NETWORK-ORIENTED CONTROLS OF INTERNET SERVICES

by

Hani T. Jamjoom

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2004

Doctoral Committee:
Professor Kang G. Shin, Chair
Professor H V Jagadish
Assistant Professor Mingyan Liu
Professor Brian Noble

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

To my parents

&

my wife

ACKNOWLEDGEMENTS

At the beginning of my Ph.D. program, I was expecting a smooth and short ride to completion. I can say now that it was far from smooth and definitely not short. It took me almost six years to complete this thesis. I found that much of the journey was not purely academic. It was rather self exploring of one's ideas, ambitions, and limitations. I can remember the numerous times when my research have reached a dead-end. Without the support from those around me, I would not have been able to complete my degree.

This journey would not be possible without the endless support from my parents. This thesis is dedicated to them: my mother Prof. Fatmah B. Jamjoom and my dad Prof. Talal M. Jamjoom. They have given me the courage to follow my dreams and were and continue to be the beacons that guide me whenever I feel lost. I also want to thank my wife Rasha Al-Yafi for enduring side-by-side the twists and turns of getting a Ph.D. I could not imagine finishing this degree without her encouragements and patience.

I want to thank Prof. Kang G. Shin's for cultivating and encouraging my inner-growth toward this degree. He gave me the freedom and all the resources to explore my ideas. I especially want to thank him for his fatherly advice about life beyond school work.

This thesis had evolved over the years and those around me have helped me shape and fine-tune my work. I would like to thank Padmanabhan Pillai for his collaborative work, his incredible insight, and countless help. I also want to thank John Reumann and Chun-Ting Chou for their collaborative work and deep discussions. I also want thank my two brothers Dr. Hytham Jamjoom and Hattan Jamjoom for their support and encouragements. Finally, I want to thank Amgad Zeitoun and Mohammed El-Gendy for their friendship, support, and honest discussions in just about everything.

TABLE OF CONTENTS

CHAPTERS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER 1

INTRODUCTION

Wide use and expansion of the Internet has led to the proliferation of diverse and oftentimes complex Internet services. These services, on the other hand, have created unprecedented demands on end-servers, each of which usually hosts multiple services like Web, e-mail, and database services. The increased demands by end-users often out pace the recent progress in enhancing server's processing, storage, and networking capacities. For many Internet services, peak demands—commonly referred to as flash crowds—can be 100 times greater than the average load. Moreover, the clients' expectations for shorter response times are increasing as time-critical applications, such as e-commerce and stock trading, are being offered over the Internet. For example, an 8-second delay is regarded as the limit before an e-commerce server loses over 50% of its clients [127]. This figure will only get worse. With the prevailing "best-effort" service model, over-design alone cannot realistically provide any performance guarantees for time-critical services. It is, particularly, not the appropriate design philosophy for dealing with demand surges, such as flash crowds. If servers are not correctly provisioned to deal with the incoming load, their Quality of Service (QoS) will seriously degrade. In some cases, many servers collapse under overload—an unacceptable option.

Supporting QoS in end-servers has been addressed extensively in the literature, e.g., in [6, 14, 64, 66, 105, 117]. QoS support is mainly implemented at the operating system (OS) or application layer, each of which emphasizes different aspects of providing QoS. At the OS layer, resource and service provisioning techniques are generally designed to securely divide server resources among competing services. Their goal is to provide a virtual computing environment where multiple services can run without interfering with each other. Since these techniques lack application-level information, they only operate on the service level, i.e., giving an entire service more preference over other services. Application-level QoS solutions have emerged as an alternative for providing

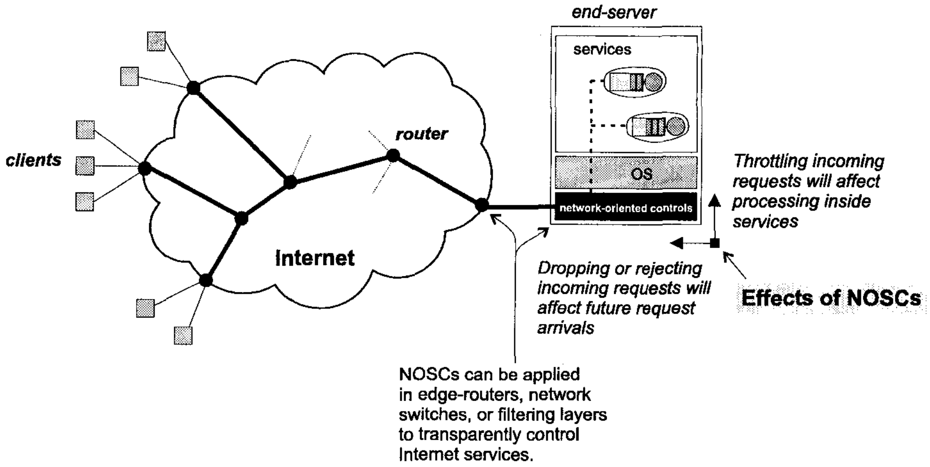

end-server
services
OS
network-oriented controls
clients
router
Internet
Throttling incoming requests will affect processing inside services
Dropping or rejecting incoming requests will affect future request arrivals
Effects of NOSCs
NOSCs can be applied in edge-routers, network switches, or filtering layers to transparently control Internet services.

Figure 1.1: Typical NOSC deployment

QoS differentiation without OS support [4, 6, 19, 44, 66, 78, 98, 119]. To further avoid modifying the service, a thin layer of control, referred to as the middleware layer, is inserted between the OS and the controlled services. Similar to OS techniques, middleware solutions try to distribute available resources to competing applications or services. Since the underlying OS enforces resource transparency (i.e., hides resource management), the middleware layer has limited abilities in providing strict service guarantees and is often restricted to only providing proportional QoS differentiation.

The combination of OS and application-level solutions has shown great promise in providing QoS support in end-servers. Unfortunately, the deployment of end-server-based QoS architectures in real systems have been seriously hampered by the large number of required changes to the OS and applications. However, since Internet services are dominated by short-lived connections arriving through the network, their internal operations—to some extent—follow the well-studied models of queueing networks. One may then seek a simpler, lighter-weight design that provides QoS guarantees and overload protection by transparently controlling arriving requests in a form of network-level admission control. Basically, arriving requests are policed by a controller that decides whether to accept or deny access to the arriving request based on the status (or health) of the controlled service. We use the term *network-oriented service controls* (NOSCs) to refer to the class of network-based approaches that provide QoS management for Internet services. Because NOSCs are applied at the network and transport layers, they require neither OS changes nor difficult offline capacity analysis. Furthermore, NOSCs can be easily integrated into network protocols, firewalls, or Layer-3+ load-balancers that connect servers to the Internet. NOSCs reduce the need for heavily over-designed servers since arriving request flows are matched to the server's capacity. This way, servers can be upgraded gradually as demand grows.

This dissertation undertakes a detailed investigation of network-oriented control mechanisms for providing QoS guarantees to short-lived Internet service requests. Because these control mechanisms are located on the path between the client and the server (Figure 1.1), the applied control will affect both the clients' future requests and the services' processing capacities. Central to our study, thus, is establishing a fundamental understanding of how arriving requests react to the applied control, and how applications react to the change in the server's load. With the combination of both views—the client's and the service's—this dissertation builds a complete picture detailing the performance, effectiveness, and limitations of using such mechanisms for controlling Internet services. With its general models, the results of this dissertation are applicable to a wide variety of Internet services. In addition, its structured approach in identifying and solving each issue is useful for other QoS models and control mechanisms. Our research methodology balances analytical modeling with empirical validation. We believe that analytical modeling provides greater insight into the problem at hand, while empirical validation provides evidences for real-world large-scale deployment.

The remainder of this chapter is organized as follows. The next section reviews the architecture of current NOSCs. Subsequently, we identify the main research problems facing NOSCs. This is followed by a detailed list of the primary contributions of the dissertation. We end this chapter by an overview of the organization of the dissertation.

1.1 Architecture of Network-Oriented Service Controls

Network-oriented controls have existed for some time. They were, however, not originally designed for controlling current Internet services. Classic network-based QoS-management approaches, such as WFQ [42] and CBQ [57], focus on fair allocation of network bandwidth for different flows. They are effective for long-lived connections, e.g., multimedia streaming, where the primary component of user-perceived latency is data transfer. Because most Internet services are dominated by short-lived connections, these mechanisms cannot be directly adopted for controlling such services. For example, in Hyper-Text Transfer Protocol (HTTP) based activities—which includes Web surfing, shopping, and banking—a separate TCP connection is commonly used to complete a single request. Each request corresponds to fetching a Web page or an embedded image, submitting a search request of credit card information, etc. In fact, the initial specification of HTTP (version 1.0) only supports this model of having a separate connection per request. Not until

HTTP 1.1 that the Persistent Connection option is introduced to multiplex multiple requests into a single connection, which should, in theory, improve the overall transfer throughput. To date, most Web browsers, still prefer issuing multiple (short) connections in parallel to maximize the client's throughput and minimize his perceived latency.

What has occurred with the advancement in Internet services and improvement in server and network capacity is, thus, a shift in the time and space granularity of requests: each request is now proportionally shorter in time and requires less network and server resources. But each server must be able to handle a much greater number of requests. Furthermore, requests are becoming more correlated, as future requests depend greatly on the successes of previous ones. A customer in an e-commerce site, for instance, will first browse for a product, add it to a shopping cart, and then check out. If NOSCs are to be effective in providing QoS support and overload protection, they must correctly deal with how requests are issued by clients and how servers process such requests. In particular, we identify three design objectives that have influenced the development of NOSCs:

- **QoS differentiation:** QoS differentiation can be enforced across two dimensions: service and client. For the former, a server that is hosting multiple services may need to differentiate among the levels of service that each is getting. In contrast, a server may also need to differentiate between different client groups. In both cases, the server must first classify incoming requests based on the destination service or the originating client group. QoS differentiation is then achieved by appropriately throttling incoming traffic (as a whole) to match the server's capacity while giving higher priority (or weight) to traffic flows belonging to different services or client groups—a similar idea to Rate-Controlled Static Priority [71] and DiffServ [22]. Since each service may react differently to the applied controls, NOSCs focus on determining *what* and *how much* to control to reach their desired goal.

- **Minimize client-perceived latency:** From a client's perspective, delay is the most important aspect in the performance of servers and networks. NOSCs, however, cannot reduce the processing delay of requests inside an Internet service (because they do not modify the actual service). To this end, NOSCs treat the server as a black box where it can bound the client's delay using two approaches. First, it can enhance the performance of existing controls before a request is admitted into service. This is akin to using Random Early Drop (RED) as a replacement for drop tail inside Internet routers to improve the controllability of long-lived TCP connections. Second, it can perform a form of low-level admission control, where new

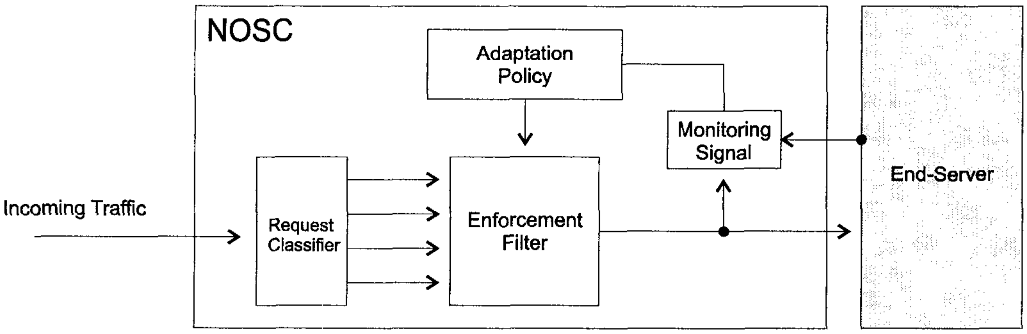


Figure 1.2: Architecture overview of NOSCs

(connection) requests are admitted only if they do not violate some performance criteria. This is similar to RSVP [124] and measurement-based admission control [69], but here server capacity is the resource under control instead of network capacity.

