughput after the peak. While the peak throughput of TUX, shown in 4.8(a), is not changed, *abort rst-syn* results in close to 10% improvement after the peak. The throughput in TUX with *reserv rst-syn* is also more than 12% higher compared to *reserv*, and this gap increases at higher rates.

Note that the improvement in throughput with *rst-syn* is consistent across the board for all request rates, and increases at higher request rates. When compared to *default*, *abort rst-syn* results in more than a 27% improvement in throughput after the peak



(a) Throughput

(b) Response Time

(c) Errors

(d) Queue Drops

Figure 4.8: Evaluation of TUX with alternatives to silent SYN drop

in the μserver, and around an 8% increase in throughput after the peak in TUX. The corresponding numbers for improvements in throughput provided by ***reserv rst-syn*** over ***default*** are 35% in the μserver and 15% in TUX. 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. In ***reserv rst-syn*** we reject a connection attempt at the SYN stage instead of the ACK stage, resulting in one less packet sent and received by the server per connection, hence, its throughput is slightly better than that of ***abort rst-syn***. The gains obtained from ***rst-syn*** can be explained by the general principle of dropping a connection as early as possible under overload [63].

As shown in Figures 4.7(b) and 4.8(b), the response time is reduced by two orders of magnitude with ***rst-syn*** in both the μserver and TUX. Recall that by response time, we imply the average response time measured at clients for successful connections. The response time is the sum of the connection establishment time and the 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. Once a server accepts a client connection (i.e., puts it on the listen queue), the amount of time spent in transferring data is small for the one-packet workload compared to the connection time, so it does not affect the response time significantly. On an average it takes around 3 seconds for the client to establish a connection, but less than 100 milliseconds to get a response once the connection has been established, when either ***default*** or ***abort*** are used at the server. On the other hand, with ***rst-syn***, a client connection attempt fails immediately. 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. We can get higher throughput as well as lower response time with ***rst-syn*** because the server is under persistent overload. The server can reject some clients immediately, but it 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.

Figure 4.7(c) demonstrates using the μserver that rejecting connections at the SYN stage does not result in a higher error rate. That is, 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***. The error rate when TUX is used in conjunction with ***abort*** and ***abort rst-syn*** (shown in Figure 4.8(c)) is higher than ***default*** because as described in Section 4.1.3, aborting connections when SYN/ACK ACKs cannot be accommodated in the listen queue is counter productive in TUX. That is, the increase in the error rate is due to ***abort***, not because of ***rst-syn***. Note that the error rate with ***reserv*** and ***reserv rst-syn*** is lower than that with ***default***, even in TUX. As expected, the error rate is equal to the rate of queue drops with ***abort rst-syn*** and ***reserv rst-syn*** because the retransmission of connection establishment segments is eliminated.

It is instructive to examine how many connection attempts, both successful as well as failed, require more than one SYN transmission with ***default*** or ***abort***. This can

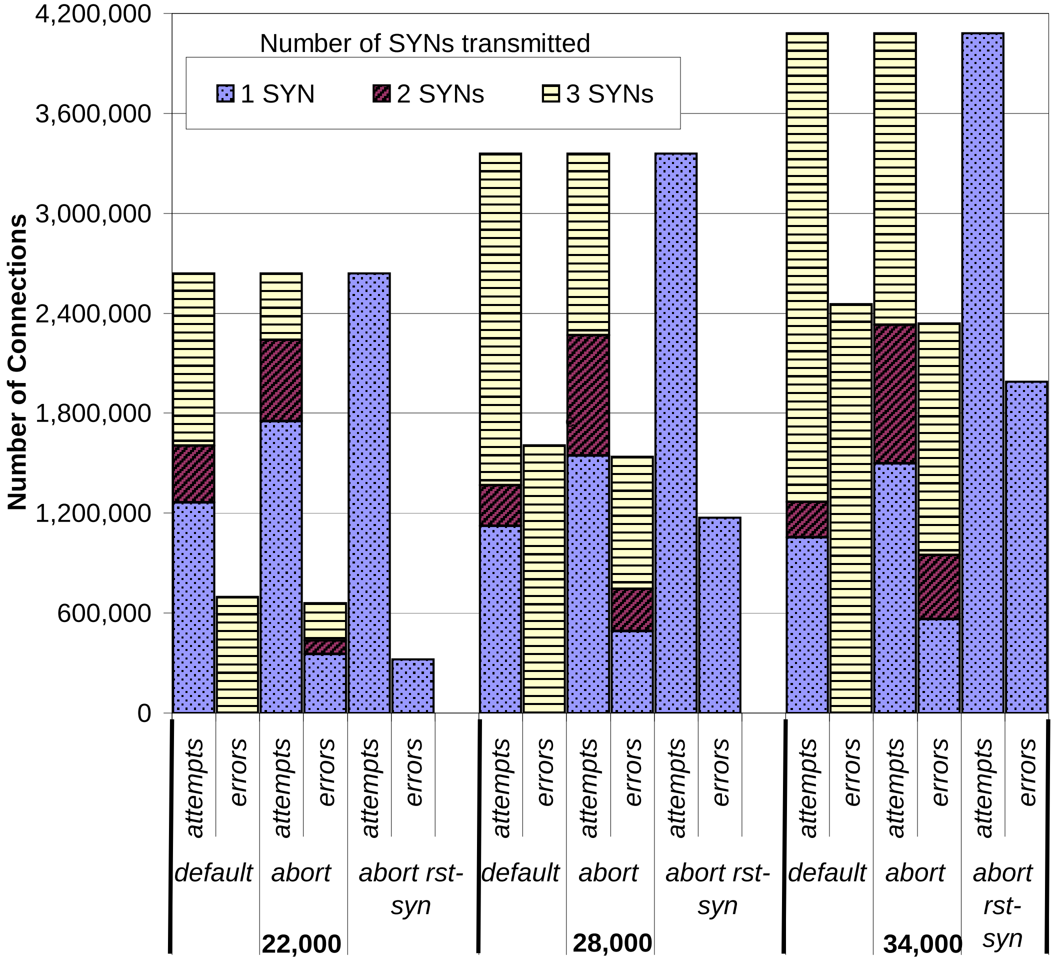


Figure 4.9: Breakdown of connection attempts and failures based on number of SYNs transmitted with *default*, *abort*, and *abort rst-syn* in TUX for three different request rates

provide a better understanding of why the throughput is lower and the client response times are higher with both these mechanisms when compared to *abort rst-syn* (or *reserv rst-syn*). Figure 4.9 shows a breakdown of all connection attempts and failures (errors) based on the number of SYN segments transmitted using *default*, *abort*, and *abort rst-syn* while running the µserver. Note that the connection attempts and errors for each mechanism are shown for three different request rates, namely, 22,000, 28,000, and 34,000.

Recall that a 10 second timeout allows at most three SYN transmissions, two of which occur after the previous SYN retransmission timeout expires. Recall also that the number of connections attempted is determined based on the request rate and the duration for which that rate was run. Connection attempts include all connections that were: (i) successfully serviced, (ii) rejected (either through ***abort*** or ***rst-syn***) by the server, or (iii) timed out by the client. Rejected or timed out connections add up to failed connections (i.e., errors). Any connection, whether successful or rejected that lasts for less than 3 seconds requires only one SYN transmission (i.e., the original connection attempt). Similarly those connections that last for more than 6 seconds and 9 seconds require two and three SYN transmissions, respectively. Finally, a connection that has to be timed out by the application results in three SYN transmissions. We accordingly added counters to `httperf` to categorize connections based on how many SYNs they transmitted. Note that each connection is counted exactly once based on the number of SYN transmissions. For example, a connection that is established on the third SYN retransmission, and gets a response from the server after 9.5 seconds is categorized as requiring three SYN transmissions.

The number of connections attempted ("attempts") by ***default***, ***abort***, as well as ***abort rst-syn*** is same for each request rate shown in Figure 4.9. On the other hand, the number of failed connections ("errors") differs in these mechanisms, it is the lowest with ***abort rst-syn***. The number of successful connections can be obtained by factoring out the "errors" bar from the "attempts" bar. For example, with ***default***, all connection attempts that require only two SYN transmissions represent successful connections because there are no connections which fail after just two transmissions. All client connections require a single SYN transmission with ***abort rst-syn***, after which they are either successfully serviced or reset by the server (using ***rst-syn***). Note that the work involved at the server to deliver data to the clients in the one-packet workload is minimal, hence, no connections timeout *after* they are established. With ***default***, a failed connection requires three SYN retransmissions, after which the client application gives up and times out a connection for which its TCP stack has not received a SYN/ACK from the server (i.e., a connection where the application's `connect()` call does not succeed for 10 seconds). With ***abort***, a connection can fail for two reasons – if a client does not receive a SYN/ACK after three SYN transmissions, or if it does receive a SYN/ACK on *any* of its SYN transmissions, but the server responds to its subsequent SYN/ACK ACK with a RST due to a listen queue overflow.

Figure 4.9 shows that as the load exerted on the server increases, a typical connection attempt involves the transmission of more than one SYN segment with *default* as well as with *abort*. At 22,000 request/sec, the percentage of connections requiring only one SYN transmission is 48% with *default* and 66% with *abort*. However, at 34,000 requests/sec, this figure drops to just 26% with *default* and 37% with *abort*. In comparison, the percentage of connections requiring three SYN retransmissions (which consists mostly of failed connections) is 69% with *default* and 43% with *abort*. As described earlier, with *abort rst-syn*, all connections (both successful and failed) require a single SYN transmission. The lack of SYN retransmissions not only reduces the client response times, but also allows the server to spend more time on established connections, resulting in higher throughput.

Recall from Section 3.1.1, that we have not configured TUX and μserver to facilitate a direct comparison. While higher response times obtained with TUX (shown in Figure 4.8(b)) in comparison to those obtained with the μserver (shown in Figure 4.7(b)) may appear to be suggestive, this difference is because of the different configurations used in both the servers. Both the servers use different listen queue sizes – 128 in the μserver compared to 2048 in TUX[21]. The larger listen queue allows more connections to buffered before TUX can work on them, thereby increasing the average response time. That is, there is an inverse relationship between the number of connections buffered in the listen queue and the average response time for those connections[22]. The response times obtained with TUX using a smaller listen queue (with 128 entries) is comparable to those obtained with the μserver.

Thus, *rst-syn* is effective in improving server throughput and reducing client response times on *regular* clients by eliminating SYN retransmissions. As discussed in Section 4.1.7, there are some security implications of using *reserv* (with or without *rst-syn*) in production environments. While *abort* takes a reactive approach to SYN/ACK ACK drops rather than avoiding them, it is significantly easier to implement than *reserv*. We

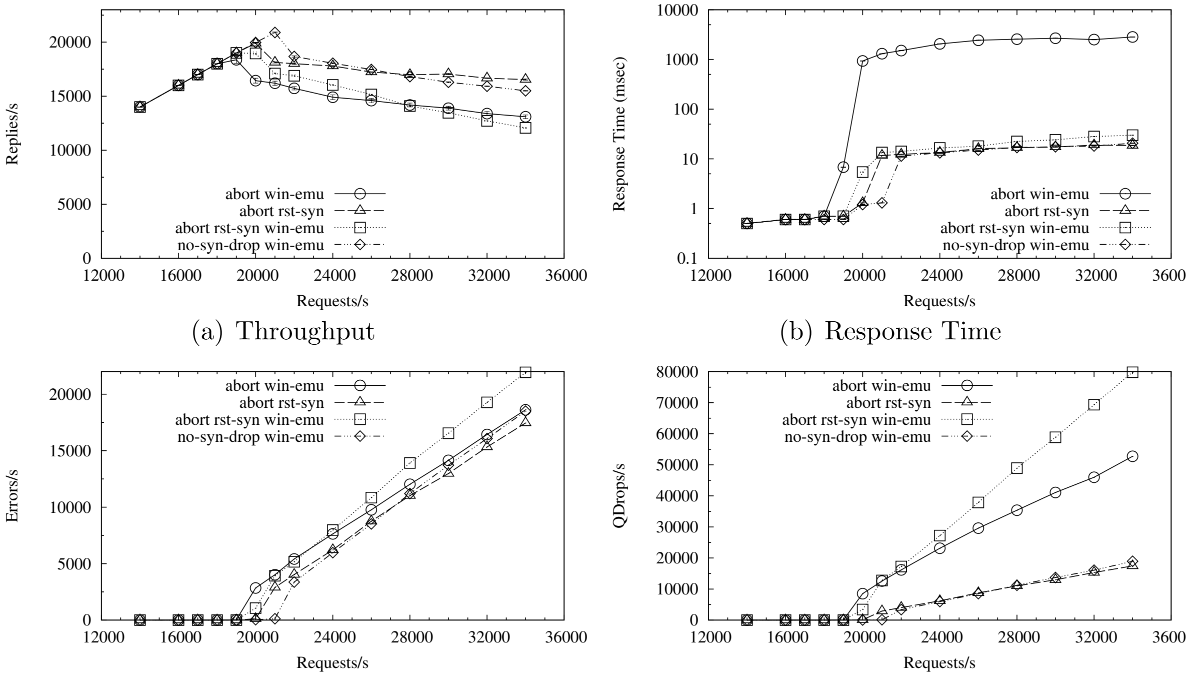
[21] Recall that the listen queue sizes were chosen to be the default values in Linux.

[22] This relationship suggests that making the queue sizes too large is counter productive because it increases client response times and might result in more timed out connections. This result is well known in queueing theory.

consider *abort rst-syn* to be a more viable mechanism that adequately addresses server performance as well as security requirements. Hence, in the remainder of this thesis, we only present the results of *abort rst-syn*.

Windows-like Clients

In this section, we compare the results of *abort rst-syn* and *no-syn-drop* against *abort*[23] when used with the *win-emu* clients. We only present the µserver results in this section. TUX results are similar and do not provide additional insight, so we omit them. We also include the results of *abort rst-syn* on *regular* clients (denoted "*abort rst-syn*") for comparison. Figure 4.10 summarizes these results.



(a) Throughput

(b) Response Time

(c) Errors

(d) Queue Drops

Figure 4.10: Evaluation of the µserver with *win-emu* clients

Figure 4.10(a) shows that the throughput with *abort rst-syn win-emu* is lower than

[23] We chose *abort* because it is implemented in most UNIX TCP stacks, noting that *default* has slightly lower throughput than *abort*.

that with *abort rst-syn*. This is because of the SYNs retried by *win-emu* clients upon receiving a RST. The throughput provided by *abort rst-syn win-emu* is initially comparable to that obtained with *abort win-emu*, however, as the request rate increases, its throughput drops below that obtained with *abort win-emu*. A client that does not receive a SYN/ACK in response to its SYN retransmits a SYN at most twice with *abort win-emu*, but up to three times with *abort rst-syn win-emu*, which explains the gap in the queue drops between these two mechanisms in Figure 4.10(d). As shown in Figure 4.10(b), the client response time is reduced by two order of magnitude with *abort rst-syn* compared to *abort*, even on *win-emu* clients. Windows TCP stacks (and hence our emulation thereof) 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[24].

By avoiding the retransmission of 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. This demonstrates that it is possible to deploy mechanisms that are effective in reducing response time and improving server throughput in currently deployed client TCP stacks, including those on Windows. A mechanism like *no-syn-drop* obviates the need for techniques to filter retransmitted SYNs with a lower overhead, such as those proposed by Jamjoom and Shin [45]. The efficiency of techniques such as SYN policing [85] can also be improved by using *no-syn-drop* to eliminate retransmitted connection attempts by clients[25].

[24] *abort rst-syn* will fail to improve the response times with a *win-syn-retry* TCP stack that uses an exponential backoff for the resending a SYN in response to a RST. However, we are not aware of such a TCP stack.

[25] Note that with *no-syn-drop*, we might have to perform SYN/ACK ACK policing instead of SYN policing.

4.1.7 Security Implications of Connection Establishment Mechanisms

In recent years, resilience in the face of Denial of Service (DoS) attacks, especially SYN flood attacks by malicious clients has been a major criteria influencing the implementation choices for connection establishment in many server TCP stacks [54]. Techniques such as SYN cache, SYN cookies, and early discard of invalid ACK segments have been proposed to prevent malicious clients from consuming most of the server resources, thereby denying service to legitimate clients. In this section we briefly discuss the security implications of the connection establishment mechanisms discussed in this thesis, focusing on the susceptibility of the server to SYN flood attacks. Note that we do not consider generic denial of service attacks (e.g., attacks attempting to flood the server's network link with invalid TCP/IP packets) because the server TCP stack, by itself, is unable to provide protection against such attacks without an external firewall or packet filtering mechanism. As described in Section 2.2.3, protecting servers from generic denial of service attacks is still an area of active research.

In a SYN flood attack malicious clients attempt to overwhelm the server by sending a constant stream of TCP SYN requests. The source address in these requests is typically forged to be non-existent or unreachable hosts, leaving the server with many incomplete connections in its SYN queue, each waiting for a SYN/ACK response. Each of these SYN queue entries not only utilize server resources but they also fill up the SYN queue, thereby increasing the probability of the subsequent dropping of SYNs from legitimate clients. As described in Section 4.1.5, TCP SYN cookies [14, 54] can be used to obtain a virtually unbounded SYN queue without storing any information about incomplete connections at the server. Current production TCP stacks (in many UNIX-like systems) already use SYN cookies in conjunction with mechanisms such as ***default*** and ***abort*** to protect the server from SYN flood attacks.

This leads to the following question – *Are the two new connection establishment mechanisms described in this thesis, **no-syn-drop** and **reserv** vulnerable to SYN flood attacks*? The ***no-syn-drop*** mechanism guards against SYN flood attacks by relying on SYN cookies. It is no more vulnerable to other network flooding attacks than other connection establishment mechanism. Our current implementation of ***reserv***, on the other hand, does introduce additional security concerns because it assumes that all clients that receive a SYN/ACK respond immediately with a SYN/ACK ACK. By sending bogus SYN segments, malicious clients can not only occupy entries in the SYN queue, they can also consume listen queue reservations. This can cause SYN segments from legitimate clients to be dropped because no listen queue reservations are available (in addition to SYN drops because the SYN queue is full). Note that the size of the listen queue is typically smaller than the size of the SYN queue in most TCP stacks. All of the proactive solutions to avoid listen queue overflows, discussed in Section 4.1.2, are similarly vulnerable to SYN flood attacks. There is a trade-off between the extra work (e.g., processing and sending a RST in response to a SYN/ACK ACK that is dropped) and involved in reactive approaches to listen queue overflows such as ***abort*** and the vulnerability to SYN flood attacks entailed by proactive solutions.

We could use SYN cookies to get an unbounded SYN queue in conjunction with an overbooking of the listen queue, a probabilistic drop of SYNs (similar to Random Early Drop [33]), and a periodic clean up of least-recently-used items, to improve the robustness of ***reserv*** against SYN flood attacks. However, given the complexity involved with such an implementation of ***reserv*** and the fact that server performance with ***abort*** is close to that obtained with ***reserv*** (when both are used in conjunction with ***rst-syn***), it is unclear whether there are any real advantages[26] to incorporating ***reserv*** in current production TCP stacks. If network-based proposals to counter denial of service attacks such as ingress and egress filtering, rate limiting, and unicast reverse path forwarding become widespread and succeed in ensuring that all TCP segments that make their way to the server are from legitimate clients, it would be feasible to deploy our existing ***reserv*** implementation, without requiring any security-related changes.

[26] Note that *reserv* is more correct than *abort* based on a strict interpretation of RFC 793.

4.2 Impact of TCP Connection Termination on Server Performance

In this section, we examine alternatives to the standard connection termination mechanism suggested in RFC 793, at the server TCP stack as well as in the client application, and evaluate their impact on the throughput of overloaded servers

As demonstrated in Figure 2.3 and as described in Section 2.1.2, a graceful connection termination by the client, either with half-closed connection semantics (through the `shutdown()` system call) or in half-duplex fashion (through the `close()` system call), results in a FIN being sent to the server. The server TCP stack has no way of knowing whether the client is using half-closed connection semantics. Most server TCP stacks assume that connections are terminated using half-closed semantics by the clients.

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 the server TCP stack supports half-closed connections, it continues to make queued data available to the server application (i.e., the web server) through `read()` calls, even after receiving a FIN from a client. Only when the data queue is completely drained will `read()` return EOF (0) notifying the server application that the client has terminated the connection. Any prior non-zero `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 application does not use half-closed connection semantics for terminating a connection.

Figure 4.11 illustrates the flow of TCP segments from a client (s01) that has timed out and terminated its connection through a `close()` call (line 6). The server (suzuka) TCP stack immediately acknowledges the client's FIN (line 7). After 25 seconds, the server application is able to `read()` the request, and writes the appropriate response (line 8). The following `read()` returns EOF, so the server application issues a `close()` call to terminate its end of the connection (line 9). The client application is not using half-closed connection

```
(1) 12:41:57.297282 s01.1024 > suzuka.55000: S 1159260550:1159260550(0) win 5840
(2) 12:41:57.297305 suzuka.55000 > s01.1024: S 882478789:882478789(0)
    ack 1159260551 win 5792
(3) 12:41:57.297531 s01.1024 > suzuka.55000: . ack 1 win 5840
(4) 12:41:57.297532 s01.1024 > suzuka.55000: P 1:80(79) ack 1 win 5840
(5) 12:41:57.297554 suzuka.55000 > s01.1024: . ack 80 win 46
(6) 12:41:58.797487 s01.1024 > suzuka.55000: F 80:80(0) ack 1 win 5840
(7) 12:41:58.837288 suzuka.55000 > s01.1024: . ack 81 win 46
(8) 12:42:23.079113 suzuka.55000 > s01.1024: P 1:1013(1012) ack 81 win 46
(9) 12:42:23.079131 suzuka.55000 > s01.1024: F 1013:1013(0) ack 81 win 46
(10) 12:42:23.079240 s01.1024 > suzuka.55000: R 1159260631:1159260631(0) win 0
(11) 12:42:23.079364 s01.1024 > suzuka.55000: R 1159260631:1159260631(0) win 0
```

Figure 4.11: `tcpdump` output of a client that times out

semantics so the client TCP stack responds with a RST[27] to every server reply (lines 10, 11).

Client-initiated connection termination is common for HTTP 1.1 persistent connections. In most server TCP stacks, this can result in web servers wasting resources on connections even after they have been terminated by the client. For HTTP 1.0 connections, the server typically initiates the connection termination after sending a response. Note that studies have advocated the use of client-initiated connection termination to prevent the server from having many connections in the TIME_WAIT state [31, 81]. It is also possible for both sides to simultaneously terminate their end of the connection. While it is unlikely to happen in a browser-web server environment, a simultaneous connection termination obviates the need for the server to support half-closed connections.

4.2.1 Disabling Support for Half-Closed Connections

We describe a mechanism that can help isolate the impact of supporting half-closed connections on server throughput under overload in this section. In particular, we are interested in answering the following question – *If the server TCP stack were to drop support for*

[27] Note that a RST from the client can cause subsequent system calls by the server application on that socket to fail with an `ECONNRESET` error.

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 problems with *not* supporting half-closed connections. Disabling support for *all* half-closed connections at the server will break clients that do rely on half-closed semantics (i.e., they will be unable to receive data from the server following a FIN)[28]. 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 thesis, we disable support for 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 (i.e., connections which have been terminated through a `close()` call without relying on half-closed semantics). 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. That is 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[29]. Alternatively, the TCP protocol 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 TCP segment such as one with the FIN and RST flags both set). We perform experimental evaluation in this thesis to determine whether such approaches are worth pursuing.