- **Improved reaction to overload:** Overload protection is one of the primary goals that motivate using NOSCs. Flash crowd events (FCEs) and distributed denial of service attacks (DDoS) can easily overwhelm even the most powerful servers. NOSCs are well-suited for providing this type of overload protection since FCEs and DDoS attacks can be both detected and mitigated at the network or transport layers. Therefore, appropriately-configured NOSCs can act as the first-line of defense against server overload.

The high-level architecture of most NOSCs adheres to traditional control models, which consists of a *controller* and an *input* to be controlled. In NOSCs, network traffic, which is represented by arriving requests, is the input. The controller refers to the software (or hardware) mechanisms that change the behavior of the input as it is passed to the output. A controller can be open or closed loop. In a closed-loop design, the controller computes the difference between a target and measured output value, and uses it to fine-tune its controls. Open-loop controllers do not feed back any information from the output. More specifically, the controller can consist of four basic components:

- **Packet classifier:** splits incoming requests into different classes based on client groups or destination services. Without this classification process, all incoming requests are treated equally. Therefore, the packet classifier is a crucial component for providing QoS differentiation, however, is not necessary for providing overload protection.

- **Enforcement filters:** specify how to control the incoming traffic or requests. They generally have an open-loop design. Examples of this component include random dropping and token buckets—both are used to shape the underlying traffic to meet a specific transmission rate. We refer to enforcement components as *filters* as they filter out nonconforming traffic.

- **Adaptation policies:** describe how these filters are manipulated based on specific input conditions. They provide the feedback loop for implementing more complex control algorithms. For instance, a rate-based dropping technique can increase the corresponding drop probability based on the load on the server to force more TCP connections to back off, thus reducing the arrival rate of incoming traffic.

- **Monitoring signal:** is a key element in providing the adaptation mechanism. Queue thresholds, such as overflow and underflow, are commonly used to trigger the adaptation policy to take corrective actions. Signals reflect the health of the system as well as the effectiveness of the underlying controls.

Different designs and implementations of these four components give rise to drastically different NOSCs. This dissertation looks at different design alternatives with the aim of finding those that are best suited—in terms of performance, robustness, and implementation—for controlling Internet services.

1.2 Dissertation Focus

The initial work on NOSCs, which we briefly survey in Chapter 2, shows promising results. However, several key problems must be addressed to reach the full potential of NOSCs. In general, what is required is a departure from the *ad hoc* approach of designing NOSCs to a more structured approach that provides a fundamental understanding between applied controls, the resulting reaction from the underlying traffic, and the change in client-perceived latency. This is a fundamental shift in the ideology of designing NOSCs from "what to hope will" to "what to expect to" happen when NOSCs are deployed in real systems.

Located between the client and the service—either inside a router or at the entry point of a server—the applied control has effects in two directions. In one direction, it affects the client's traffic as the controller decides what to let through and what to deny access. In the other direction, the applied control also affects the performance of the server as it dictates the maximum arrival rate (and possibly the distribution) of requests into service. Therefore, any study of NOSCs must be two sided: client-centric and server-centric. Combining both views would yield a concrete characterization of how NOSCs should be implemented, when they can be used, and what are their underlying limitations. We detail the exact problems below.

- *Gap between request arrival models and controls:* The majority of existing studies view request arrivals at an Internet service as a train of uncorrelated events from the perspective of the applied control. Such studies have looked at snapshots of flow aggregates (under specific load conditions) to characterize the various aspects of request arrivals (generally using an ON/OFF traffic model). While this is useful in many aspects of networking research, their models are too rigid to describe how request arrivals will change as different controls are applied to incoming traffic. Describing the arrivals with a Weibull distribution, for example, does not show how future arrivals will be affected by an increase in the number of dropped requests. Without this information, one cannot accurately predict the sensitivity of arriving requests to the applied controls. This may result in having a controller that oscillates between over- and under-enforcement. There is then a need for a request arrival model that incorporates this crucial feature. The new model must characterize the distribution of request arrivals and how it will change with different enforcement policies. This new understanding will not only improve the design of future control mechanisms, but will also allow more realistic testing before they are deployed in real systems.

- *Inadequacy of traditional enforcement mechanisms:* The way that traffic is controlled has remained relatively unchanged with buffering or dropping (either randomly or using token buckets) as the *defacto* mechanisms. Such mechanisms have proven their effectiveness at controlling long-lived TCP connections. However, by adopting them directly for controlling Internet services, they are assumed to also be effective for short-lived requests—an assumption that must be validated. What is needed is, thus, a rigorous investigation of the traditional controls that evaluates their effectiveness, pinpoints their shortcomings, and enhances their capabilities based on the new understanding of the intrinsic behavior of arriving requests. Such new mechanisms can provide improved performance for both the client and the server.

  New enforcement mechanisms are also needed. Application-level and OS-level QoS mechanisms have relied on admission control to allow only those requests that can be served while meeting their performance targets without causing the violation of targets for previously-admitted requests. Admission control tells the connecting clients that their requests will either be admitted and served in a timely manner or rejected immediately. The capability of explicitly rejecting requests is not fully available to NOSCs as they operate at the IP level. However, NOSCs can greatly improve both the controllability and client-perceived latency by having a similar mechanism. Especially with large and unexpected request bursts or flash crowds, explicit rejection can avoid repeated attempts by the clients to gain access to the corresponding service, which may only aggravate the overload situation.

- *Search for the optimality of applied controls:* The importance of determining the relationship of the applied control and request arrivals is outlined above. There is, nonetheless, a greater need for a deep understanding of how the control mechanisms (either the traditional or newly-proposed) will affect the client-perceived latency. This is especially necessary as each control policy has a wide range of possible parameter settings. A brute-force study of all possible control policies with all possible parameter selections is a daunting task. By creating powerful (analytic) models that are capable of expressing client-perceived delay as a function of the applied control, the search from the optimal control policy is greatly simplified. The result is the ability to choose the enforcement policy that minimizes the network delay while meeting the desired control targets.

- *No translation between network load and server load:* Classic network control mechanisms (like WFQ and CBQ) have focused on the fair partitioning of network resources among competing traffic. The translation between network load and resource requirement, therefore, is straightforward as the length of each packet is enough to calculate the required buffer size, consumed bandwidth, and transmission overhead. When NOSCs are applied to Internet services, this direct translation is no longer valid as single requests can produce huge processing delays inside an Internet service. This is particularly important for QoS-sensitive services, where NOSCs provide a form of low-level admission control. To appropriately control arriving requests, it is crucial to have enhanced understanding—either through empirical or analytical studies—of how these requests will be processed inside an Internet service. The goal here is, thus, to provide a mechanism that translates between arriving requests (or network load) to server load (or server resource requirements). This capability can be used to appropriately configure the NOSC policies, investigate the interactions between different services, and study the true extent for which NOSCs can be used to provide QoS guarantees/differentiation.

- *Rigid controller architecture:* Many of the existing NOSCs are designed with a specific deployment environment in mind. This rigidity hinders further deployment and adoptability of these techniques, as there is a wide range of services that may need specific controller customization. However, the basic components of any NOSC are relatively simple and can be designed in a modular way that allows a kind of plug-and-play approach for providing service-specific customization. Combining easy-customization with enhanced understanding and improved controls as outlined above will ensure large-scale and cost-effective deployment.

1.3 Primary Contributions

This dissertation makes several key contributions toward advancing the state of the art in using NOSCs. They are motivated by the issues outlined in Section 1.1 and are centered around the need for fundamental understanding of the relationship between the applied control, the reaction of the underlying traffic, and the resulting change in client-perceived latency.

- **Traffic analysis:** We rigorously analyze the effects of controls on incoming traffic to an Internet service. This is a crucial step for designing and evaluating any network control mechanism. Based on our empirical study, we introduce the *persistent client* model that captures how client requests do not simply go away when dropped. Instead, we show that there is a long-range dependency between the applied controls and future request arrivals. Our proposed model is a departure from traditional approaches where client models are based on observed traffic aggregates (mostly through traffic traces). In contrast, our model is created after careful analysis of a wide range of individual clients accessing different web services. Combining our observations of different clients, we build a coherent picture detailing the expected behavior of the aggregate behavior of arriving requests.

  With our new model, we unfold several unexpected aggregate behaviors in routers and end-servers during different load scenarios. In particular, we unveil a surprising discovery of re-synchronization of dropped requests. This has a strong implication on the control of bursty request arrivals. In particular, we show that traditional traffic policing mechanisms (e.g., Random Early Drop and Token Bucket Filters) often underestimate the effects of the enforcement policy on the underlying client-perceived latency.

  Because our work relies on actual implementation and real measurements, we have built

Eve, a scalable, extensible, and programmer-friendly tool that allows for fast generation of clients to test high-capacity Internet servers. Eve is used to simulate thousands, or tens of thousands, of persistent clients with very high accuracy. It is a crucial component in our evaluation as it integrates the proposed user-model with on-line measurements.

- **Optimal drop policy:** Based on the proposed client model, we develop an analytical model that relates the drop probability and the connection-establishment delay, which can easily dominate the total client-perceived latency when accessing an Internet service. Our model is constructed such that it can be applied to a wide range of drop policies. With this capability, we are able to explore the effects of different drop policies and propose *persistent dropping*, a simple and novel mechanism that we prove to minimize the client-perceived delay and the number of retransmitted requests while achieving the same control targets as traditional control mechanisms. In particular, persistent dropping achieves three important goals: (1) it allows routers and end-servers to quickly converge to their control targets without sacrificing fairness, (2) it minimizes the portion of client delay that is attributed to the applied controls, and (3) it is both easily implementable and computationally tractable. We also present two working implementations of persistent dropping that can be deployed in routers or end-servers.

- **Reject message extensions for TCP connections:** As mentioned earlier, admission control, from the point of the network stack, is a high-level operation that has been used extensively by QoS mechanisms. There are many situations where it is more advantageous to both the server and the client to perform this admission control before the connection is fully established. After surveying possible solutions, we show that there is no universal mechanism for providing admission control at the IP level. We, thus, propose a new IETF Internet standard for extending the Internet Control Message Protocol (ICMP) to provide explicit rejection capability. Our choice of adding a new ICMP message, as opposed to changing the TCP stack implementation, is motivated by the need for a simple implementation that can be incrementally deployed. We show that this new capability is an effective way for server and router to deal with clients' persistence. It also provides the client with immediate information, rather than relying on timeouts to infer the server's state.

- **Improved traffic prediction:** Because incoming requests to an Internet service can be inherently bursty, we introduce a new multi-stage filter that can perform accurate traffic prediction of future arrivals based on the applied control policy. Traffic prediction was based on our persistent client model and was augmented with a rejection mechanism to create the *Abacus Filter* (AF), a new traffic policing mechanism. AF complements our persistent dropping technique by better adapting to changing arrival patterns. The deriving principle is to maximize the number of admitted requests subject to the maximum acceptable network delay constraint. By modeling the time of the next TCP retransmissions, AFs estimate the amount of available future capacity and only admits portion of a burst that, with high probability, will eventually be admitted. This way, bursty traffic or a sudden jump in network load does not affect the overall delay of successful connections.

- **Analysis of impact of control on Internet services:** The above contributions focus on analyzing and improving request controls inside the network. We complete the picture by rigorously analyzing the effects of controls on the corresponding Internet services. Here we introduce the *multi-threaded round robin* (MTRR) server model to analyze the internal dynamics of a typical Internet service. This model is a generalization of existing time-sharing process models and is more representative of a wide range of current service implementations. For this model, we develop powerful, yet computationally-efficient, mathematical relationships that capture three important elements: (1) the effects of concurrency on the performance (in terms of throughput and response time) of a service, (2) the interactions between threads/processes of different services, and (3) the relationship between requests arrival and service performance.

  Our analytical model is general enough to describe the interactions between the applied controls and multiple services sharing the same machine. In particular, it is used to address important issues in the design of effective QoS mechanisms: (1) predict the true impact of the applied control—expressed by the maximum allowed request arrival rate—on the client-perceived latency, (2) estimate the corresponding performance of all running services, (3) describe the stability criterion of the underlying server, (4) find the rate limits that guarantee certain response times to different client groups (e.g., paid customers are given preferential treatment over the non-paying ones).