Assuming that *most* clients will not use half-closed connection semantics is not unreasonable. Stevens [80] points out that most client applications (except for a few remote shell implementations) use the `close()` system call instead of using half-closed connections. Zandy and Miller [90] describe a socket enhancement detection protocol that relies partly on the fact that very few applications `read()` from half-closed connections following a connection termination. All the modern web browsers that we are aware of do not use half-closed semantics. Thus, most client applications will continue to work correctly after the support for half-closed connections is disabled at the server TCP stack.

[28] Additional problems with not supporting half-closed connections are outlined in Section 4.2.2.

[29] Specifically the listening socket and all the sockets derived from it, including those that refer to client connections.

We refer to our connection termination policy that does not support half-closed connections as ***rst-fin***[30]. Our current implementation of ***rst-syn*** disables support for all half-closed connections in the server TCP stack. That is, we use a system-wide parameter configurable through the `net.ipv4.tcp_rst_to_fin` sysctl to effect ***rst-fin***, this option was chosen for the ease of implementation. As alluded to earlier, we intend to implement ***rst-fin*** as an application-specific option that can be configured through the `setsockopt()` system call in the future. This would allow Internet servers fine-grained control over support for half-closed connections by selecting ***rst-fin*** based on application semantics.

When ***rst-fin*** is enabled all FIN segments that do not have piggy-backed data (i.e., a segment containing data along with a FIN flag) 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 server 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 (i.e., no information is propagated by the client TCP stack to the application), provided the socket descriptor has been destroyed using a `close()` call. Figure 4.12(a) illustrates how we disable support for half-closed connections.

Note that when ***rst-fin*** is enabled, the server TCP stack sends a RST in response to a FIN instead of sending a FIN/ACK segment. This allow clients which are expecting half-closed connection semantics (i.e., waiting for a `read()` following a `shutdown()`) to be notified of an abnormal connection termination, instead of getting an EOF marker indicating that the server has no more data to send. Sending a RST in our opinion, is not only more correct, but it also frees the server from having to process a final ACK (in response to the FIN sent) from the client. In this way, ***rst-fin*** can reduce the amount of time the server spends processing TCP segments on connections that clients do not care

[30] An abbreviation of "RST to FIN".

about.

If there is data piggybacked with the FIN from a client, the server TCP stack does not invoke ***rst-fin***, but follows the usual connection teardown process that permits half-closed connections. This is done to avoid penalizing client TCP stacks that implement the connection termination optimization described by Nahum et al. [66] (summarized in Section 2.2.2), and piggyback their last data segment on the FIN. Note also that when a RST is sent in response to a FIN, the TCP connection transitions to the `CLOSED` state. Hence, if this RST is lost in a WAN environment, the subsequent retransmission(s) of FIN (when the FIN retransmission timeout expires) from the client will receive a RST from the server TCP stack.

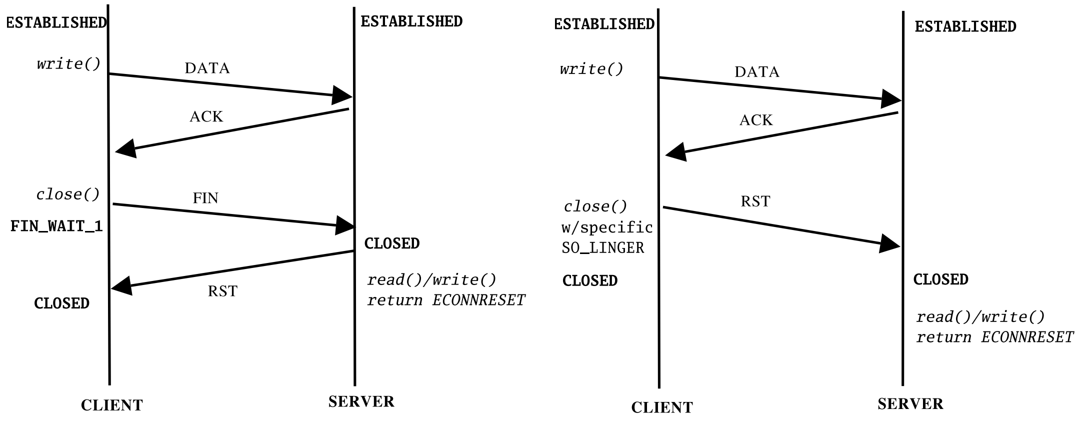
4.2.2 Connection Termination through Abortive Release

Some browsers such as Internet Explorer 5 and 6 terminate connections through an abortive release [8]. 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[31]. The abortive release procedure is illustrated in Figure 4.12(b). Applications typically use an abortive release in response to abnormal events, for example, 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, for example, when the browser window is closed by the user, or a routine event, for example, when the user hits the stop button to stop loading a page or when the browser closes a connection due to a (browser-based) timeout. 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" [32].

A RST from the client precludes support for half-closed connections because the server TCP stack immediately throws away all information associated with that connection, and subsequent system calls on that connection return `ECONNRESET` to the server application.

[31] An abortive release thus corresponds to the ABORT operation specified in RFC 793.



(a) `rst-fin` Termination (b) Abortive Release (`close-rst`)

Figure 4.12: Alternatives to supporting half-closed connections

Many socket APIs provide an option to terminate connections through an abortive release. In the BSD socket API, applications can activate the linger parameter in the `SO_LINGER` socket option with a zero timeout to force the subsequent `close()` call on that socket to have the abortive release semantics.

The reasons why Internet Explorer terminates all connections in an abortive fashion are not clear. The argument that doing so prevents servers from being stuck with a lot of connections in `TIME_WAIT` state (suggested in [8]) is incorrect because the connection termination is initiated by the client. Only sides that initiate graceful connection termination have to transition to the `TIME_WAIT` state. There are usually enough ephemeral ports at the client to allow connections in `TIME_WAIT` state, hence the client application need not perform an abortive release when terminating a connection. A RST sent by the client does bypass the `CLOSE_WAIT` and `LAST_ACK` TCP states shown in Figure 2.3 at the server, allowing the server TCP stack to transition directly to the `CLOSED` state. As alluded to earlier, 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 [8]), we try to answer the following question – *Does an abortive connection termination by client* *applications improve server throughput?* We use the `close-with-reset` option provided in `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***[32].

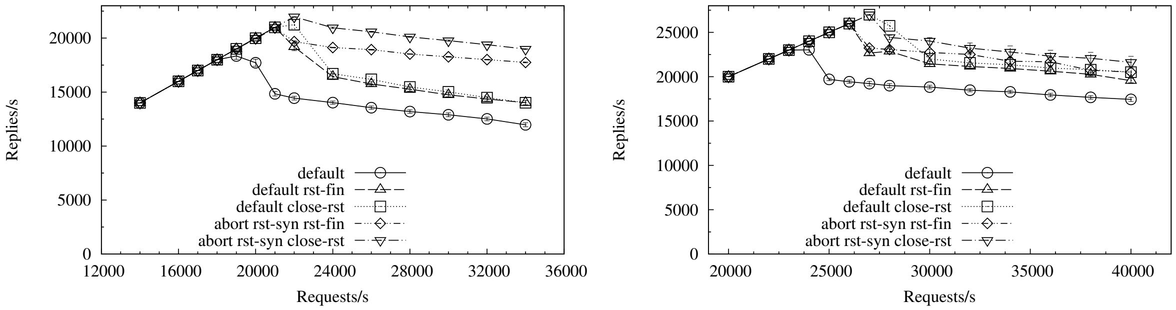
It is important to note that both ***rst-fin*** and ***close-rst*** can result in lost data if data segments arrive (out of order) on a connection after a FIN or RST, or if there is data pending in the socket buffer that has not been read (or written) by the application when a FIN or a RST is received 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) or when clients terminate connections on a timeout. It might not be appropriate in other application-level protocols or protocol primitives that require reliable delivery of data sent prior to connection termination, and do not use application-level acknowledgements. We reiterate that we take an exploratory approach toward studying the impact of TCP connection termination mechanisms on server throughput, not a prescriptive one.

4.2.3 Evaluation of Alternative Connection Termination Mechanisms

Dropping 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. For completeness, we do include a response-time graph toward the end of this section to demonstrate that it is not affected by connection termination mechanisms.

Figure 4.13 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

[32] An abbreviation of "client connection close with RST".



(a) μserver

(b) TUX

Figure 4.13: Evaluation of alternative connection termination mechanisms

establishment mechanisms, in particular, to check if a particular connection termination mechanism 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 in combination with *`abort rst-syn`*.

Figures 4.13(a) and 4.13(b) demonstrate that when compared to *`default`*, *`default rst-fin`* results in a 17% improvement in the peak throughput with the μserver and 13% improvement in peak throughput with TUX. 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 which were serviced successfully. For example, with *`default`* in the μserver at 30,000 requests/sec, 1,659,594 out of the 6,638,376 total TCP segments received after connection establishment represent a client acknowledgement of the server's FIN. The processing of these ACKs is avoided with *`abort rst-fin`*. 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`* in both the μserver and TUX.

It is also evident that connection termination mechanisms complement connection establishment mechanisms. The throughput yielded by both *`rst-fin`* and *`close-rst`* increases when they are coupled with *`abort rst-syn`*. The improvement in throughput is especially significant in the μserver. As shown in Figure 4.13(a), when compared to

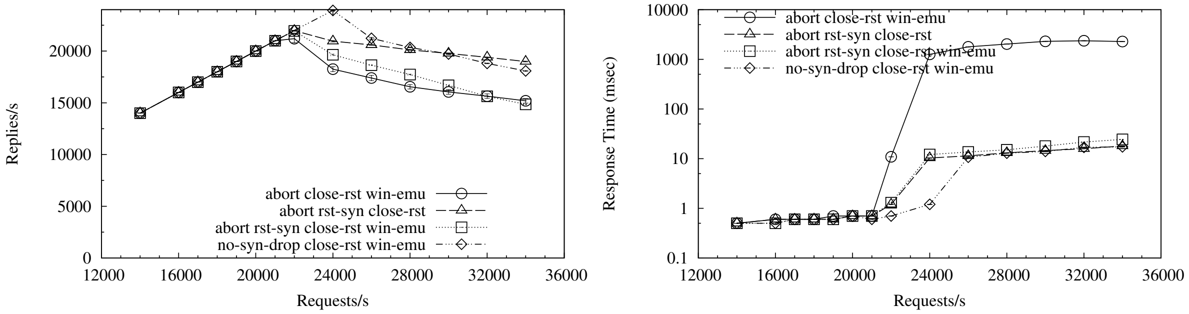
default rst-fin, ***abort rst-syn rst-fin*** provides more than 20% higher throughput (after peak), and ***abort rst-syn close-rst*** provides close to 30% higher throughput. Note that with ***abort rst-syn close-rst***, the server is able to stop working on a connection upon receiving a RST from a client without having to transmit any subsequent TCP segments. On the other hand, with ***abort rst-syn rst-fin***, the server TCP stack has to transmit an additional RST after receiving a client FIN. The increase in throughput in TUX obtained with ***rst-fin*** and ***close-rst*** when used in conjunction with ***abort rst-syn*** is comparatively smaller (around 5%).

The results of this section suggest that stopping work on connections immediately after they have been terminated by clients can improve the throughput of an overloaded server. The server can stop working on terminated connections by disabling support for half-closed TCP connections (on a per-application basis), or if "cooperative" clients use an abortive release for tearing down connections. Alternatively, existing event notification mechanisms can be enhanced to notify a server application of a connection terminated by a client *before* it makes further system calls on the socket associated with that connection, for example, by adding a `POLL_FIN` bit to the `poll()` system call. The server application could then choose to terminate the client connection immediately, dropping support for half-closed connections in the process.

Emulation of Typical Web Clients

In this section, we review the impact of different connection management mechanisms on the performance of overload servers when used with typical (as of 2005) web clients – Internet Explorer browsers running on Microsoft Windows operating systems [8]. That is, we use Linux clients that emulate the ***win-syn-retry*** behaviour (i.e., ***win-emu*** clients) along with `httperf` configured to terminate connections using an abortive release (i.e., ***close-rst***). These typical web clients will be denoted as ***close-rst win-emu*** in the following discussion.

We evaluate the following connection establishment mechanisms with typical web clients – ***abort***, which is implemented in most UNIX TCP stacks (and is a configurable option in Linux), ***abort rst-syn***, which is used in some Windows TCP stacks, and ***no-syn-drop***, which to the best of our knowledge is not deployed in any existing TCP stacks, but can be



(a) Throughput

(b) Response Time

Figure 4.14: Evaluation of the μserver with typical web clients

used to work around *win-syn-retry* clients. We only include the results obtained with the μserver in Figure 4.14, the results obtained with TUX are similar.

As alluded to earlier, the connection termination mechanism used does not affect the response times. The results of *abort close-rst win-emu* in Figure 4.14(b) are comparable to those of *abort win-emu* in Figure 4.10(b). By eliminating the retransmission of connection establishment segments, *abort rst-syn* and *no-syn-drop* provide two orders of magnitude reduction in response times compared to *abort*. The throughput of *abort rst-syn* is lower with typical web clients due to the overhead of handling SYN segments retried by *win-syn-retry* TCP stacks. As described in Section 4.1.6, by preventing the TCP-level retransmission of connection attempts by clients, *no-syn-drop* can provide more than a 20% increase in throughput (after the peak) compared to *abort* on typical web clients.

In summary, mechanisms that avoid the retransmission of connection attempts from client TCP stacks can improve server throughput and reduce client response times under overload. Unfortunately, *win-syn-retry* client TCP stacks, which do not follow the recommendations of RFC 793 with respect to handling of a RST received in response to a SYN, can nullify the effectiveness of *rst-syn*. However, *no-syn-drop* can be used in such cases to improve server throughput by up to 40% and reduce client response times by two orders of magnitude. Note that although its approach might be ungainly, *no-syn-drop* works equally well with *regular* as well as *win-syn-retry* clients. We have also demonstrated that disabling support for half-closed connections and the abortive release of connections by client applications can improve the throughput of an overload server by more than 15%. In the next chapter, we will revisit some of these connection management mechanisms and evaluate them under different operating environments than the ones used in this chapter.

Chapter 5

Performance Evaluation Under Different Environments

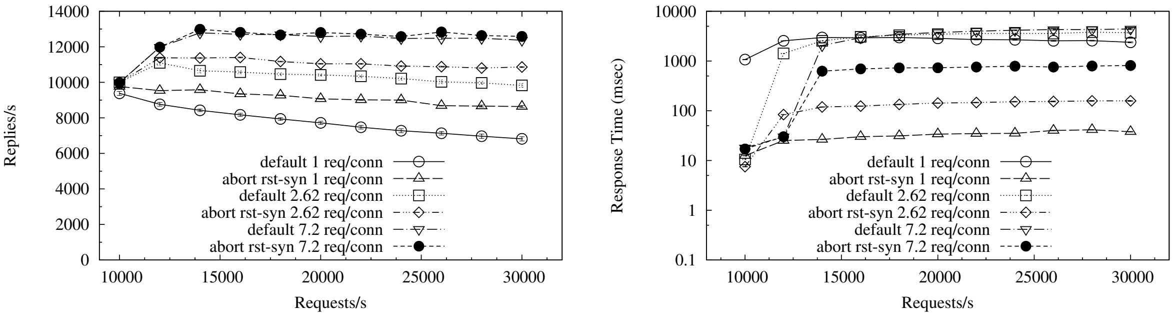
In this chapter, we evaluate some of the connection management mechanisms described in Chapter 4 under different environments. For the experiments presented in Chapter 4, we used a single workload (namely, the one-packet workload) to effect persistent overload with a 10 second client timeout on a Xeon (x86) machine running Linux 2.4.22 as our server. In this chapter, we examine whether the implementation choices for connection management, especially connection establishment, continue to have an impact on server throughput and client response times when used with different workloads, bursty traffic, different client timeouts, early packet drops, and different platforms. Note that for clarity, we only modify one parameter at a time in each of the following sections.

For brevity, we only present results of the μserver on ***regular*** clients in this chapter. The TUX results are qualitatively similar and do not provide additional insight. Also, as demonstrated in Section 4.1.6, the ***no-syn-drop*** mechanism can be used to achieve the effect of ***rst-syn*** on Windows-like (***win-syn-retry***) clients. Thus, in this chapter we focus on presenting the throughput and response time results obtained with the μserver under different environments. The analysis of the results with different connection management mechanisms (using supporting metrics such as error rate and queue drops) was provided in Chapter 4, we will not repeat that discussion in this chapter.

5.1 SPECweb99-like Workloads

In this section, we present the results of connection management mechanisms on the in-memory, SPECweb99-like workloads. As described in Section 3.2.2, we use three SPECweb-99-like workloads, which differ in the average number of requests per connection. The single-request-per-connection (referred to as "1 req/conn") workload extends the one-packet workload with variable-sized file transfers. The 7.2-requests-per-connection (referred to as "7.2 req/conn") workload uses the SPECweb99-stipulated 7.2 average requests per connections. The 2.62-requests-per-connection (referred to as "2.62 req/conn) workload uses an average of 2.62 requests per connection to mimic the behaviour of current web browsers, particularly Internet Explorer.

In Figure 5.1, we compare the throughput and response times obtained with ***abort rst-syn*** to that obtained with ***default*** on the three SPECweb99-like workloads. Recall that ***abort rst-syn*** seeks to eliminate the retransmission of SYN segments by client TCP stacks, while ***default*** represents the default connection establishment mechanism in Linux.



(a) Throughput

(b) Response Time

Figure 5.1: Evaluation of connection establishment mechanisms with the μserver on SPECweb99-like workloads

Figure 5.1(a) shows that ***abort rst-syn*** does not improve upon the throughput of ***default*** on the "7.2 req/conn" workload. This is not surprising, given that this workload does not aggressively create new connections. Even when the target request rate is 30,000 requests/sec, the rate of new connections is substantially smaller (only around 3,900 connections/sec). As a result, the implementation choices for connection establishment do not have a significant impact on server throughput. However, the response time with ***abort*** *rst-syn* is significantly (around 80%) lower than that obtained with *default*, in spite of the fact that the "7.2 req/conn" workload contains multiple file transfers with an average transfer size of 15 KB. That is, the transfer time for this workload is not negligible, yet the higher overall response time with *default* indicates that the transfer time is a small fraction of the connection establishment time. Thus, during overload the connection establishment time dominates the overall response time. Note that the reduction in response time is not because fewer clients are serviced. The error rate with *abort rst-syn* is in fact slightly lower than that with *default*, as explained in Section 4.1.6.

On the connection-oriented "1 req/conn" workload, *abort rst-syn* provides more than a 20% improvement in throughput after the peak and two orders of magnitude reduction in response time when compared with *default*. These differences are similar to those reported for the one-packet workload in Section 4.1.6, which indicates that the connection-oriented nature of these workloads dominates the amount of content transferred.

Finally for the "2.62 req/conn" workload, with *abort rst-syn*, the throughput after the peak is around 10% higher and the response time is two orders of magnitude lower when compared with *default*. As described in Section 3.2.2, while SPECweb99 incorporates many aspects of the workloads seen in production web sites, it overestimates the number of requests per connection issued by modern browsers. These results clearly indicate that the impact of the connection establishment mechanisms on the server throughput depends on the rate of new connections at the server. At the same time, mechanisms such as *abort rst-syn* provide a substantial reduction in client response times even on workloads that are not connection-oriented.

In order to evaluate whether immediately stopping work on connections that have been terminated by clients can improve server throughput on the SPECweb99-like workloads, we present the results of *default* and *default close-rst* in Figure 5.2. Note that the throughput obtained with *default rst-fin* was similar to that obtained with *default close-rst*, but we leave out its results from Figure 5.2 for clarity.

The difference in the throughput obtained with *default close-rst* and *default* is negligible for the "7.2 req/conn" workload, as expected, because this workload does not stress the connection management mechanisms of the server. By ensuring that the server does not spend resources on connections that have been terminated by the clients,

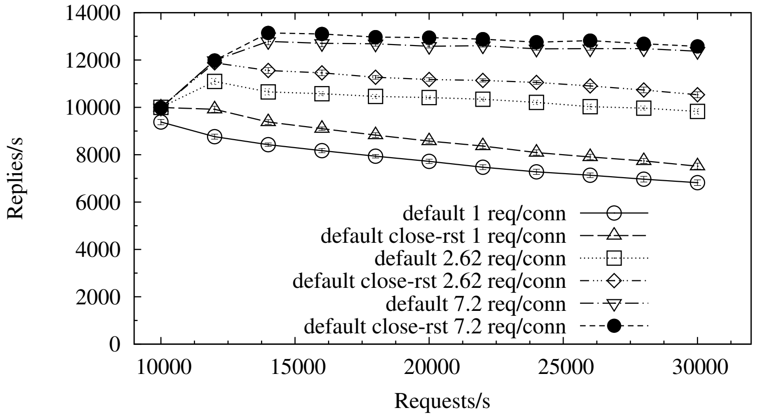


Figure 5.2: Evaluation of connection termination mechanisms with the μserver on SPECweb99-like workloads

close-rst can provide around an 8% improvement in throughput after the peak on the "2.62 req/conn" workload, and an 11% increase in throughput on the "1 req/conn" workload.

The results in this section indicate that given current browser and TCP stack implementations, connection establishment mechanisms such as ***abort rst-syn***, which prevent the TCP-level retransmission of connection attempts from clients, significantly reduce client response times and increase the throughput of web servers during overload, without requiring any additional effort on the part of web site administrators[1]. Additionally, an abortive release of connections by client applications (as well as disabling support for half-close connections at the server TCP stack) can result up to 10% improvement in the throughput of an overloaded server.

5.2 Bursty Traffic

We use persistent overload (described in Section 3.2.4) in our evaluation of connection management mechanisms in this thesis because it is representative of the load handled by real-world servers during flash crowds. During persistent overload, most of the retrans-

[1] Other than turning on the sysctls that enable these mechanisms, if they are not on by default.

mission of connection establishment attempts by the client TCP stacks occur while the server is still under overload. As demonstrated in Chapter 4, under persistent overload, connection establishment mechanisms such as ***abort rst-syn*** and ***no-syn-drop*** increase server throughput and reduce client response times by notifying client TCP stacks to eliminate the retransmission of connection attempts. In this section, we conduct a *preliminary* investigation of the performance of such mechanisms under bursty traffic. Under bursty traffic, overload conditions at the server are transient, with short-lived bursts of heavy traffic (overload bursts) interrupting protracted periods of light load. Unlike persistent overload, the duration of the bursts is short enough that the overload has subsided by the time the majority of clients retransmit their SYNs.

We stress that the results presented in this section are preliminary. It might be possible to size the SYN and listen queues to avoid queue drops during bursty traffic. Recall that the implementation choices for connection establishment come into play only when there are queue drops. We are in the process of developing a mathematical model for sizing these queues so as to minimize the number of queue drops. In this section, we use a simple method for generating bursts that allows us to compare the behaviour of ***abort rst-syn*** and ***default*** under bursty traffic with the default sizes for the SYN and listen queues (which were also used in all other experiments).

Figure 5.3 illustrates our simple approach for generating bursty traffic. It consists of two alternating phases – an overload burst that lasts for b seconds where the server receives sustained connection attempts at a rate of R' requests per second, and an inactivity period, where the server receives no client connections[2] for 5 seconds. Although in general inactivity periods can be variable length, we assume a fixed 5 second inactivity period to simplify our analysis. The server can handle some connections "left over" from the burst (especially those connections which are initiated toward the end of the burst), if they are retried by the clients during the inactivity period. By using an inactivity period instead of a period of light load, our model recreates a worst-case scenario for mechanisms such as ***abort rst-syn***, which abort client connection attempts immediately during overload (i.e., whenever queue drops occur), even though the server might in a position to handle

[2] Unfortunately, our workload generator does not support the creation of no connections, so we had to create one connection per second during the inactivity period.