- **Integrated adaptation mechanism:** Combining the results from the above contributions, we design a general architecture for integrating signaling information with dynamic control. This mechanism, which we call *Adaptive Packet Filters* (APFs), requires minimal OS modifications to facilitate its deployment and maximize its portability. It uses a simple and robust technique that can provide immediate reaction to varying load conditions. An APF follows our design objectives of implementing its components as plug-in modules. This allows us to easily integrate our new control filters (i.e., persistent dropping and abacus filters). The current implementation targets Layer-4 network devices and shows full overload protection as well as service differentiation.

In conclusion, this dissertation shows that NOSCs are a viable component for providing QoS support for current and future Internet services.

1.4 Dissertation Roadmap

Chapter 2 begins with a broad overview of existing control mechanisms. It starts by describing controls in queueing networks. It then reviews current traffic and server models and describes the fundamental mechanisms that are used by servers to provide QoS guarantees—both at the OS level and at the application level.

Chapter 3 develops a detailed model of requests arriving at a typical Internet service. The model is based on an empirical analysis of Web browsers. It focuses on the interaction between incoming requests and the applied control. Here we introduce the notion of *persistent clients* and show how it affects the controllability of the underlying traffic. The chapter also describes an architecture for generating such clients. This model will be used in subsequent chapters to evaluate our proposed control techniques.

Chapter 4 introduces *persistent dropping*, a novel control mechanism for dealing with client persistence. The chapter also presents an analytical model of different dropping strategies and shows that persistent dropping is indeed the optimal drop strategy that minimizes the portion of client delay resulting from the applied controls.

Chapter 5 extends our persistent dropping model to handle the arrival of bursty service requests. With this, we introduce two new capabilities: (1) an explicit low-level mechanism for servers to reject incoming TCP connections, and (2) an accurate traffic predictor to maximize the throughput of the controlled server while keeping client delay in-check. Using the combination of both mechanisms, we develop the *Abacus Filter*, a multi-stage filtering scheme. The delay-control properties of various filtering mechanisms are also analyzed. AFs are shown to exhibit tight delay control and better complement traditional admission control policies.

Chapter 6 sets the foundation for understanding the limits of using network-oriented approaches for controlling Internet services. The chapter introduces a multi-threaded round-robin (MTRR) server to model the operation of a wide variety of Internet services. This model, in combination with an empirical analysis, is used to create a coherent picture of the underlying dynamics of a typical Internet service. Being equipped with this capability, we answer key questions in the design and performance of rate-based QoS differentiation.

Chapter 7 describes a general architecture for integrating signaling information with dynamic filter selection. It combines the findings from the previous chapters to introduce an effective network-oriented architecture for controlling server load and providing client-side QoS differentiation.

Chapter 8 concludes the dissertation by summarizing its contributions and suggesting possible topics for future investigations.

CHAPTER 2

RELATED WORK

Network traffic management techniques receive a fair amount of attention for their ability of controlling congestion while maximizing systems' throughput. The underlying idea behind most management solutions is to control the arrival of incoming traffic based mainly on measurement queue occupancy. Looking at the broader picture, ongoing research in providing overload protection and QoS support to Internet services can be grouped into three major categories: application- and middleware-level QoS support, OS-level resource reservation, or network-based QoS differentiation. In this chapter, we start by contrasting a few selected approaches with respect to adaptation, resource control, and system integration. Because much of our work is based on analytical models of arriving requests and service disciplines, Section 2.2 gives a general overview of techniques and models for characterizing incoming network traffic. We also review, in Section 2.3, several analytical techniques of queueing systems.

2.1 End-server QoS Models

Supporting QoS has been the focus of a wide range of research and commercial projects. They have advocated including QoS support in client machines, servers, network switches, and high-speed routers, in almost every network protocol, every OS, and endless applications. Common to all such efforts is a simple promise of increasing the utility of the end-user—here, both utility and end-users are loosely defined. In this subsection, we only survey popular end-sever QoS models with a particular focus on techniques that are useful in the development of our work.

2.1.1 Application-level QoS Support

Supporting QoS at the application level is generally preformed without OS's involvement. This technique is important for differentiating between client groups, e.g., paying and non-paying customers. It can also be used to prioritize certain important services over other less-important ones. Application-level QoS support is implemented by either modifying the actual application or adding a middleware layer that resides between the application and the OS. Because both approaches have limited control over system resources, they are almost always restricted to implementing one of the following two operations (or both) to provide QoS support:

- **Queueing-based Control:** By intercepting incoming requests from the OS, the middleware (and similarly the application) layer can slow down or police arriving requests based on the destination service or source client. By slowing down the arrival rate of one client group, for example, and leaving the other ones unaffected, that client group will receive a lower QoS.
- **Thread-based Allocation:** For multi-threaded services, the allocation and priority of threads can be manipulated by the middleware layer or by the service itself to increase its share of the system capacity. Services with more threads will receive a larger portion of the system's resources, which, in theory, implies that such a service have higher QoS than those allocated smaller numbers of threads. We review some of the analysis techniques of threading models in Section 2.3.

Queueing-based controls include many of the popular admission control schemes [20, 33–35, 40, 113]. Web2k [20], for instance, modifies a web server to include a two-stage mechanism. The first stage accepts and classifies incoming requests into different queues, with each queue representing a different service-level priority. The second stage then decides which request to admit into service to achieve the target QoS. A similar idea is implemented in [113], where they use a feedback control mechanism is used to monitor the response time of admitted requests, compare it to the expected delay by a $G/G/1$ queueing model, and adjust the acceptance rate using a Proportional Integral (PI) controller.

In contrast, several studies [1, 4, 44, 77, 83, 84, 98, 119] have used thread allocation to provide application-level QoS. Vasiliou *et. al* [119] focused their thread-based approach on providing a simple method for creating new scheduling disciplines. Similarly, Pandey [98] defined an object-oriented language to specify resource requirements for different client requests. There, the capacity of the server is divided into multiple channels based on the sending rate of the server, where each channel can represent a separate worker thread. Each request is then allocated a variable number of channels based on its resource specification.

Almeida *et. al* [4] have examined the difference between application- and kernel-level thread-based approaches as well as work-conserving vs. non-working conserving techniques. They observed that application-level approaches provide marginal improvements to the more preferred service class at the expense of large performance impact on the less=preferred service class. They also found that non-work-conserving techniques provided better QoS differentiation, but with lower system utilization.

Application-level QoS controls have two main advantages over OS or network-based mechanisms:

- More detailed information is available to the QoS mechanism. For instance, session-based admission control that is proposed in [35] monitors customer sessions (which consists of multiple requests) to a web server and adjusts its admission control policy to maximize the number of successful sessions.

- More adaptation options are available as applications are the ones doing the control. For example, in [1], a control-theoretic approach is used to implement a content adaptation mechanism that modifies the size of the content based on the load conditions of the server. Basically, as the server becomes overloaded, the QoS controller reduces the overhead per request by possibly using lower-resolution images or removing dynamic content. Another approach is proposed by Welsh *et. al* [120], where the application is modeled as a series of separate stages. Each stage represents a different processing task. A control approach is then proposed to adjust the allocated thread pool for each stage until an optimal request flow through the system is reached.

On the flip-side, application-level solutions suffer from several disadvantages. Most importantly, they require modifying the actual application, which is often a very involved and costly task. While middleware solutions eliminate this problem, they often require applications bind to them. For example, IBM's Workload Manager (WLM) requires compile-time binding to its queueing abstraction [6]. In contrast, the binding between WebQoS and managed applications happens at runtime since WebQoS intercepts the application's socket calls in its `-lsocket` substitute [66].

Both solutions fail if managed applications do not cooperate with the middleware (e.g., by managing their own I/O, multi-threading, and directly talking to the OS).

2.1.2 Kernel-level QoS Support

OS-based resource control mechanisms provide firmer QoS guarantees than their application-level counterparts since resource limits are enforced as soon as requests enter the OS. Kernel-level QoS support dates back to large mainframes where *virtualization* was a necessary part of dividing the large capacity of a mainframe. With today's cheap off-the-self machines becoming increasingly powerful, several solutions emerged to incorporate QoS support into modern OSs with the goal of providing performance isolation and QoS guarantees to Internet services that are hosted on a single machine [12, 14, 26, 105, 117]. The fundamental idea behind any kernel-level mechanism is to improve its resource accounting capability. This way resource usage (whether it is CPU, memory, disk, or network bandwidth) is accurately charged to each running application or process. Then, buy allocating each application a fixed resource quota, it can be guaranteed a minimum amount of resources regardless of the state of the other running applications.

The Scout OS [117] provides a novel *path* abstraction, which allows each OS activity to be charged to the path that triggered it. Therefore, when network packets are received, their path affiliation is recognized by the OS. Subsequent processing is charged to its path. Resource Containers and Lazy Receiver Processing (LRP) [12, 14] provide resource guarantees similar to Scout using the Resource Container (RC) abstraction. Ideally, all services bind to RCs, in which case the LRP mechanism of charging processing time to the destination service's RC provides full performance control. Signalled Receiver Processing (SRP) achieves a similar goal with less OS changes [27] by moving all packet processing into the application's process context. SRP, however, does not provide configurable QoS differentiation.

While providing excellent resource isolation, the mechanisms mentioned above fail to address the issue of adapting resource reservations to match the server's capacity. If administrators or the application developers misjudge server capacity, the above resource controls will fail to insulate competing applications. Another drawback of resource-reservation-based solutions is that it is unclear how these solutions would be deployed to achieve QoS differentiation in server farms. Cluster Reserves (an RC extension) attempt to address this problem [8]. Nevertheless, the proposed approach still suffers from the drawbacks of having to map QoS differentiation requirements to explicit resource reservations.

2.1.3 Network-Level QoS Support

Some of the fundamental ideas behind general network-level QoS support (including NOCSs) can be traced back to optimal control in queueing theory (discussed later). This general notion was further explored in the design of network routers [9, 57, 71, 123] with the emphasis on achieving fair-share allocation of resources across competing traffic sources as well as providing resource provisioning when performance guarantees are required. Logically, network-level QoS support consists of four fundamental functions: specification, admission control, classification, and enforcement. The first step for any application requiring its packets to receive certain QoS is to specify its requirements. If the network can support or guarantees the application's requirements, then the network traffic of that application is allowed to enter into the network. This can be seen in protocols like RSVP [124]. A network router must then be able to classify incoming packets into different traffic classes. A class can represent a single application (or single flow) as in IntServ [25]; it can also represent a group of flows (or aggregate flows) as in DiffServ [22]. Finally, the router must also implement an enforcement policy that guarantees that each traffic class adheres to its specification.

Traffic enforcement techniques, which are also known as Active Queue Management (AQM), have gained considerable attention in the literature. These techniques are important not only in providing QoS support but also because they improve the flow of traffic and guard against congestion under the existing "best-effort" design of the Internet. The two popular approaches to regulate network traffic are token buckets and random dropping. Token buckets [71, 110] specify two parameters: a bucket size and refill rate. The basic idea is that each packet consumes one token every time the packet is allowed to pass through (i.e., not dropped). This allows an average acceptance rate that is equivalent to the bucket's refill rate. It also allows for occasional bursts of up to the bucket's size. Therefore, if the average arrival rate is too high or the burst length is too long, the bucket will run out of tokens and incoming packets are simply dropped. Alternatively, random dropping techniques such as Random Early Drop (RED) [56], drops incoming packets with a specified probability. This probability is adjusted based on dynamic measurements of the arrival rate or buffer occupancy. Because aggressive flows will dominate packet arrivals, random dropping is found to enforce better fairness among competing flows.