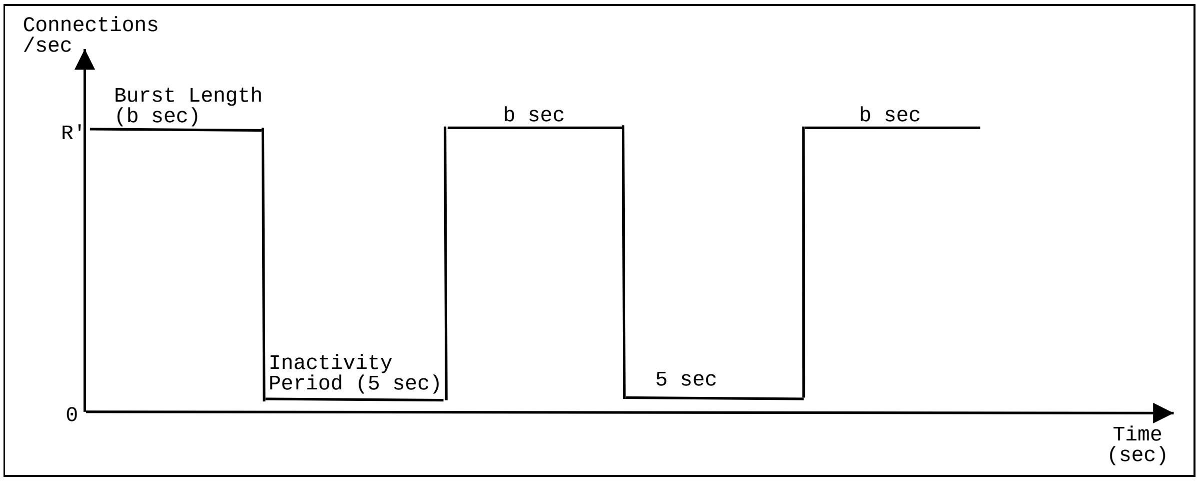


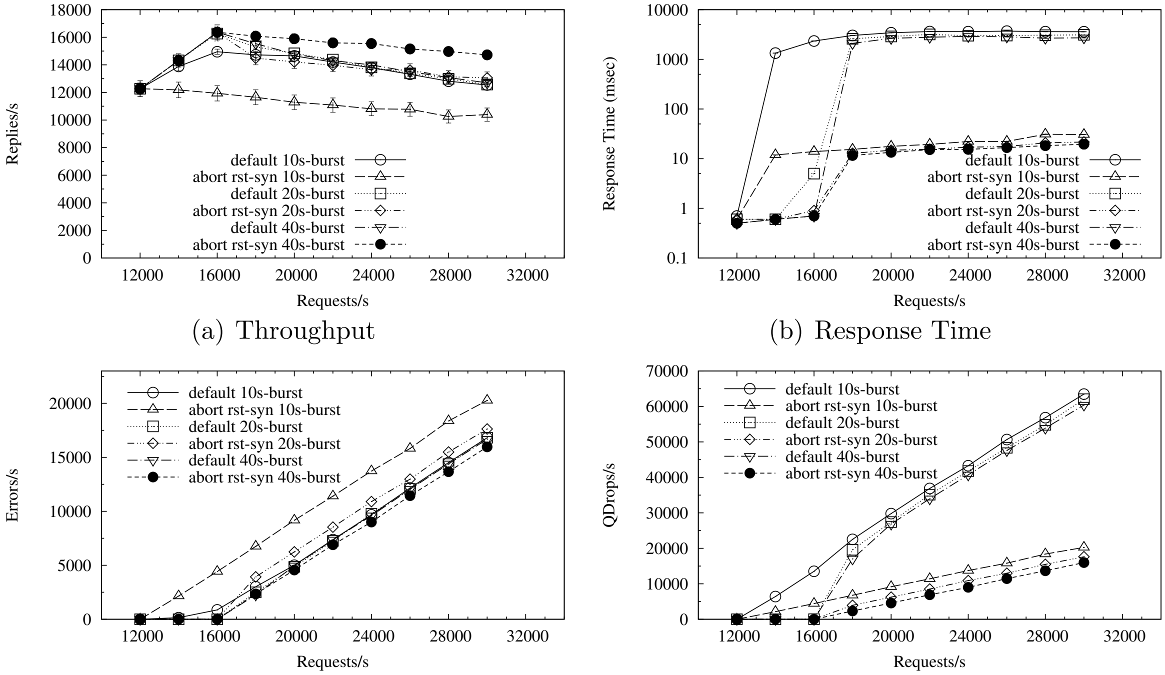
Figure 5.3: A simple approach for generating bursty traffic

them later.

We use three different burst lengths – 10 seconds (denoted "10s-burst"), 20 seconds (denoted "20s-burst"), and 40 seconds (denoted "40s-burst") to study how the duration of the overload burst relative to the inactivity period affects server throughput and response time. Note that each data point for the experiments in this section was run for 225 seconds to ensure that the number of bursts and inactivity periods is evenly balanced[3]. We also use a request rate (*R'*) during the burst to ensure that the average request rate achieved for a particular data point matches the target rate (*R*) specified to `httperf`. For example, with a 10 second burst length and a 5-second inactivity period, to achieve an overall target rate of 10,000 requests/sec, a net rate of 15,000 request/sec is used during the burst. Without an increase in the rate during the burst, the average request rate achieved would be much lower because requests are suppressed during the inactivity period. The results comparing ***abort rst-syn*** and ***default*** for the three burst lengths appear in Figure 5.4

As alluded to earlier for the 10 second burst, Figure 5.4(a) shows that ***abort rst-syn*** results in around 25% lower throughput compared to ***default***. By aborting connections every time there is a queue drop, ***abort rst-syn*** is able to reduce the queue drop rate as well as the response times (shown in Figures 5.4(d) and 5.4(b)). However, as shown in Figure 5.4(c), this reduction comes at the cost of a higher error rate because connections

[3] 225 is the least common multiple of 15, 25, and 45.



(a) Throughput

(b) Response Time

(c) Errors

(d) Queue Drops

Figure 5.4: Evaluation of the µserver under bursty traffic

are aborted immediately on a queue drop, so overall fewer clients get serviced with ***abort rst-syn*** for the 10 second burst workload. On the other hand, with ***default***, clients whose initial SYN segments are dropped can get a response from the server on a subsequent SYN retransmission, especially if it occurs during an inactivity period at the server. Note that with a 10 second burst and a 5 second inactivity period, the probability of SYN retransmissions occurring during an inactivity period is high.

Interestingly, as the burst length increases to 20 seconds, the difference in throughput between ***abort rst-syn*** and ***default*** becomes negligible. With a 40 second burst, the throughput of ***abort rst-syn*** is higher than that of ***default***, which is similar to what we observed under persistent overload. As the burst length is increased relative to the length of the inactivity period, most of the client retransmissions with ***default*** happen not during an inactivity period, but while the server is still overloaded, thereby lowering the server throughput. The response time obtained with ***abort rst-syn*** is two orders of magnitude lower than that obtained with ***default*** for all the three burst lengths. Note that the reduction in response time is not due to an increase in the error rate.

Using a 10 second inactivity period with the same three burst-lengths did not lead to a significant change in the throughput or the response time (for both ***abort rst-syn*** and ***default***) when compared with the results shown in Figure 5.4. This section demonstrates that there is a crossover point with respect to how long an overload burst lasts relative to the length of the inactivity period, after which connection establishment mechanisms such as ***abort rst-syn*** perform as well as or better than ***default*** in terms of server throughput. Note that the client response times are significantly lower when ***abort rst-syn*** is used instead of ***default***, irrespective of the length of the burst. The discussion in this section also applies to the ***no-syn-drop*** mechanism, which aborts client connection attempts at the ACK stage instead of the SYN stage as done in ***rst-syn***. Both these mechanisms can result in suboptimal throughput during short-lived bursty traffic when client applications (or end users) do not retry a connection immediately upon receiving an explicit failure notification from the server.

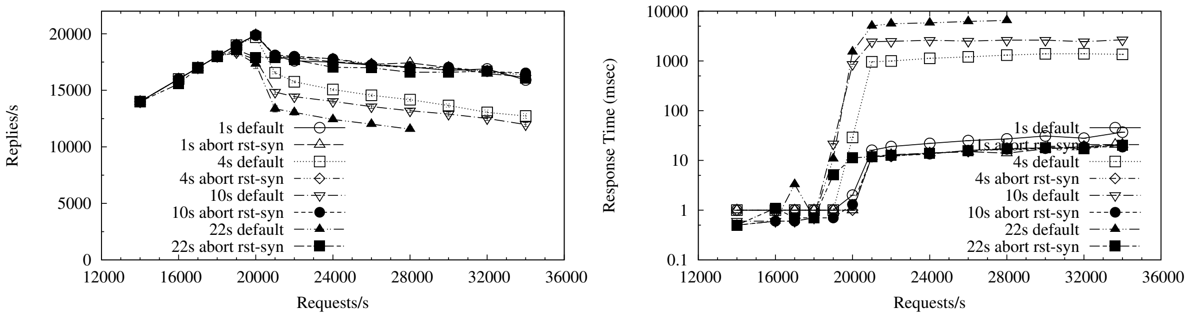
In this section, we used a simple model of bursty traffic along with a fixed-length inactivity period. In the future, we intend to perform a more rigorous analysis to determine exactly where this crossover point lies for variable-length, variable-rate periods of overload and light load. Dynamic techniques can then be used to *detect* persistent overload and notify some client TCP stacks to stop the retransmission of connection attempts. One such simple dynamic technique that addresses both bursty traffic and persistent overload is described below. The server TCP stack assumes short-lived bursty traffic when queue drops first occur. It checks periodically if queue drops continue to occur for t seconds. If queue drops do last for t seconds, persistent overload is detected and the TCP stack notifies clients to stop the retransmission of connection attempts. Although the server throughput with this mechanism will be slightly lower at the start of the persistent overload period, the server is not adversely affected by bursty traffic. The value of t can be determined analytically (based on the crossover point) or empirically using traces from flash crowds.

We believe that the automatic detection of persistent overload can form the basis of interesting research in the future. Note that the mechanisms discussed in this thesis do not have an impact on server behaviour under light loads. Mechanisms such as ***abort*** *rst-syn* or *no-syn-drop* provide a significant improvement in throughput during persistent overload. However, these mechanisms might result in suboptimal throughput during short-lived bursts of heavy traffic, techniques to automatically detect persistent overload can help eliminate this limitation. Both *abort rst-syn* and *no-syn-drop* provide more than an order of magnitude reduction in client response times during bursty traffic as well as under persistent overload. We believe that the results reported in this thesis make a strong case for deploying mechanisms that notify clients to stop the TCP-level retransmission of connection attempts in production TCP stacks.

5.3 Different Client Timeouts

For all of our experiments in this thesis we use a 10 second client timeout as explained in Section 3.2.3. That is, if the client application (`httperf`) does not receive a SYN/ACK or a response to its request from the server within 10 seconds, it gives up on its connection, marking it as an error. The client application typically times out because it does not receive a SYN/ACK in response to its SYN. The timeout in turn influences the total number of SYN transmissions by the client TCP stack. As shown in Figure 4.5, the Linux TCP stack retransmits SYN segments five times for up to 93 seconds after a client application issues a `connect()` system call (if no SYN/ACK is received for any of the SYN transmissions). A timeout of 10 seconds allows for at most three SYN transmissions, including retransmissions 3 and 9 seconds after the original transmission. This is similar to the behaviour of TCP stacks in Windows 2000 and XP.

In this section, we study the impact of using different timeout values on the throughput and the client response times when the server is under overload. We use the following timeout values for our experiments – 1 second, 4 seconds, 10 seconds, and 22 seconds. These values have been chosen to allow for one second after the transmission of the first, the second, the third, and the fourth SYN respectively (in Linux), after which the client gives up on the connection if it does not receive a server response. In Figure 5.5, we compare the throughput and the response time obtained with *abort rst-syn* to that obtained with *default* with these four timeout values. Note that using a 22 second timeout we were unable to generate more than 28,000 requests/sec with *default* using just 8 client



(a) Throughput (b) Response Time

Figure 5.5: Evaluation of the μserver with different client timeouts

machines because the clients ran out of ephemeral ports. For this reason, data points after 28,000 requests/sec are missing for the *default* results with the 22 second timeout in Figure 5.5.

Figure 5.5(a) demonstrates that as the timeout increases, the throughput provided by *default* decreases. In contrast, the throughput with *abort rst-syn* is not affected by the timeout value because no SYN retransmissions are permitted. The improvement in throughput after the peak with *abort rst-syn* in comparison to *default* is close to 20% with a 4 second timeout, more than 27% with a 10 second timeout, and more than 35% with a 22 second timeout. The throughput as well as the response time obtained with *default* is similar to that obtained with *abort rst-syn* with a 1 second timeout because there are no SYN retransmissions.

As shown in Figure 5.5(b), the response times obtained with *abort rst-syn* are two orders of magnitude lower than that obtained with *default* for timeout values over 1 second. This is because clients can get service on later SYN retransmissions with *default*. Note that we could increase the gap in the response times between *abort rst-syn* and *default* even further (i.e., to three orders of magnitude) simply by using a larger timeout (e.g., 46 seconds).

The results in this section demonstrate that as the timeout value used by client applications or users increases, the improvement in throughput and response times provided by connection establishment mechanisms such as *abort rst-syn* (which prevents the TCP-level retransmissions of connection attempts) increases as well.

5.4 Early SYN Drops

In this section we describe a mechanism that allows an early discard of client connection establishment packets under overload anywhere in the networking stack. This mechanism relies only on existing data structures available in the Linux TCP stack, in particular, it only uses the listen and SYN queues. Unlike previous work by Voigt et al. [85] or Iyer et al. [40], no external admission control technique is required. We refer to our mechanism as *early-syn-drop*.

Researchers have advocated dropping packets as early as possible in the networking stack in order to minimize the resources spent on dropped packets [63]. In this section, we demonstrate that dropping SYN segments earlier in the networking stack does not significantly reduce their negative impact on server throughput. That is, the high overhead of processing SYN segments during high loads is not an artifact of a poor implementation of TCP connection establishment in Linux. In the process, we point out that techniques such as persistent dropping [45] that seek to filter our retransmitted SYN segments with a lower overhead are unlikely to succeed in improving server throughput.

Figure 5.6 outlines the code path[4] taken by a TCP SYN segment in the NAPI-enabled Linux networking stack. The packet is first processed by the driver and then by a NAPI-based kernel thread when it is received. An `skbuff` encapsulating the packet is created, and it is passed on to the IP layer for processing. Note that Figure 5.6 shows the code path for IPv4, however, the code path for IPv6 is similar. A packet destined for the local machine is then passed on to the TCP layer for further processing. The `tcp_v4_rcv` function (line 13) serves as an entry point into the TCP layer from the IP layer, while `tcp_v4_conn_request` (line 25) implements the bulk of the SYN processing, including dropping SYN segments or responding with a SYN/ACK. As seen in Figure 5.6, multiple procedures (many of which are inlined hash table lookups) need to be invoked at the TCP layer before a SYN can be processed.

We now describe how *early-syn-drop* is implemented. A SYN segment is normally dropped whenever the SynQ3/4Full or ListenQFullAtSyn rules are triggered. However, as indicated in Section 4.1.3, when *abort* is used in our environment, SYN segments are

[4] In the figure, indentation distinguishes called functions from the callee, and all function at the same level of indentation in a particular component occur in sequence.

```
    [Device Driver / Software Interrupt]
(1) e1000_intr (interrupt handler)
(2)   netif_rx_schedule  (schedules software interrupt)
      [software interrupt scheduling code omitted]
(3)   net_rx_action      (run when software interrupt is scheduled)
(4)     e1000_clean (dev->poll)
(5)       e1000_clean_rx_irq
(6)       [NAPI accounting code omitted]
(7)         netif_receive_skb   (calls ip_rcv)
    [IP]
(8)  ip_rcv
(9)    nf_hook (netfilter hook, disabled in our configuration)
(10)   ip_rcv_finish
(11)     ip_local_deliver
(12)       ip_local_deliver_finish   (calls tcp_v4_rcv)
    [TCP]
(13) tcp_v4_rcv
(14)   __tcp_v4_lookup
(15)     __tcp_v4_lookup_established
(16)     __tcp_v4_lookup_listener
(17)   tcp_v4_do_rcv
(18)     tcp_v4_rcv_established (fast path for ESTABLISHED sockets)
(19)     tcp_v4_hnd_req (otherwise)
(20)       tcp_v4_search_req
(21)         tcp_v4_check_req (req found, try to transition from SYN_RECV to ESTAB.)
(22)           tcp_v4_syn_recv_sock (fails if listen queue is full)
(23)         __tcp_v4_lookup_established (if req not found)
(24)     tcp_rcv_state_process
(25)       tcp_v4_conn_request (for LISTEN sockets, process SYN, try to send SYN/ACK)
```

Figure 5.6: TCP connection establishment code path in Linux with NAPI

dropped only because the listen queue is full[5]. Hence, as long as the listen queue is full[6], SYN segments can be dropped earlier in the networking stack. Whenever a SYN or a SYN/ACK ACK is dropped due to ListenQFullAtSyn or ListenQOverflow, a flag is set. It is unset whenever space is created in the listen queue following an `accept()` call. This flag is visible across the various layers in the networking subsystem and when set, earlier methods can drop SYN segments without passing them on to subsequent functions (i.e., higher up the networking stack). The ***early-syn-drop*** mechanism thus provides basic admission control functionality that relies only on the listen queue data structure in the TCP stack. By enforcing packet drops only when queue drops occur, ***early-syn-drop*** ensures that the server is never under-utilized during overload.

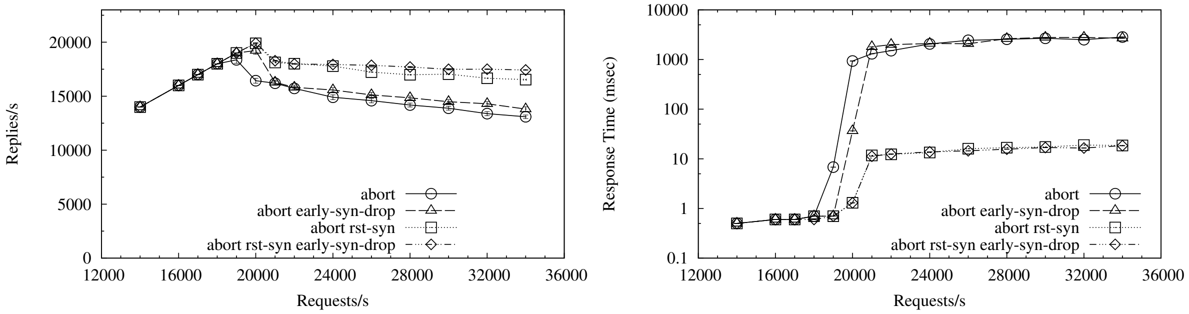
The flag indicating that SYNs can be dropped is visible throughout the networking subsystem allowing ***early-syn-drop*** to be enforced at various stages in the network packet processing. We have explored two such options – at the entry point of the TCP layer (i.e., in the `tcp_v4_rcv` function (line 13)), and at the entry point of the IP layer (i.e., in the `ip_rcv` function (line 8)). Dropping SYN segments at the entry point of the TCP layer obviates the need for hash table lookups when the listen queue is full. Similarly, dropping SYN segments at the entry point of the IP layer reduces the resources spent on dropped packets in further IP and TCP processing. Note that some packet filters and firewalls (e.g., `netfilter`) use hooks at the entry point of the IP layer to enforce dropping of packets. While we did not use a generic packet filter because of the high overhead involved, we added code at various stages in the networking stack to drop SYN segments by examining the flags field in the TCP headers. We did not observe a significant difference in server throughput (or response time) between the implementation of ***early-syn-drop*** at the entry point of the IP layer and that at the TCP layer. In this section, we will present the results of ***early-syn-drop*** implemented at the entry point of the TCP layer because this approach preserves protocol layering. Note that it might be possible to implement ***early-syn-drop*** in the driver allowing certain packets to be dropped even before they are pulled off the network card. To our knowledge, the e1000 driver used in our evaluation does not support selective packet drops on the network card. We intend to explore the

[5] Queue drops due to the SYN queue being full can also be avoided by using SYN cookies.

[6] It would be trivial to add a rule to enforce early drops when the SYN queue is full.

possibility of enforcing *early-syn-drop* directly at the network card in the future.

It is possible to use *early-syn-drop* in conjunction with other connection establishment mechanisms. In fact, we can use the *early-syn-drop* mechanism in combination with *default* or any of the connection establishment alternatives shown in Figure 4.1. To isolate the impact of *early-syn-drop*, we present the results of *abort* and *abort rst-syn* when used with and without *early-syn-drop* in Figure 5.7. While *abort early-syn-drop* implies a silent, but early drop of SYN segments, with *abort rst-syn early-syn-drop*, whenever SYN segments are dropped early, a RST is sent to the clients. Note that *abort rst-syn* (without any early SYN drops) serves as a yardstick to measure the effectiveness of *abort early-syn-drop*.



(a) Throughput

(b) Response Time

Figure 5.7: Evaluation of the μserver with early SYN drops

Figure 5.7(a) shows that *early-syn-drop* fails to yield a substantial improvement in throughput. The increase in throughput over *abort* with *abort early-syn-drop* is less than 5%, and nowhere as significant as the improvement provided by *abort rst-syn*. In fact, even *abort rst-syn early-syn-drop* fails to yield more than a 5% improvement in throughput over *abort rst-syn*. We treat all received SYN segments uniformly, irrespective of whether they are retransmitted or not. Hence, as shown in Figure 5.7(b), our implementation of *early-syn-drop* cannot reduce the client response time, which is influenced by retransmitted SYN segments.

Note that *early-syn-drop* is effective in dropping SYNs early when the listen queue is full. For example, at 30,000 requests/sec, with *abort early-syn-drop*, 4,979,497 out of the 7,610,976 SYN segments received are dropped, out of which 4,968,015 SYNs are dropped early, and only 11,482 SYNs are dropped through the normal SYN-processing code path[7].

Thus, the ***early-syn-drop*** mechanism is effective in ensuring that SYN segments are dropped early in the networking stack during server overload. However, an early drop fails to reduce the overhead of processing SYN segments. Consequently, it does not improve server throughput under overload compared to mechanisms that explicitly notify clients to stop the TCP-level retransmission of connection attempts. In light of the results in this section, we expect that mechanisms such as ***rst-syn*** and ***no-syn-drop*** will provide better server throughput during flash crowds compared to approaches like persistent dropping [45], which require SYN segments to be processed at the server before they are dropped. We intend to determine if this is the case in future work.