The two mechanisms—token buckets and random dropping—are implicit in the fact that the sending application is indirectly notified of the drop using TCP's congestion control mechanisms—namely, receiving duplicate acknowledgments or timing out. Explicit mechanisms are also developed and are thought to improve the controllability of underlying traffic. Specifically, Explicit Congestion Notification (ECN) [53] aims to improve the performance of the underlying network by encoding the congestion state of routers into a single bit. The basic operation of ECN requires the cooperation of routers and end-hosts. When a router experiences a congestion build-up, it marks the congestion bit in the packets that are ECN-enabled (via the ECN-Capable Transport bit). The receiving host will then send its TCP acknowledgment with a ECN-Echo bit set. This tells the sender to reduce the corresponding TCP window. In general, ECN does not improve the performance of TCP. It, however, reduces the number of unsuccessful transmissions and also increases the fairness among competing TCP flows.

The results from network-resource management studies cannot be used directly to build effective NOSCs for two reasons:

- Network devices (which these algorithms are developed for) have well-defined computing environments with predictable overheads. Network processors, for instance, are generally non-preemptive where packets are processed, one at a time, until completion, and work can be expressed as a function of the packet lengths. On the other hand, server environments, which host the services under control, often concurrently operate on multiple requests and there is little correlation between incoming packet lengths and the amount of work that is requested.

- Network devices have a predetermined resource architecture where a fixed set of parallel queues buffer incoming packets, and once packets are processed, they are forwarded to one of a fixed set of output queues. In contrast, servers employ much more complex resource management causing unpredictable interactions between different applications.

Early network-oriented controls of Internet services received a fair amount of commercial attention since they are completely transparent to both applications and server OSs. The main use of network-oriented performance management for servers today is adaptive load-balancing [36, 48]. Frontends route incoming requests based on load measurements taken on the backends. Incoming requests are dispatched to the least-loaded backend. In addition to load-balancing, these solutions allow static rate definitions for different traffic classes. Extreme's *ExtremeWare* [48] allows the definition of rate limits for different flows (based on packet source and/or destination). Similarly, Cisco's *LocalDirector* [36] gives Application Service Providers (ASPs) control over the rate at which requests are forwarded to different machines. Both *ExtremeWare* and *LocalDirector* have a rigid design in the sense that they do not allow administrators to integrate arbitrary monitoring inputs with arbitrary packet-filtering policies. Instead, they provide pre-configured filtering and routing (e.g., load-balancing) solutions.

2.2 Traffic Characterization and Modeling

Studying network traffic is important for the design and implementation of effective traffic control mechanisms, network engineering, protocol enhancements, and application design. The field of network measurements is very broad and includes studies that focus on timing, protocols, multimedia, QoS support, etc. In this section, we only focus on those aspects that are important to the design of NOSCs. Namely, we first look at the burstiness of network traffic, its causes and potential hazards. We then review existing modeling techniques of TCP. Subsequently, we examine two potential causes of overload: Denial of Service (DoS) attacks and Flash Crowd Events (FCEs). Finally, we compare different techniques of generating representative workloads.

2.2.1 Self-similarity of network traffic

Self-similarity describes a phenomena of having structural similarities across multiple time scales. Compared with a Poisson-distributed traffic arrivals, for instance, self-similar traffic remains bursty even when the resolution (or granularity) of view is decreased (i.e., the level of aggregation is increased). Poisson arrivals, in such case, will smooth out to appear as white noise. More precisely, a stochastic process $X(t)$ is said to be *self-similar* when it is invariant in distribution under suitable scaling of time and space, or for any non-negative a, then $\{X(at)\} = \{bX(t)\}$, where b is also non-negative [46]. Other definitions of self-similarity have looked at the change in the autocorrelation function as a function of time scale [99]. Nonetheless, these definitions are consistent with the one above.

Two important characteristics are commonly attributed to self-similar processes. The first is being heavy-tailed. This is a consequence of long-range dependence, which reflects an autocorrelation function that decays hyperbolically, implying that very few and large observations have non-negligible probability of occurrence. The second is a slow decaying variance, which as we discuss shortly, has important implications on the measurement and simulation of the underlying process.

There are several causes for self-similar traffic behavior. What is important, though, is that each one can contribute to self-similarity at different time scales. At the smallest level, congestion control in TCP have been shown to contribute to this behavior [62]. Asymptotic self-similarity or large-time scaling, however, occurs due to the heavy-tailed ON/OFF source behavior [104]. Basically, source packets are sent using an alternating ON period, containing a train of packets, followed by an OFF period. Both ON and OFF periods are heavy-tailed and can be attributed to typical user access behavior to a Website: first downloading the main object and its embedded objects, then pausing for a while to digest the incoming information (also referred to as *think time*) [16].

Self-similarity then exists at different levels of the network protocol stack and also at different locations in a network. In particular, self-similarity has been observed in WAN traffic and in LAN traffic [50,52,80,99]. It has been also observed at the TCP connection arrival level, regardless of whether the application is Telnet, FTP, or Web [41,49,102,111]. Furthermore, it has been observed in high-level application functions such as session lengths of E-business workloads [87].

The importance of self-similar characterization of network traffic is far reaching. It affects how networks should be designed, how queues should be managed, and how measurements and simulations should be done. The most critical feature of self-similar traffic is having a heavy tail. With this, the length of a busy period or packet train can be arbitrarily long [51,80]. There is no natural length of a burst, and thus, packet loss decreases very slowly with increasing buffer length [82]. Non-bursty models do not show that, and thus, underestimates the necessary buffer length. This also decreases the effectiveness of conventional congestion control models.

Self-similarity, due to the slow decaying variance, also changes how statistical tests and confidence intervals should be computed [18]. In fact, traditional tests are inadequate since their computation of standard deviation are wrong by a factor that tends to infinity as the sample size increases. Computing the variance under self-similarity requires much larger sample sizes. This implies that when data samples are collected in a measurement environment, these measurements will slowly convergence to the average-case [39]. This is because large observations have a dominating effect even though they are also extremely infrequent.

There are several methods for detecting self-similarity in network measurements. The simplest one constructs a variance-time plot, where the variance of the process $X(t)$ is plotted over different time scales. To compute the process at time scale m (with $m = 1$ being the smallest value), we simply let $X^{(m)} = \sum_{i=1}^{m} X_{tm-m+i}/m$. This construction is then used to plot the variance of $X^{(m)}(t)$ as a function of m with the y-axis drawn in log scale. If the process under investigation is self-similar, the variance will decay at a linear rate as explained earlier. The slop of the resulting line is often referred to as β and relates to the Hurst parameter as $H = 1 - \beta/2$ [99]. Other similar techniques for estimating the Hurst parameter do exist (e.g., R/S analysis). In general, they share the same principle of identifying (directly or indirectly) the rate that the variance decays.

The two most common self-similar processes are fractional Gaussian noise and fractional ARIMA processes [101]. These can be synthesized using several techniques: (1) multiplexing a large number of alternating renewal processes with heavy-tailed ON/OFF periods, (2) counting the number of customers in an $M/G/\infty$ system with a heavy-tailed service distribution, (3) using the Random Midpoint Displacement method [96], (4) taking the inverse transformation of the wavelet coefficients corresponding to a wavelet transform of a fractional Brownian motion, (5) generating a sample path of a fractional ARIMA process [60], and (6) taking the inverse Discrete Time Fourier Transforms of a complex sample path of a FGN power spectrum [101].

The above techniques are useful at identifying and generating the general asymptotic self-similar traffic. However, such characterization, which is attributed to session and user behaviors, only explains the underlying traffic over large time scales. Protocol-specific mechanisms contribute to structurally different self-similar behaviors at much smaller time-scales. These are often called *multifractal* and simply imply that the scaling exponent (i.e., Hurst parameter) changes with time. With this, a new type of analysis, *multifractal analysis (MFA)*, was introduced in [2, 106]. The strength of this mechanism is that it can better detect scaling behavior of network traffic, and when used to generate synthetic load, it seems to also better approximate real traffic.

Despite the wide acceptance of self-similarity, some studies have suggested that this phenomenon does not apply to all network traffic. In particular, Morris and Lin [93] have demonstrated, through a limited Web trace, that the variance of the examined traces decays as fast as a Poisson process. Furthermore, Cao *et. al* [28] have demonstrated that large aggregates of Internet traffic, typical of backbone traffic, tend to behave as though it were Poisson-distributed. Collectively, these studies show that there is no single model that can be universally applied to explain the dynamics of network traffic. In effect, each individual trace is only a snapshot that reflects a specific network condition, so it may exhibit drastically different characteristics from other traces. Furthermore, as Internet services and client applications continue to evolve, so will the underlying traffic.

2.2.2 Analytic Modeling of TCP

Most of measurement studies focused on the characterization of TCP traffic. Several studies, however, attempted to establish an analytical framework for studying TCP [29, 54, 67, 88, 97, 98, 109]. In almost all cases, the analyzed models are simplifications of the actual inter-workings of TCP. However, such models capture the core behavior of TCP and aim to provide better understanding of TCP's internal dynamics. To this end, using an analytical approach, as oppose to a purely empirical one, has several advantages. By expressing the relationship between drop probability and TCP throughput, for instance, a clear link between cause and effect is established. This is particularly important for control mechanisms such as RED [56], where the drop probability is actively enforced. Having an analytical model can, then, be used to predict the resulting delays. Analytical modeling can also be used to provide a basis for comparison between different congestion control implementations. This, of course, is predicated on the ability of creating accurate models describing the behavior of the mechanisms under investigation.

The basic goal behind such studies is to express the throughput of TCP's congestion control as a function of the round-trip time (RTT) and the drop probability. Both of [29, 97] are based on the evolution of the TCP window during the various scenarios of congestion/flow control: slow start, fast transmit, fast recovery, receiving duplicate ACKs, receiving triplicate ACKs, losing a SYN packet, etc. Depending on where the drop occurs and which can be probabilistically applied to each scenario, an estimate of the throughput is computed. Alternatively, both of [88, 114] use differential equations to derive a similar model. However, they use different assumptions of the TCP implementation (assumed to generally follow TCP Reno) and account for different congestion scenarios. Overall, the derived equations only reflect the steady-state or long-term throughput of TCP under specific drop and RTT values.

2.2.3 Flash Crowd Events and Denial-of-Service Attacks

The measurements, models, and control mechanisms that were discussed so far pertain to well-behaved TCP flows. By well-behaved, we mean that they originate by a legitimate non-malicious access, and regular congestion control can regulate the flow of excess packets into the network. There are, however, certain phenomena where traditional controls are not effective; consequently, the measurements and proposed models do not apply to them. Two commonly-cited phenomena are *Flash Crowds Events (FCE)* and *Denial of Service (DoS)* attacks.

Flash Crowds Events: When a large number of legitimate users suddenly become interested in the same information, such as the *slashdot* effect or Victoria Secret's webcast [23], it causes a huge surge of requests to the server hosting such information [70]. This surge quickly exhausts all available resources, thus denying service to many users. There are two potential problems with

FCEs. First, when servers become overloaded due to the sudden increase in demand, they cannot sustain their maximum throughput. If not properly provisioned, they may totally collapse, denying service to all users. Second, it is often the case that a user accesses a server multiple times in a single session. Then, with the high arrival rate during an FCE, the success probability for a single connection is low enough that no user will be able to complete an entire session.

Denial of Service: There are several forms for this type of attack; they are generally characterized by the exploitation of weaknesses in a networking protocol to starve legitimate users from accessing a specific network or server resource. In its simplest form, the attacker floods the network with bogus packets such that all of the bandwidth is consumed. Variation of this attack sends actual request packets to exhaust server resources. The send rate does not have to be high. In fact, a few spoofed SYN packets are enough to totally block systems with small SYN backlog queues. Since servers are becoming more powerful and include certain measures to protect them from simple DoS attacks, new and more intelligent attacks are continually emerging. For instance, when multiple sources are used, their combined power can easily overwhelm even the most powerful servers [45, 59]. This is referred to as a Distributed Denial of Service (DDoS) attack and is responsible for millions of dollars of lost revenue to service providers.