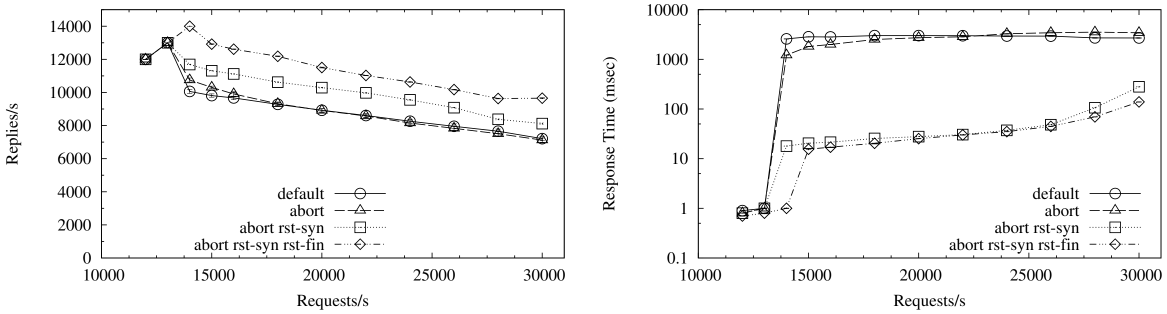
5.5 Different Hardware Platforms and TCP Stacks

We use a Xeon (x86) machine running a Linux 2.4.22 kernel as the server in all of our experiments. In this section, we revisit some of the TCP connection establishment and termination mechanisms described in Chapter 4, evaluating their performance on a Itanium-2 (ia64) server machine that runs a current Linux 2.6.11 kernel. We show that the results of the connection management mechanisms presented in this thesis are not sensitive to the hardware platform used for their evaluation.

The Itanium-2 is a 64-bit architecture that uses features such as an explicitly parallel instruction set that make it very different from the 32-bit x86 architecture. As described in Section 3.1.2, the Itanium-2 server used for the experiments in this section also has a larger cache and main memory compared to the x86 machine used in the rest of our experiments. Recall also that the implementation of connection management in the Linux 2.6 TCP stack (as of the current version, 2.6.11.6) has not changed from that in the 2.4 stack. In Figure 5.8, we present the throughput and response time results for ***default***, ***abort***, ***abort rst-syn***, and ***abort rst-syn rst-fin***.

The results shown in Figure 5.8 are qualitatively similar to those presented in Chapter 4.

[7] Drops of SYN segments through the normal code path must take place in order to trigger subsequent early SYN drops.



(a) Throughput

(b) Response Time

Figure 5.8: Evaluation of the μserver on an Itanium-2 server and a Linux 2.6 kernel

Using ***abort*** fails to provide a substantial improvement in the throughput or the response time over ***default*** due to the overhead of processing retransmitted SYN segments. With ***abort rst-syn***, there is close to a 15% improvement in throughput after the peak, and nearly two orders of magnitude reduction in client response times, when compared with ***abort*** and ***default***. Using ***rst-fin*** in conjunction with ***abort rst-syn*** results in 30% higher throughput after the peak compared to ***default***.

Note that in absolute terms, the results of *all* the connection management mechanisms (including ***default***) are worse on the Itanium-2 server than those on the Xeon server. In particular, the peak throughput is 38% lower. We believe that is because the design and implementation of efficient compilers and kernel subsystems on the Itanium architecture is still under active development. However, the results of the connection management mechanisms are qualitatively similar to those on the Xeon server used in the rest of our experiments. Thus, the results reported in this thesis are not an artifact of the server hardware platform or the version of the (Linux) TCP stack used in the experiments.

The results presented in this Chapter reaffirm the fact that the implementation choices for TCP connection management studied in this thesis can have a significant impact on server throughput and client response times in a variety of environments. We hope that our work can assist application developers, system administrators, and protocol implementers in choosing an appropriate mechanism to ensure good Internet server performance under overload.

Chapter 6

Conclusion

In this thesis, we provide a better understanding of the behaviour of an overloaded server. In particular, we describe some of the causes of the drop in server throughput and the increase in client response times 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 review the existing approaches for implementing connection establishment in TCP stacks in Linux, various flavours of UNIX, and Windows. We point out a shortcoming in the default Linux implementation of connection establishment (***default***). The silent drop of SYN/ACK ACK segments from clients when the listen queue overflows at an overloaded server can cause a disconnect between the TCP states at the clients and the server, resulting in unnecessary traffic at the server as well as in the network. TCP stacks in UNIX and Windows implement the ***abort*** mechanism to prevent this disconnect by sending a reset (RST) to clients whose SYN/ACK ACK segments are dropped. The Linux TCP stack should similarly enable ***abort*** by default. Although proactive mechanisms such as listen queue reservation (***reserv***) can be used avoid listen queue overflows, they are more vulnerable to SYN flood denial of service attacks in the current Internet infrastructure.

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 in a variety of operating environments. Unfortunately, a mechanism such as ***rst-syn***, which sends RST to clients whose SYN segments are dropped, cannot be effectively deployed because some client TCP stacks, notably those on Windows operating systems, do not follow RFC 793 and instead retry a SYN immediately upon receiving a RST from the server for a previous SYN. We describe a mechanism called ***no-syn-drop*** that prevents the retransmission of connection attempts even in Windows-like TCP stacks. In ***no-syn-drop***, no SYN segments are dropped, instead the server TCP stack uses ***abort*** to reset connections from clients whose SYN/ACK ACKs cannot be accommodated in the listen queue. While the ***no-syn-drop*** approach is ungainly, it requires no modifications to client-side TCP stacks and applications, or server-side applications.

We believe that the results reported in this thesis make a strong case for TCP stack implementors to develop a clean mechanism to allow an overloaded server to explicitly notify clients to stop the retransmission of SYN segments. A silent drop of SYN segments as suggested in RFC 3360 [34] is harmful to Internet server throughput and client response time during persistent overload conditions such as flash crowds. Such an overload notification mechanism might be ICMP-based [44], it could leverage the text-based extensions suggested in RFC 793 to describe the cause of a TCP RST, or it could use a specially crafted TCP segment (e.g., a segment with the SYN, ACK, and FIN flags set)[1]. The client TCP stack can propagate this notification to the application through an appropriate socket-level error message such as `ESERVEROVRLD`.

We demonstrate that supporting half-closed TCP connections can lead to an imprudent use of resources at an overloaded server when client applications terminate connections using the half-duplex semantics offered by the `close()` system call. We describe a mechanism called ***rst-fin*** that disables support for half-closed connections at the server TCP stack. 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 ***rst-fin*** improve server throughput by 10%-15% on most workloads. While typical web transactions (i.e., HTTP GET requests) can operate

[1] A specially crafted segment can allow a client which receives a RST to distinguish between server overload and the lack of listening socket at the requested port.

with the relaxed data-delivery guarantees entailed by an abortive connection release or ***rst-fin***, this trade-off between strict reliability and higher throughput might not be acceptable for all application protocols. Instead of clients terminating connections abortively, an application-specific socket option could be implemented to allow server applications to select the ***rst-fin*** approach based on application protocol semantics.

6.1 Future Work

In the future, we intend to conduct a detailed study of the impact of TCP connection establishment mechanisms on server throughput under short-lived bursty traffic. In particular, we plan to investigate exactly how long a burst needs to last before it can be treated as persistent overload and examine whether the SYN and listen queues can be sized to minimize queue drops during short-lived bursts. We also intend to explore techniques to distinguish bursty traffic from persistent overload conditions so that the server can notify clients to stop the retransmission of connection attempts only when those retransmissions are likely to occur while the server is still under overload.

Additionally, we plan to examine the impact of TCP connection management mechanisms on server throughput and client response times under wide-area network conditions, using tools such as NIST Net [20] to simulate wide-area traffic characteristics such as delays and packet loss. We also intend to reevaluate the performance of the connection management mechanisms discussed in this thesis with specialized TCP stacks, such as those offloaded on to the network card or those implemented on dedicated packet-processing CPU cores, when such stacks become widely available. Furthermore, it would be instructive to assess the impact of the mechanisms studied in this thesis on dynamic-content workloads.

Finally, the results reported in this thesis indicate that it is possible to sustain tens of millions of hits per hour using an out-of-the-box server in our environment. This suggests that a well-designed user-space web server with a modified TCP stack running Linux on a uniprocessor commodity Xeon machine can provide good performance under overload. We intend to evaluate whether these developments obviate the need for maintaining elaborate web server farms running specialized load balancers in order to host high-volume web sites.

Appendix A

Glossary

abort: Connection establishment mechanism that aborts a client connection with a RST when ListenQOverFlow is triggered upon receiving a SYN/ACK ACK. Implemented in TCP stacks in FreeBSD, HP-UX, Solaris and Windows. Implemented but not enabled by default in the Linux TCP stack.

abort early-syn-drop: Connection establishment mechanism that aborts a client connection with a RST whenever a SYN/ACK ACK is dropped and effects an early drop of SYNs while the server is overloaded. See also early-syn-drop.

abort rst-syn: Connection establishment mechanism that aborts a client connection with a RST whenever a SYN/ACK ACK or a SYN is dropped (due to ListenQOverflow, SynQ3/4Full, or ListenQFullAtSyn).

abort rst-syn early-syn-drop: Connection establishment mechanism that aborts a client connection with a RST whenever a SYN/ACK ACK or a SYN is dropped. Note that SYNs are dropped early while the server is overloaded. See also early-syn-drop.

abort rst-syn close-rst: Connection management mechanism that uses *`abort rst-syn`* for connection establishment with clients that terminate their connections using an abortive release (*`close-rst`*).

abort rst-syn rst-fin: Connection management mechanism that uses *`abort rst-syn`* for connection establishment in conjunction with *`rst-fin`* for connection termination (which disables support for half-closed TCP connections at the server).

abort rst-syn win-emu: Connection establishment mechanism that uses *`abort rst-syn`* when dealing with *`win-emu`* clients, which retry a SYN upon receiving a RST for a previous SYN.

ACK stage: The second step in the three-way TCP connection establishment handshake concerned with the processing of a SYN/ACK ACK. See also SYN stage.

client TCP stack: The client-side functionality provided by the TCP stack in an operating system.

close-rst: An abortive release of a connection, where the client application forces its TCP stack to terminate a connection by sending a RST. Implemented by Internet Explorer 5 and 6.

default: The default connection establishment mechanism in the Linux TCP stack (implemented in the 2.4 and the current 2.6 kernel, as of 2.6.11.6), which silently drops SYN segments (due to SynQ3/4Full or ListenQFullAtSyn) as well as SYN/ACK ACK segments (due to ListenQOverflow).

early-syn-drop: A mechanism that allows an early drop of SYN segments anywhere in the networking stack while the connection establishment queues are full due to server overload.

ListenQFullAtSyn: A rule dictating that a SYN segment be dropped because the listen queue is full. Enforced at the SYN stage. See also ListenQOverflow and SynQ3/4Full.

ListenQOverflow: A rule dictating that a SYN/ACK ACK segment be dropped because there is no space in the listen queue. Enforced at the ACK stage. See also ListenQFullAtSyn and SynQ3/4Full.

no-syn-drop Connection establishment mechanism that does not drop any SYN segments, but relies on ***abort*** to reject client connections on ListenQOverflow-induced SYN/ACK ACK drops.

queue drop: A SYN or SYN/ACK ACK segment that cannot be accommodated in the SYN or the listen queue and has to be dropped.

regular clients: TCP stacks which give up on a connection upon receiving a RST in response to a SYN and indicate an error to the application, thus following the recommendations of RFC 793. See also win-syn-retry.

reserv: Connection establishment mechanism that implements listen queue reservation at the SYN stage.

reserv rst-syn: Connection establishment mechanism that sends a RST in response to a SYN whenever a reservation cannot be made in the listen queue.

rst-fin: Connection termination mechanism that disables support for half-closed TCP connections at the server. All FIN segments are assumed to indicate that a client is not interested in further read or write activity on a connection, hence, the server sends a RST in response to a client's FIN.

rst-syn: Connection establishment mechanism that sends a RST to a client whenever a SYN segment is dropped. Implemented in some Windows TCP stacks.

server TCP stack: The server-side functionality provided by the TCP stack in an operating system.

SYN/ACK ACK: A TCP segment used by the client to acknowledge that it has received the SYN/ACK sent by the server.

SYN drop: A silent drop of a SYN segment without any client notification, as implemented in most UNIX TCP stacks.

SYN stage: The first step in the three-way TCP connection establishment handshake concerned with the processing of a SYN. See also ACK stage.

SynQ3/4Full: A rule dictating the dropping of a SYN segment because the SYN queue is three-quarters full. Enforced at the SYN stage. See also ListenQFullAtSyn and ListenQOverflow.

sysctl: An administrator-configurable parameter in Linux.

win-emu: Our emulation of the win-syn-retry behaviour in the Linux client TCP stack. See also win-syn-retry.

win-syn-retry: Behaviour of some client TCP stacks, notably those on Windows operating systems, which do not follow the recommendations of RFC 793 and retry a SYN upon receiving a RST for a previous SYN.

Bibliography

[1] *socket - Linux socket interface*, May 1999. Linux Programmer's Manual, Section 7.

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

[3] Akamai. http://www.akamai.com.

[4] B. Ang. An evaluation of an attempt at offloading TCP/IP protocol processing onto an i960RN-based iNIC. HP-Labs, HPL-2001-8. Technical report, January 2001.

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

[6] M. Arlitt and T. Jin. Workload characterization of the 1998 world cup web site. *IEEE Network*, 14(3):30–37, 2000.

[7] M. Arlitt and C. Williamson. Understanding web server configuration issues. *Software – Practice and Experience*, 34(2):163–186, 2004.

[8] M. Arlitt and C. Williamson. An analysis of TCP Reset behaviour on the Internet. *SIGCOMM Computer Communication Review*, 35(1):37–44, 2005.

[9] M. Aron and P. Druschel. TCP implementation enhancements for improving web server performance. Technical Report TR99-335, Rice University, June 1999.

[10] 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*, pages 252–262, 1998.

[11] G. Banga and P. Druschel. Measuring the capacity of a web server. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)*, Monterey CA, December 1997.

[12] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Operating Systems Design and Implementation*, pages 45–58, 1999.

[13] 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, FREENIX Track*, June 1999.

[14] D.J. Bernstein. *TCP SYN cookies.* `http://cr.yp.to/syncookies.html`.

[15] N. Bhatti and R. Friedrich. Web server support for tiered services. *IEEE Network*, 13(5):64–71, September 1999.

[16] R. Braden. RFC 1644 - T/TCP – TCP extensions for Transactions, functional specification. `http://www.ietf.org/rfc/rfc1122.txt`, 1989.

[17] R. Braden. RFC 1122: Requirements for Internet hosts – communication layers. `http://www.ietf.org/rfc/rfc1644.txt`, 1994.

[18] T. Brecht. Private communication, November 2003. brecht@cs.uwaterloo.ca.

[19] T. Brecht, D. Pariag, and L. Gammo. accept()able strategies for improving web server performance. In *Proceedings of the 2004 USENIX Annual Technical Conference, General Track*, June 2004.

[20] M. Carson and D.Santay. NIST Net - A Linux-based network emulation tool. *To appear in SIGCOMM Computer Communication Review.* Available at `http://-www-x.antd.nist.gov/nistnet/nistnet.pdf`.

[21] CERT Coordination Center. *Denial of Service Attacks.* `http://cert.org/-tech_tips/denial_of_service.html`.

[22] A. Chandra and D. Mosberger. Scalability of Linux event-dispatch mechanisms. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 231–244, 2001.

[23] L. Cherkasova and P. Phaal. Session based admission control: a mechanism for improving the performance of an overloaded web server. HP-Labs, HPL-98-119. Technical report, June 1998.

[24] D. Clark, V. Jacobson, J. Romkey, and H. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, pages 23–29, June 1989.

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

[26] Standard Performance Evaluation Corporation. *SPECweb99 Benchmark*. `http://-www.specbench.org/web99/`.

[27] Standard Performance Evaluation Corporation. *SPECweb99 Design Document*. `http://www.specbench.org/web99/docs/whitepaper.html`.

[28] M. Crovella, R. Frangioso, and M. Harchol-Balter. Connection scheduling in web servers. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[29] D. Libenzi. *epoll Web Page* . `http://lse.sourceforge.net/epoll/`.

[30] P. Druschel and G. Banga. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *OSDI '96: Proceedings of the 2nd USENIX symposium on Operating Systems Design and Implementation*, pages 261–275, 1996.

[31] T. Faber, J. Touch, and W. Yue. The TIME-WAIT state in TCP and its effect on busy servers. In *Proceedings of INFOCOM 1999*, pages 1573–1583, March 1999.

[32] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Trasfer Protocol – HTTP/1.1. `http://www.w3.org/-Protocols/rfc2616/rfc2616.html`, June 1999.

[33] V. Firoiu and M. Borden. A study of active queue management for congestion control. In *INFOCOM*, pages 1435–1444, 2000.

[34] S. Floyd. RFC 3360: Inappropriate TCP Resets considered harmful. http://-www.ietf.org/rfc/rfc3360.txt, 2002.

[35] M. Freedman, E. Freudenthal, and D. Mazires. Democratizing content publication with Coral. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*, San Francisco, California, March 2004.

[36] D. Freimuth, E. Hu, J. La Voie, R. Mraz, E. Nahum, P. Pradhan, and J. Tracey. Server network scalability and TCP offload. In *USENIX 2005 Annual Technical Conference, General Track*, pages 209–222, April 2005.

[37] L. Gammo, T. Brecht, A. Shukla, and D. Pariag. Comparing and evaluating epoll, select, and poll event mechanisms. In *Proceedings of the 6th Annual Linux Symposium*, July 2004.

[38] J. Hu and G. Sandoval. *Web acts as hub for info on attacks.* http://news.com.com/-2100-1023-272873.html.

[39] Red Hat Inc. *TUX 2.2 Reference Manual*, 2002. http://www.redhat.com/docs/-manuals/tux/TUX-2.2-Manual.

[40] R. Iyer, V. Tewari, and K. Kant. Overload control mechanisms for web servers. In *Proceedings of Workshop on Performance and QoS of Next Generation Networks*, 2000.

[41] J. Levon. *OProfile - A System Profiler for Linux*, 2004. http://oprofile.-sourceforge.net.

[42] Van Jacobson, Craig Leres, and Steven McCanne. *tcpdump manual.* Available at http://www.tcpdump.org/tcpdump_man.html.

[43] H. Jamjoom, P. Pillai, and K.G. Shin. Resynchronization and controllability of bursty service requests. *IEEE/ACM Transactions on Networking*, 12(4):582–594, 2004.

[44] H. Jamjoom and K.G. Shin. Reject message extension for ICMP. Internet Draft (Expired), IETF. http://www.eecs.umich.edu/~jamjoom/publications/draft--jamjoom-icmpreject-00.txt, 2002.

[45] H. Jamjoom and K.G. Shin. Persistent dropping: An efficient control of traffic aggregates. In *Proceedings of ACM SIGCOMM 2003*, Karlsruhe, Germany, August 2003.

[46] P. Joubert, R.B. King, R. Neves, M. Russinovich, and J.M. Tracey. High-performance memory-based web servers: Kernel and user-space performance. In *USENIX Annual Technical Conference, General Track*, pages 175–187, 2001.

[47] 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*, pages 252–262. IEEE, May 2002.

[48] M. Kaashoek, D. Engler, G. Ganger, and D. Wallach. Server operating systems. In *Proceedings of the 7th ACM SIGOPS European workshop*, pages 141–148, 1996.

[49] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-sale: Surviving organized DDoS attacks that mimic flash crowds. In *2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[50] J. Kay and J. Pasquale. Profiling and reducing processing overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, 4(6):817–828, 1996.

[51] K. Kong and D. Ghosal. Pseudo-serving: a user-responsible paradigm for Internet access. In *Proceedings of the Sixth International Conference on World Wide Web*, pages 1053–1064, 1997.

[52] HP Labs. *The μserver home page*. `http://hpl.hp.com/research/linux/userver`.

[53] J. Lemon. Kqueue: A generic and scalable event notification facility. In *BSDCON2000*, 2000.

[54] J. Lemon. Resisting SYN flood DoS attacks with a SYN cache. In *BSDCON2002*, 2002.

[55] C. Lever, M. Eriksen, and S. Molloy. An analysis of the TUX web server. Technical report, University of Michigan, CITI Technical Report 00-8, Nov. 2000.

[56] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *SIGCOMM Computer Communication Review*, 32(3):62–73, 2002.

[57] P. McKenney and K. Dove. Efficient demultiplexing of incoming TCP packets. In *SIGCOMM '92: Conference proceedings on Communications architectures and protocols*, pages 269–279, 1992.

[58] Microsoft. *Windows Sockets 2 API – setsockopt Function.* `http://msdn.microsoft.-com/library/winsock/winsock/wsaaccept_2.asp`.

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

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

[61] J. Mogul. The case for persistent-connection HTTP. *SIGCOMM Computer Communication Review*, 25(4):299–313, 1995.

[62] 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*, pages 25–30, May 2003.

[63] J. Mogul and K. Ramakrishnan. Eliminating receiver livelock in an interrupt-driven kernel. In *Proceedings of the USENIX Annual Technical Conference*, pages 99–111, San Diego, CA, 1996.

[64] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67, June 1998.

[65] E. Nahum. Deconstructing SPECweb99. In *the 7th International Workshop on Web Content Caching and Distribution (WCW)*, August 2001.

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

[67] M. Ostrowski. A mechanism for scalable event notification and delivery in Linux. Master's thesis, Department of Computer Science, University of Waterloo, November 2000.

[68] V.N. Padmanabhan and K. Sripanidkulchai. The case for cooperative networking. In *First International Workshop on Peer-to-Peer Systems (IPTPS '01)*, pages 178–190, 2002.

[69] V. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. *ACM Transactions on Computer Systems*, 18:37–66, 2000.

[70] V. Pai, P. Druschel, and Willy Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, 1999.

[71] D. Pariag. Private communication, November 2004. db2pariag@cs.uwaterloo.ca.

[72] D. Pariag. Using accept() strategies to improve server performance. Master's thesis, Department of Computer Science, University of Waterloo, November 2004.

[73] J. Postel. RFC 793: Transmission Control Protocol. `http://www.ietf.org/rfc/-rfc793.txt`, September 1981.

[74] Y. Ruan and V. Pai. The origins of network server latency and the myth of connection scheduling. In *ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS) – Extended Abstract. Full version available as Technical Report TR-694-04, Princeton University*, June 2004.

[75] J.H. Salim, R. Olsson, and A. Kuznetsov. Beyond Softnet. In *USENIX – 5th Annual Linux Showcase and Conference*, pages 165–172, November 2001.

[76] B. Schroeder and M. Harchol-Balter. Web servers under overload: How scheduling can help. In *18th International Teletraffic Congress*, 2003.

[77] A. Shukla, L. Li, A. Subramanian, P. Ward, and T. Brecht. Evaluating the performance of user-space and kernel-space web servers. In *Proceedings of IBM Center for Advanced Studies Conference (CASCON)*, pages 189–201, 2004.

[78] O. Spatscheck and L. Peterson. Defending against denial of service attacks in Scout. In *OSDI '99: Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 59–72, 1999.

[79] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS 2002)*, pages 203–213, March 2002.

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

[81] W.R. Stevens. *TCP/IP Illustrated, Volume 3.* Addison Wesley, 1996.

[82] M. Thadani and Y. Khalidi. An efficient zero-copy I/O framework for UNIX. SMLI TR95-39, Sun Microsystems Laboratories. Technical report, 1995.

[83] 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.

[84] A. van de Ven. *kHTTPd Linux HTTP accelerator*. `http://www.fenrus.demon.nl`.

[85] 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*, Boston, June 2001.

[86] R. von Behren, J. Condit, F. Zhou, G. Necula, and E. Brewer. Capriccio: Scalable threads for Internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[87] L.A. Wald and S. Schwarz. The 1999 Southern California seismic network bulletin. *Seismological Research Letters*, 71(4), July/August 2000.

[88] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *Proceedings of the Eighteenth Symposium on Operating Systems Principles*, October 2001.

[89] M. Welsh, D. Culler, and E. Brewer. Adaptive overload control for busy Internet servers. In *Proceedings of 4th Usenix Symposium on Internet Technologies and Systems (USITS)*, March 2003.

[90] V. Zandy and B. Miller. Reliable network connections. In *MobiCom '02: Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, pages 95–106, 2002.