Detecting and recovering from an FCE or a DoS attack is not easy, mainly because of the lack of cooperation by the sending sources. This is obvious for the DoS case. But for FCEs, the lack of cooperation is not from exiting users, as they exponentially back off every time a SYN packet is dropped, but from new ones that are not aware of the severity of the problem. Several solutions have emerged to deal with these growing threats in both the commercial and academic communities. Companies like Arbor Networks, Asta Technology Group, and Mazu Networks aim at detecting and preventing such scenarios by fingerprinting each attack. This builds a profile of possible scenarios similar to virus detection techniques. Traffic flowing through Internet Service Providers or servers are then monitored and checked against the database of fingerprints. Once an anomaly is detected, corrective actions are taken by isolating the sources that are creating the anomalies.

On the academic forefront, many different mechanisms have been proposed to detect and protect from these scenarios [12, 33, 85, 91]. The idea is similar to those proposed by commercial products of studying previous attacks, but generally focus on a specific type of attacks. In [33], for instance, FCEs are detected by live measurements of applications' change in response rate. Admission control is enforced when response rate drops beyond a certain limit. To improve the effectiveness of the protection mechanisms, Mahajan *et. al* [85] proposed aggregate-based congestion control (ACC) where the applied controls (namely dropping malicious packets) are pushed back through the network routers to be as close to their source as possible. This will obviously save network bandwidth and router resources.

When one service becomes subjected to a DoS attack or an FCE, one danger is that the attacked service consumes all available server resources such that other services running on the same host are also affected. Here, resource isolation mechanisms, such as the one described in Section 2.1.2, are very useful. In fact, several projects have emerged to address this exact problem. Banga and Druschel have proposed Lazy Receiver Processing (LRP) that performs accurate accounting of resource usage during kernel processing of network packets. The corresponding applications are then appropriately charged and are denied service once they exhaust their quota. To a lesser extent, a similar idea was proposed by Mogul [91]. There, the problem of Receive Livelock is addressed where the large number of interrupts (and the corresponding interrupt processing) caused by high packet arrival rates may consume all of the server's time, which effectively drives the server to a complete halt.

2.2.4 Load generations

Generating representative workloads is one of the most important parts of evaluating any QoS technique. Here commercial tools, like LoadRunner by Mercury Interactive, provide the most functionality when compared to research or open source projects. However, because of their proprietary architecture, little is known about their internal mechanisms. We, thus, limit our discussion to three well-known tools: SPECWeb99 [37], SURGE [16], and httperf [94].

When creating a client emulation tool or software, the following four objectives are considered: accuracy, efficiency, extensibility, and scalability. Simply, one would like a tool to accurately represent a specified client model. It should efficiently minimize the amount of resource needed to execute such model. It should scale beyond the boundaries of a single machine. Finally, it should be easily extended to create new and sophisticated client models and to allow a large degree of customization. Efficiency and scalability are necessary for maximizing the number of clients that can concurrently being emulated. Accuracy and extensibility are necessary for usability.

We contrast the different tools with respect to the above four objectives:

- **Extensibility:** SPECWeb99 [37] and SURGE [16] follow a black-box architecture and are created as monolithic applications. Any modification may require intimate knowledge of the tool's internal behavior. In contrast, httperf [94] provides hooks for extension, but it tends to be limited to certain functionality.

- **Efficiency:** httperf implements its clients using a single process in conjunction with non-blocking I/O to sustain the offered rate. Each I/O operation is considered a separate event, e.g., creating a new connection, sending data, or timing out an existing connection. The single-process event-driven approach has two disadvantages. First, it does not scale easily across multiple machines and across heterogeneous client population. Second, it is error-prone since programming non-blocking I/O can be tricky. Alternatively, SPECWeb99 and SURGE use threads (or processes) to simplify their programming. Each client here is represented by a thread and a process, and operates independently from other emulated clients. However, traditional threading libraries or multi-processing sacrifice performance for simplicity. This is particularly true for kernel-level threads, which incurs a large performance hit due to frequent switching between the running threads. To our best knowledge, none of these load generation tools utilize user-level threads such as [47], which can both simplify the implementation and improve efficiency.

- **Accuracy:** The accuracy of load generation depends on many factors; most importantly, it depends on the implementation of the source traffic model. The basic idea is to emulate real users and each user is configured to send service requests based on some specific model or a pre-collected trace. For SPECWeb99 and SURGE, each emulated user is represented by a separate process or thread. httperf, on the other hand, uses an event-driven approach as outlined earlier. Each of these three tools focuses on implementing a different user behavior; each claims of being more representative of real users.

- **Scalability:** As servers grow in capacity, so does the need to emulate a larger number of users. However, an individual client emulation machine may not be powerful enough to stress-test these high-capacity servers. There is, thus, a need for such tools to grow beyond the boundaries of a single machine. Only SPECWeb99 has this capability, where any arbitrary number of machines can be configured. Both SURGE and httperf require manual configuration and data collection.

Overall, each of these tools has its strengths and weaknesses, possibly because each is developed with a specific goal in mind. SURGE has the most realistic *individual* client models. It, however, does not scale well nor does it emulate the arrival of new clients under overload scenarios. This concept will be revisited in Chapter 3. httperf, on the other hand, can be used to stress test powerful servers, but is not based on the behavior of real users. Finally, SPECWeb99 is the most popular and lies somewhere between the two.

2.3 Analysis of Queueing Systems

This dissertation follows an analytical approach for studying and improving NOSCs of Internet services. Specifically, our interest lies in one specific design problem, namely the control of general time-sharing systems with multi-threaded servers and multiple queues. This is indicative of the behavior of typical Internet services, where our main objective is to provide QoS guarantees to different queues while maximizing system utilization. This subsection briefly reviews some of the fundamental approaches for modeling and analyzing typical server-based systems.

The underlying concepts behind resource-sharing is embedded in many of the mechanisms in today's OSs and network systems. The basic idea is simple: divide the available resource into small chunks and distribute these chunks to competing applications or services. Resource-shared systems have been studied extensively in the literature, e.g., in [61, 73, 118, 121]. Its importance is not only apparent in computer-based technologies, but also in manufacturing and operations research. In general, modeling resource-shared systems requires simplifying the system under analysis to allow for mathematical tractability, while retaining those aspects that are essential to the system's behavior. In many cases, analytical models do not provide exact solutions. Nonetheless, they provide significant insight into the internal dynamics of real systems.

There are two resource-shared systems that are of particular importance to our study: time-shared and polling systems. Both improve our understanding of NOSCs when applied to Internet services.

- **Time-shared systems:** In its simplest form, a time-shared system consists of a server (e.g., CPU) and a buffer to hold arriving requests. The server processes each request for a fixed length of time, called a *quota*; if the request is not finished by the end of the allocated time quota, the server appends it to the end of the buffer and starts serving the next request in queue. The basic structure of a time-shared system mimics the operation of a typical multitasking OS. Here, each request represents an application or a task and the quota represents the time slice that each application is assigned before the CPU switches to a different application.

More detailed behavior can also be incorporated into the model, such as service discipline, priorities, switching overheads, multiple resources, etc.

- **Polling systems:** In network systems, e.g., routers and phone exchanges, the analogous of a time-shared system is a polling system. Here, a single server (typically a network processor) services incoming packets from multiple queues. The processor will switch between each queue using different policies (e.g., gated, exhaustive, limited, decrementing) [118]. Depending on the level of detail, the model may be continuous- or discrete-time, have single or infinite buffers, include switchover times or not, allow noncyclic order or only cyclic order of service, etc. What differentiates polling systems from time-shared ones is that switching generally occurs at the end of packet processing (i.e., there are no time quotas).

There are several methods for analyzing resource-shared systems; generally, they are centered around the use of stochastic processes. Simple well-behaved systems can be studied using Markov chains (both continuous and discrete). What complicates the analysis of a system are three fundamental parameters: the arrival distribution of requests, the service that each requires, and the resource-sharing model. Here, well-behaved systems would refer to these having Poisson arrivals and exponential service times, which are characteristic of call centers but not of Internet services—discussed later.

Unfortunately, Markovian analysis becomes less adequate for analyzing more complex behaviors. Several alternative techniques are available. A common approach approximates the system by using multiple stages of Markovian queues (also referred to as *phase-type distributions*) [74]. This method relies on a very useful result proving that for any arbitrary distribution G of a non-negative random variable, there is a sequence of phase-type distributions that converges in distribution to G. Furthermore, this sequence can be chosen such that they are arbitrarily close to the first (finite) r moments of G. This approach is desirable since it uses existing results from Markov chains. However, with complex distributions, the resulting stages (and consequently the state-diagram) becomes so complicated that this approach often produces inexact expressions requiring numerical solutions. A similar approach can be used to study multi-resource systems [79]. Here, each stage (or multiple of stages) represents a different resource. Similarly, depending on the complexity of the system model, exact solutions may become increasingly difficult. Renewal Theory and Regenerative processes provide more powerful methods for analyzing complex systems [107, 121]. The fundamental idea is to first study the embedded Markov chain, which is simpler and more tractable. Then, use well-established results to relate the embedded chain to the larger and more complex system model.

Analytical models of time-shared and polling systems provide a precise way of expressing the system's performance as the configuration is changed. Little's formula, for instance, relate the mean request arrival rate, λ, wait time, W, and the resulting number of requests inside the system, N, by $N = \lambda W$. Analytical models can, thus, be used to predict (to some extent) the performance of Internet services, which is clearly useful for optimizing NOSCs.

CHAPTER 3

PERSISTENT CLIENTS

This dissertation begins with an effort to model the behavior of real clients. The focus here is to incorporate a model of the client's reaction to the applied control, which, in subsequent chapters, will be used to improve existing NOSCs. The chapter also studies the impact of the client's reaction on the underlying traffic characteristics. Finally, this chapter presents an efficient and accurate tool for generating representative client traffic.

3.1 Introduction

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

Client persistence is not limited to FCEs but because it is ultimately rooted into TCP congestion control, it is a major stumbling block to providing responsive aggregate traffic controls during server overload and network congestion. As mentioned in Chapter 1, one aspect of this dissertation is to study the impact of control on aggregate traffic destined for web servers. We take the first step by studying the internal dynamics of typical Web clients and building a coherent model that captures the most important elements of the client's behavior for designing effective control mechanisms.

The focus of this study is on Web traffic. Unlike video or audio traffic, it is dominated by short-lived connections. Thus, we mainly focus on TCP connection establishment requests, namely SYN packets, as they represent the beginning of new user browsing sessions. Our approach differs from recent studies [16, 49, 89, 100, 125], where aggregate traffic is treated as a black-box and is characterized with inter-arrival time, round-trip time, time-scale dynamics, etc. Instead, we look inside the box to characterize the behavior of the building blocks of the aggregates, namely, the individual clients. Similar to [11, 35, 65], we are interested in exploring the interactions between clients, the network, and the end-server. We observe the existence of a hierarchy of factors contributing to the behavior of clients in FCEs. This has led us to a new model—which we call *persistent clients*—that is different from traditional models in that clients do not simply go away when their requests are dropped. Instead, they repeat their attempts until they succeed or eventually quit. Based on this new observation of client persistence, we build a coherent picture detailing the expected behavior of the entire aggregate. We found that this unfolds several unexpected behaviors of aggregate traffic during overload.

It is important for us to define what we mean by clients in the context of controlling aggregate traffic. Clients here are not just referring to real (human) users, but also (and more generally) to the suite of protocols (e.g., HTTP, TCP/IP) and applications (e.g., Internet Explorer, Netscape) that are used by the user to retrieve the information from a web server. To avoid any confusion, we use the term *user* to refer to real users and the term *client* to refer to our general definition above. We are, thus, interested in the interaction between the web browser, the Hyper-Text Transfer Protocol (HTTP) that is used to request the data, and finally the network protocols, namely TCP and IP.

The contributions of this chapter are fourfold. First, we analyze the dynamics of the internal mechanisms of individual clients in case of packet loss. We show that the persistence of individual clients plays an important role in aggregate traffic, which is further exacerbated by the allowed parallelism of web browsers. This is different from the traditional view of network traffic where the majority of packets belong to non-returning clients. Second, we analytically characterize the effects of client persistence on aggregate traffic and on client-perceived latency. Third, we present a surprising discovery of a re-synchronization property of retransmitted requests that exacerbates client-perceived delay when traditional control mechanisms are used. Fourth, we create an accurate model of persistent clients and explore the implementation challenges of effectively emulating such behavior.

This chapter is organized as follows. We first look at the anatomy of persistent clients in Section 3.2, where we isolate the contributing factors of client persistence. We then study the impact of client persistence on network traffic in Section 3.3. In Section 3.4, we combine our results to create a coherent model of persistent clients. We then, in Section 3.5, introduce *Eve*, a powerful and accurate persistent client emulation tool. This chapter ends with concluding remarks in Section 3.6.

3.2 Anatomy of Persistent Clients

Many factors contribute to the persistence of clients, whereby the client keeps trying to access the server (normally at a later time) even after server overload or network congestion is detected. Some factors of this persistence are embedded in the applications and protocols that clients use. These are not design flaws, but are often necessary to the proper operation of clients, e.g., TCP congestion control. Other factors are due to purely human habits. We isolate five (non-user related) factors that can affect the persistence of a typical client's access to a web server (Figure 3.1). In the process of analyzing them, they are grouped in two separate categories: network-level and application-level factors. While we only focus on non-user related factors, our proposed control mechanisms ultimately reduce the client-perceived latency; this indirectly reduces the impact of user-related factors as, for example, users are less inclined to press the reload button on their web browsers.

3.2.1 Persistence in Network Protocols

In this subsection, we investigate how the combination of TCP congestion control and different queue-management techniques in routers and end-servers may raise the severity of FCEs. We use a simple model where a client issues a single Hyper-Text Transfer Protocol (HTTP) request using a separate TCP connection. This model allows us to study a single TCP connection in isolation. Consequently, our findings fit well with clients implementing the HTTP 1.0 specification. In Section 3.2.2, we extend our results to HTTP 1.1 clients, where several HTTP requests can be multiplexed into a single connection.

Consider what could happen to our simple client's request during an FCE. Before examining the consequences of the request packets being dropped by routers or end-servers during the various

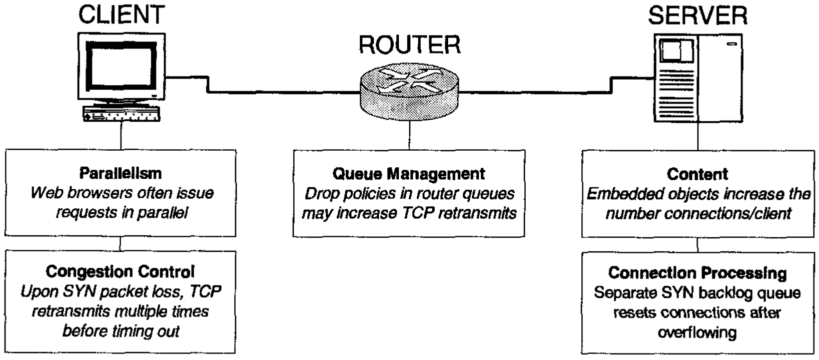

CLIENT
ROUTER
SERVER
Parallelism
Web browsers often issue requests in parallel
Queue Management
Drop policies in router queues may increase TCP retransmits
Content
Embedded objects increase the number connections/client
Congestion Control
Upon SYN packet loss, TCP retransmits multiple times before timing out
Connection Processing
Separate SYN backlog queue resets connections after overflowing

Figure 3.1: Non-user related factors affecting clients' persistence.

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

During overload, packets are lost due to any of the three types of queues filling up: router queues, server SYN backlog queues, and server (or application) listen queues. Packet drops by different queues may trigger different reactions from the client as part of recovery from packet loss. Here we consider loss of packets on the path from the client to the server. An equivalent behavior occurs on the reverse direction.

Packet Drops at Router Queues

When router queues fill up and packets are dropped, the request can be in the connection-establishment stage or the connection has already been established. In the first case, each time a SYN packet or its corresponding response is lost, an exponentially-increasing retransmission time-

		FreeBSD	HP-UX 11	Linux 2.2/2.4	Solaris 2.7	Win 9x, NT	Win 2000
Drop behavior	Drops SYN *RTO (sec)*	3, 9, 21, 45	2, 3, 9, 21, 45	3, 9, 21, 45	3.4, 10.1, 23.6, 50.6, 104.6, 164.6	3, 9, 21	3, 9
	Drops connection w/ sending reset	Connects using 3-way handshake and sends a request packet. The server drops the connection and will either send the RST packet after the connection is dropped or after receiving the request packet. The client will then abort the connection					
	Drops connection w/o sending reset	Connects using 3-way handshake and sends a request packet. The server drops the connection and will keep dropping subsequent packets from the client. The client will time out and retransmit the request packet. The timeout intervals are based on the computed RTT values during the connection establishment phase.					
	Drops connection after receiving data	All clients connect using 3-way handshake, send a request packet, receive ack, and wait					
Reject behavior	Receive ICMP *destination unreachable*	Client aborts the request				Ignores ICMP packet and retransmits after timeout	
	Receive RST packet	Client aborts the request				Retransmits SYN packet immediately after receiving RST packet	

Table 3.1: Retransmission behavior for different OSs. The measurement assumes default OS configuration. Some parameters such as the timeout before the connection is aborted, can be dynamically configured.

out (RTO) is used to detect the packet loss and the SYN packet is retransmitted.[1] The RTO values used by different client OSs are listed in Table 3.1. Established connections, in the latter case, detect and recover from packet loss using more complex mechanisms than what is just described. These have been investigated by several studies, both empirically and analytically, e.g., in [29, 97, 98].

To illustrate the effect of packet drops, we configured a Linux-based machine running Eve (described later) to emulate clients arriving independently with exponentially distributed inter-arrival time with rate 75 clients/sec. The server machine runs Eve, a homegrown server emulator, which is configured to emulate a server with a buffer capacity of 50 requests (i.e., listen buffer size = 50) and process one request at a time with exponential service times with mean 1/50 sec. A third machine

[1] Most TCP stack implementations follow Jacobson's algorithm [68, 103], where a SYN packet that is not acknowledged within an RTO period is retransmitted, but with the previous RTO period doubled. This is repeated until the connection is established or until the connection times out, at which point the connection is aborted.

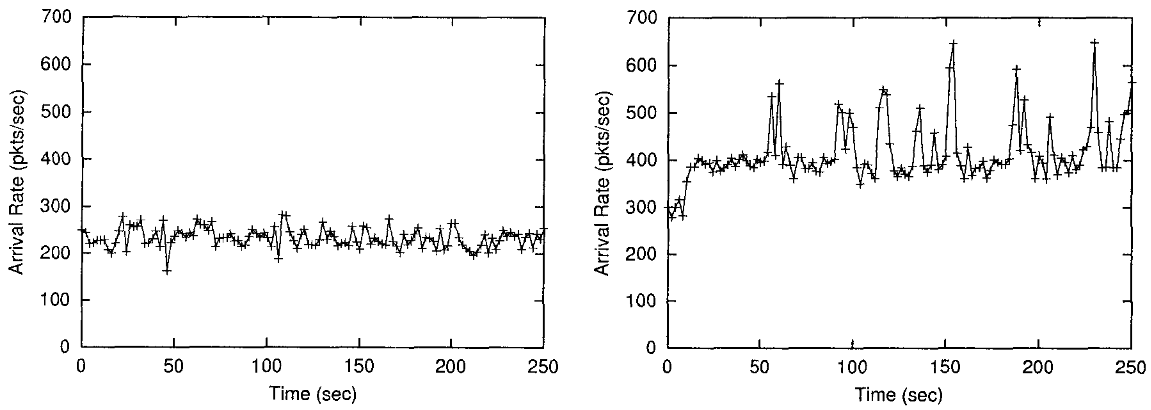


Figure 3.2: Effects of client's persistence on arrival rate: (left) Non-persistent clients and (right) Persistent clients both being dropped with probability 0.5.

is configured as a router running NIST Net to emulate a WAN with a uniform one-way delay of 50 msec. The three machines are connected via FastEthernet link and are ensured to be bottleneck-free. Finally, we used tcpdump to collect our measurements. The actual arrival rate of packets (including data packets) from the client to the server is plotted in each graph.

Figure 3.2 illustrates the increase in the number of packets when persistent clients are assumed. The figure compares server or router overload for persistent and non-persistent clients. For the tested configuration, the total amount of traffic increased by 76% when assuming persistent as opposed to non-persistent clients. It is also burstier. The client-perceived delay also increased from approximately 22 msec in the case of non-persistent clients (left figure) to 3318 msec for persistent clients (right figure). We emphasize that the delay measurements do not include the clients who timed out. In Section 3.3, we examine these issues in detail.

Packet Drops at SYN Backlog Queues

When the backlog queue at the server fills up, the server can be configured to drop incoming SYN packets, which triggers a similar retransmission behavior as discussed above. The server can also be configured to send SYN cookies to the client. A SYN cookie is simply a method for the server to avoid storing any state for half-open connections. There, a challenge is sent to the client and upon its return, the server can establish the connection as if the original SYN packet were queued properly in the backlog queue. The challenge is encoded in the TCP's sequence number and, thus, does not require any client modification. When SYN cookies are lost, the client times out and retransmits the request as described above. SYN caches are an alternative method to SYN cookies, which allow the server to store a large number of SYN packets by simply delaying the creation of connection data structures until the three-way handshake is completed [81]. Depending on the size of the cache and the arrival rate, SYN caches can fill up just like SYN backlog or router queues. Table 3.1 shows the OSs that support SYN cookies and SYN caches.

Both SYN cookies and SYN caches are effective in handling a flood of SYN packets, the majority of which are spoofed (or fake). The mechanism relies on the fact that only a small portion of the SYN-ACKs will be replied back, after which the TCP connection is fully established. When requests originate from legitimate clients, both mechanisms increase the additional work on the end-server as the resulting fully-established connections (from the clients' perspective) are dropped due to insufficient room in the application listen queue. As we show shortly, this is true even if SYN packets are not accepted when the listen queue of the application fills up.

Packet Drops at Application Listen Queues

Once application listen queues fill up, no connections can be established and the backlog queue drops incoming SYN requests.[2] However, some of the queued SYN packets can complete the three-way handshake while the listen queue is full. In this case, the OS can either drop the fully-established connections all together or drop the SYN ACK and force the client to retransmit (which may eventually cause the connection to timeout).

This introduces an important consequence of having a separate SYN backlog queue that deserves careful examination. As explained earlier, the backlog queue is generally independent of applications' listen queues to increase the server's resilience to SYN flood attacks (Table 3.1).[3] However, when the listen queue fills up, incoming SYN packets destined for the corresponding application are dropped even if there is room in the backlog queue. This is to avoid the situation where the three-way handshake is completed and the connection requests may be dropped due to the lack of room in the listen queue.

If a connection is dropped, the server may or may not send a reset packet. As shown in Table 3.1, not sending a reset packet triggers further retransmissions by the client. In fact, since RTO is based on round-trip time (RTT) estimates, the client sends larger data packets more often than when the

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

[3] SYN floods are a form of denial of service (DoS) attacks where the adversary floods the server's queue with fake SYN requests to prevent legitimate requests from being accepted by the server. Because a SYN request has a long lifetime at the server, long backlog queues as well as the use of SYN cookies or SYN caches are generally recommended [81].

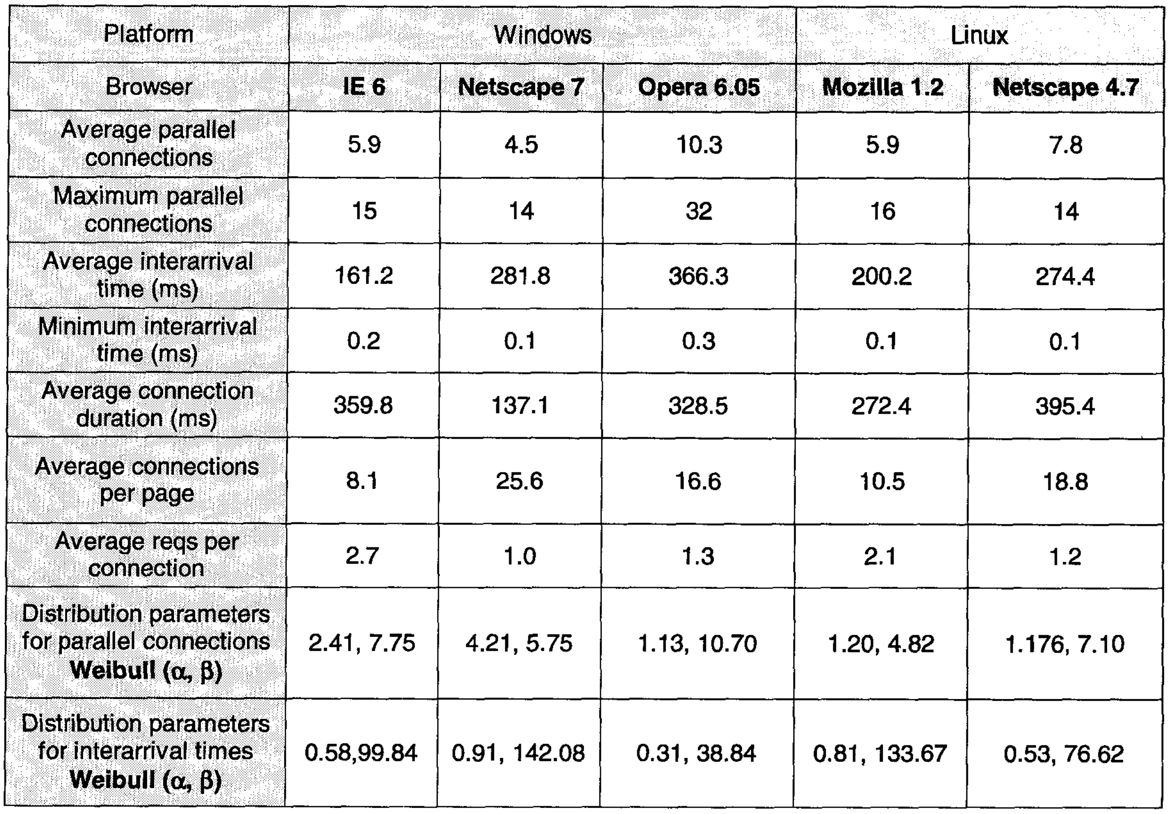
SYN packet is dropped and the connection is not allowed to complete. On the other hand, if a reset packet is sent back to the client when the connection is dropped, the server cycles between two phases. The first phase is when established connections are dropped and reset packets are sent back to the client. The second phase is when SYN packets are dropped and then retransmitted after exponentially-increasing timeouts.

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

3.2.2 Persistence in Client Applications

Almost all the web content is organized such that a main object (or page) is first retrieved and followed by all of its embedded objects. Web browsers generally issue multiple requests to the embedded objects in parallel to maximize the throughput and minimize the time of content retrieval. Even with the availability of persistent connections in HTTP 1.1, where browsers are encouraged to use a single connection to take advantage of TCP's larger window size,[4] parallelism is still used by most browsers. Traditionally, the added aggressiveness of parallelism was analyzed from the perspective of network bandwidth or server resource sharing [11]. In this subsection, we look at clients' parallelism as a factor contributing to the increase in the severity of FCEs since a single client can issue multiple *independent* connection requests. As shown in Figure 3.1, there exists an interplay between the client's browser implementation and the server's content organization (or encoding) that determines the degree of parallelism, and hence, the aggressiveness of the clients. On

[4] By using a single connection to request multiple objects, the TCP window, in theory, grows to match the throughput of using multiple short connections in parallel [90].



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

Table 3.2: Parallelism of different browsers. Values obtained from accessing 250 different links from the top 25 websites. On average, there was 21.2 unique objects and 71.2 non-unique objects per page (excluding any Java or JavaScript).

the server side, having many embedded documents forces clients to issue multiple HTTP requests (not necessarily in parallel). We have found from our experiments that there are an average of approximately 21 unique objects embedded in each page. Browsers on the client's side are free to request these objects using separate connections in parallel, series, or in a single connection using persistent connections in HTTP 1.1.

To investigate this issue more thoroughly, we analyzed the behavior of five popular browsers on two platforms, Linux (Kernel 2.4) and Microsoft Windows 2000. We focused on each browser's degree of parallelism, how the parallelism changes with different web content, and how each reacts to packet loss during the retrieval of the primary or the embedded web content. Ten random links from the top 25 websites[5] were used to test the five browsers. Each browser was configured to request the same 250 links. An intermediate machine running `tcpdump` was used to intercept and

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

record all packets to and from the server. To improve the consistency of our results, we cleared the data cache after every visited page. Furthermore, each browser was configured to fetch the set of links 5 times over various times of the day. This way, we also minimize the effects of the time of day on our conclusions. We note, however, that there is a large set of configurations for each browser. Testing all of them would require long hours of manual labor. We, thus, used the default values as the basis for our conclusions.

Table 3.2 summarizes the parallelism of the different browsers. We derived these numbers by careful analysis of the generated logs. Our log analyzer kept track of connection start and end times by identifying the corresponding SYN, FIN, or reset packets. We were conservative in making measurements since we considered a connection terminated if it remains idle for 500 msec, which is roughly 5 times the average RTT value. Here we assumed that the implementer did not bother to close the connection before issuing a new one. The table shows (among other things) the statistical averages for the number of parallel connections and their interarrival times. The computed values are based on arrival epochs of new connections; they are not time averages. That is, each time a new connection is detected, a snapshot of the system is taken and based on their ensemble, the averages are computed. Table 3.2 also shows the mean number of HTTP requests per TCP connection, which is the number of per-page unique objects divided by the number of issued HTTP requests. We chose to use the number of unique objects as we assume browsers perform some intelligent caching. Our reported numbers are, thus, conservative since, in addition to using the number of unique objects, they also do not include any additional objects requested by any embedded Java applets or JavaScripts in the page.

Several conclusions can be drawn directly from the table.

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

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

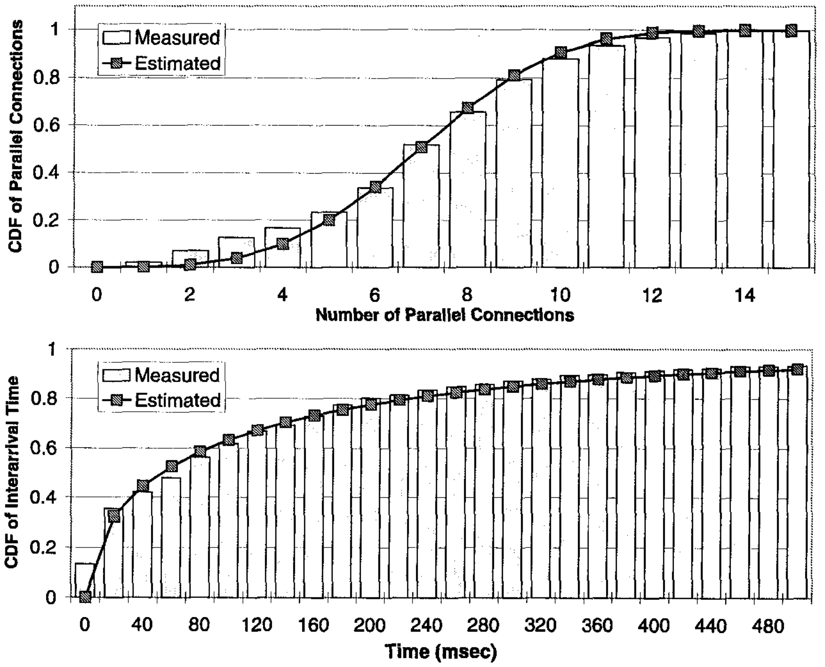


Figure 3.3: Internet Explorer 6.0 browser distributions: (top) distribution of parallel connections and (bottom) distribution of interarrival times. Both are approximated by the Weibull distribution with parameters in Table 3.2. In the bottom graph, the distribution converges to 1 after 4500 msec. We truncated the plot to magnify the important region.

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

The last two points can be verified further by inspecting the distributions of the number of parallel connections and their interarrival times. These are shown in Figure 3.3 as Cumulative Distribution Functions (CDF). We only plotted the distributions for Internet Explorer (IE) 6.0 because of space limitation. A value on the ordinate (y-axis) of the top graph, for instance, should be read as the probability of having a maximum of x parallel connections, where x is drawn from the abscissa (x-axis). Both plots, then, confirm the behaviors in C2 and C3. We first note that because measurements of the number of parallel connections are taken at the arrival epochs of new connections, the computed average should approximately be a half of the allowed maximum when the maximum allowed number of connections is close to the number of embedded objects. For example, if there are 18 embedded objects and the maximum allowed number of parallel connections is 15, then we have the following set of measurements: $\{1, 2, 3, \ldots, 15, 15, 15, 15\}$, which yields an average value of $(1 + 2 + \cdots + 15 + 15 + 15 + 15)/18 = 9.17$. Here, once we reached the maximum, the remaining three objects had to wait for three other connections to complete. This is the observed parallel behavior in Figure 3.3. The figure also verifies C3. It shows that 50% of the connections arrive less than 50 msec apart, but the bulk are spaced a few seconds apart.

Using standard distribution fitting techniques [63], we are able (in most cases) to approximate both parallel and interarrival distributions using the Weibull distributions with parameters detailed in Table 3.2. By observing the large variations in the distribution parameters, we find that no concrete conclusion can be drawn from our measurements regarding the exact distribution of browser behavior. Specifically, we observe that different times of the day produce large variations, which is probably due to changing server and network loads. As we show in Section 5.4.1, instead of using analysis of variance techniques to describe browser behavior under different load scenarios, we construct a simple model of their internal mechanisms. This model allows us to draw direct conclusions on the optimal way of controlling web clients.

Based on Table 3.2, we see the benefits of persistent connections in HTTP 1.1. This is shown in the large difference between the average number connections per page for IE 6.0 and Mozilla 1.2, and the other three browsers. There is, however, one caveat in using persistent connections: it may cause servers to run out of available file descriptors much faster than requiring connections to terminate after each request [70]. Once the application runs out of file descriptors, it no longer is able to accept additional connections, which causes the listen queue to fill up. To combat this problem, the server can dynamically reduce the keep-alive timeout in HTTP 1.1 as the number of descriptors approaches the maximum capacity.

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

3.3 Impact of Persistence on Network Traffic

To accurately study the effects of persistence on network traffic, we must identify the reactions of individual connections as well as flow aggregates to packet loss. Section 3.2 provided a detailed study of the former, namely, the individual client. In this section, we look at how client persistence affects aggregate traffic. An mentioned earlier, we continue to focus on SYN packets as they are the object of control for NOSCs.

3.3.1 Increased Aggregate Traffic

To better understand the dynamics of FCEs, we extend some of the results in [29] that pertain to connection establishment. We follow the same modeling assumptions in [29] and build on its estimate of the connection-establishment latency. We, thus, assume that end-points adhere to a TCP-Reno style congestion control mechanism [68]. However, to draw general conclusions for the entire aggregate, we must also characterize the arrivals of new connection requests, namely, their interarrival times are *independently and identically distributed (i.i.d.)*, and are exponentially distributed. Assuming *i.i.d.* implies that dropping one request packet does affect the arrival of future request packets. This matches well the observation that clients behave independently. It, however, does not consider the inter-dependency between requests from a single client. Based on these two assumptions and elementary queueing theory, one can also show that SYN-ACKs on the reverse direction have the same *i.i.d.* distribution. Furthermore, under a uniform drop policy (i.e., incoming requests are dropped with an equal probability), the retransmitted requests have *i.i.d.* and exponentially-distributed interarrival times.

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

$$EL_h = \sum_{j=0}^{n}[p^j(1-p)(T_j + RTT)] + p^{n+1}T_{abort}, \tag{3.1}$$

where p^{n+1} is the probability that the connection times out and RTT is the mean round-trip time. The first term, then, represents expected latency of successful connections, or $(1-p^{n+1})E[L_h|x \text{ succeeds}]$, which was derived in [29]. By the independence assumption, it can be easily shown that EL_h is also the mean expected connection-establishment latency of all requests.

Under our network model, we also derive Λ, the effective or aggregate arrival rate of SYN

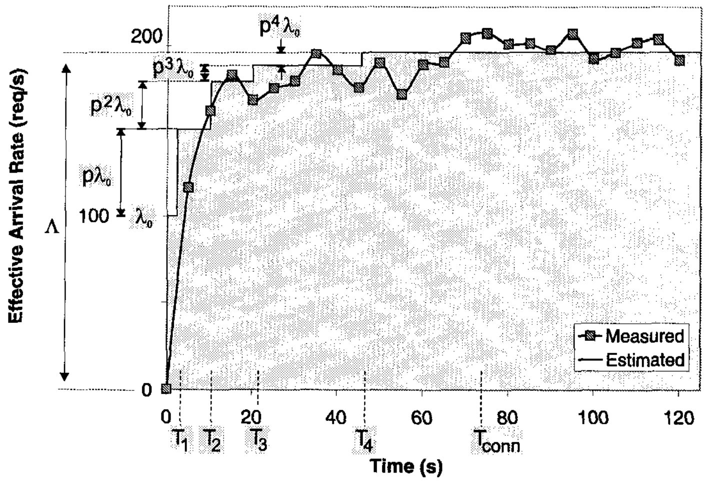


Figure 3.4: Increase in network load caused by packet loss. Clients with an average arrival rate $\lambda = 100$ reqs/s have their requests dropped with probability $p = 0.5$.

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

$$\begin{aligned}\Lambda &= \lambda_0 + \lambda_1 + \ldots + \lambda_n \\ &= \lambda_0 + p\lambda_0 + p^2\lambda_0 + \ldots + p^n\lambda_0 = \frac{1-p^{n+1}}{1-p}\lambda_0. \end{aligned} \qquad (3.2)$$

Notice the simple relationship between the arrival rates of the different transmission classes. For example, the arrival rate of the first retransmission class, λ_1, is just the arrival rate of the initial transmissions, λ_0, times the drop probability p. Based on Eq. (3.2), a 50% drop at the router will, in theory, increase the amount of new connection requests by 75%.[6] Therefore, under a uniform drop strategy—regardless whether it is random dropping, token-based dropping, or drop tail—the persistent property has two important consequences:

- The effective arrival rate increases with increased drop probability as defined by Eq (3.2). The maximum is achieved when all incoming packets are dropped and is equal to $(n+1)\lambda_0$.
- A typical rate controller only causes $p^{n+1}\lambda_0$ connections to time out; for the rest, increasing the number of retransmissions has a substantial impact on the client-perceived delay as shown

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

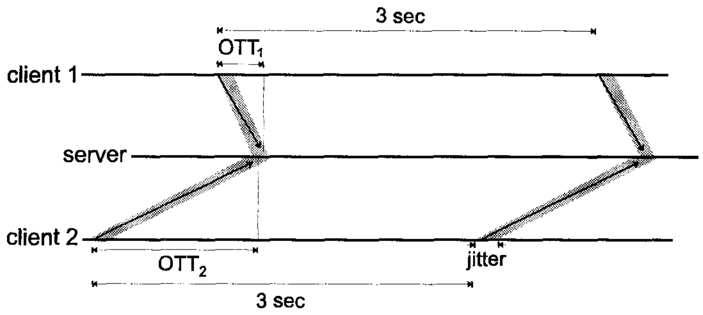


Figure 3.5: Synchronization of two SYN requests with different one-way transmission time (OTT) values and network jitter, but with the same retransmission timeout (RTO) values.

in Eq. (3.1). This probability, which we call the *effective timeout probability* (p^*), reflects the true impact of the control mechanism on the underlying traffic.

The accuracy of the above consequences rely on an important assumption, namely, that incoming requests are independent. This assumption is, unfortunately, an over-simplification of real traffic generated by real users. It is, nonetheless, necessary for mathematical tractability. In subsequent chapters, we continue to assert this assumption when building our models. We, however, relax it during the evaluation of our new NOSC mechanisms.

3.3.2 Re-synchronization of Bursty Traffic Requests

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

Consider a scenario in which a burst of new connection requests arrive at a server and are then dropped. As shown in Table 3.1, most clients will time out after 3 seconds. This timeout is measured from the transmission of a packet, not the time it is dropped. Thus, assuming that network conditions remain similar, the retransmitted requests will arrive at the server roughly 3 seconds after the originals, irrespective of network latency from the client to the server. As a result, the burst of drops will cause a burst of retransmissions to arrive roughly simultaneously at the server. This is exemplified in Figure 3.5, where the arrival of the two retransmitted packets depends only on the timeout value, T_k, and network latency jitter for each request. Therefore, any dropped burst may be repeatedly dropped and retransmitted as a burst until the corresponding requests time out, as

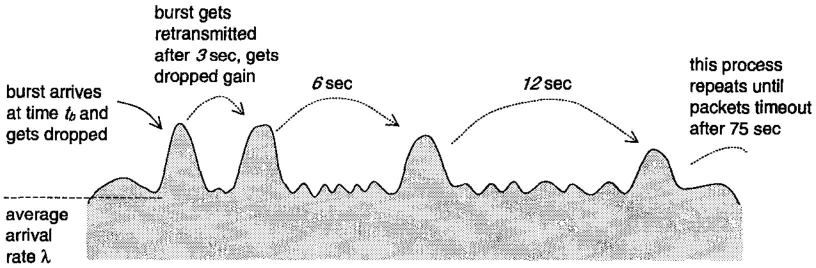

burst gets retransmitted after 3 sec, gets dropped gain
burst arrives at time t₀ and gets dropped
6 sec
12 sec
this process repeats until packets timeout after 75 sec
average arrival rate λ

Figure 3.6: Typical dropping behavior of a burst of SYN requests

exemplified in Figure 3.6. This, however, only shows how synchronized retransmissions may arise and the general effect on the arrival rate at the server. Specifically, we identify three factors that have direct effect on the synchronization of retransmitted requests: (1) the distribution of RTO values, (2) the distribution of network jitter, and (3) the length of incoming bursts. If the network jitter or the RTOs are distributed over narrow intervals, we can expect retransmissions of a dropped burst to occur in highly coherent, synchronous bursts. Otherwise, the retransmissions will be less coherent, and spread out over a longer time interval than the original incoming burst.

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

Figure 3.7(top) shows the resulting measurements for both the first and second RTOs. We

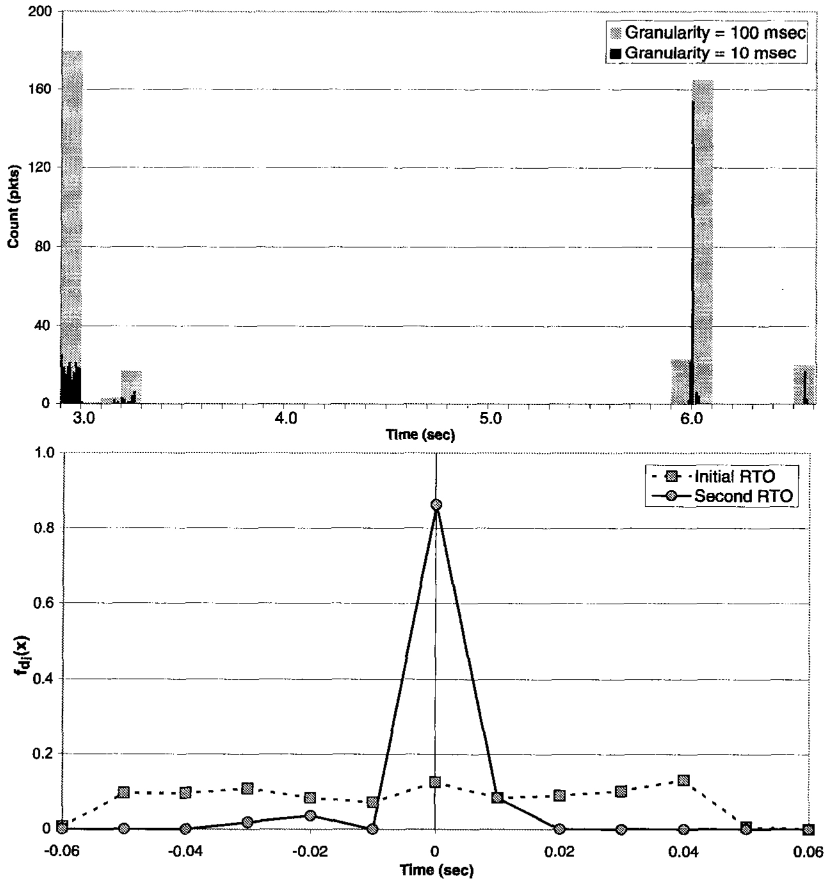


Figure 3.7: RTO distribution for Windows 2000 clients. Measurement were taken from 22 different machines with highly varying hardware. (top) The actual RTO for all clients. (bottom) Probability density function (pdf) around the mean for both initial and second RTO.

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

To estimate $f_{d_i}(x)$, we shift each data point by the mean RTO value of the corresponding samples (excluding the outlier machines), and plot the resulting distributions. Figure 3.7(bottom) shows $f_{d_i}(x)$ for $i = 0, 1$. On visual inspection, f_{d_0} seem to follow a uniform distribution, whereas f_{d_1} seem to follow the normal distribution. We use the Kolmogorov-Smirnov (K-S) test [31] to verify this. The distribution of the initial RTO passes the K-S test, which indicates that $f_{d_0}(x)$ can indeed be represented by a uniform distribution. However, the second RTO does not pass the K-S test, due mainly to the higher concentration of the data around the mean. In fact, 75% of RTO values are within a 2 msec interval and 90% of the second RTOs are within a 5 msec interval, which is close to the expected error in our measurements. The measurements for the third RTO from our Windows 98 and NT experiments (not shown) also have similar characteristics. Based on this, we can describe $f_{d_i}(x)$ as follows:

$$f_{d_i}(x) = \begin{cases} 0.01 & \text{for } i = 0 \text{ and } |x-3| < 50 \text{ msec} \\ 0.5/\epsilon & \text{for } i = 1 \text{ and } |x-6| < \epsilon, \text{ as } \epsilon \to 0 \text{ msec} \\ 0.5/\epsilon & \text{for } i = 2 \text{ and } |x-12| < \epsilon, \text{ as } \epsilon \to 0 \text{ msec} \\ 0 & \text{otherwise.} \end{cases} \tag{3.3}$$

The RTO distribution alone is not sufficient to determine retransmission burst coherency. We must also look at the distribution of jitter in network delay. Characterizing network delay, in general, has been the focus of many studies [3, 95, 126]. There are, however, two relevant points that one can conclude from these studies. First, network delay varies across different links and over long time scales. Specifically, Mukherjee [95] described packet delay using a shifted gamma distribution with varying parameters across different network paths. Second, over short time (less than 30 minutes) scales, link delay is relatively stable, except for the occasional spikes in delay. Paxson [126] found that standard predictors (such as weighted moving average) can give a good indication of future delays. Furthermore, Acharya [3] empirically verified that delay jitter is confined to a 10 msec interval for short time (less than 103 secs) scales, which is well below the RTO range. Since we are only interested in determining the degree to which a burst will remain concentrated as it is retransmitted, we can use this result to model jitter. Specifically, we can use a uniform 10 msec density function to describe network jitter, $f_J(x)$, over short time scales as follows:

$$f_J(x) = \begin{cases} 0.1 & \text{for } |x| < 5 \text{ msec} \\ 0 & \text{otherwise.} \end{cases} \tag{3.4}$$

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

R1 A dropped burst $B(t)$, once retransmitted, will spread out at most over a 220 msec interval compared to the original burst. Subsequent retransmission attempts will spread the bursts by at most an additional 20 msec per retransmission. Therefore, short bursts, once dropped, will have retransmissions that spread out significantly, and will therefore be mitigated. Longer

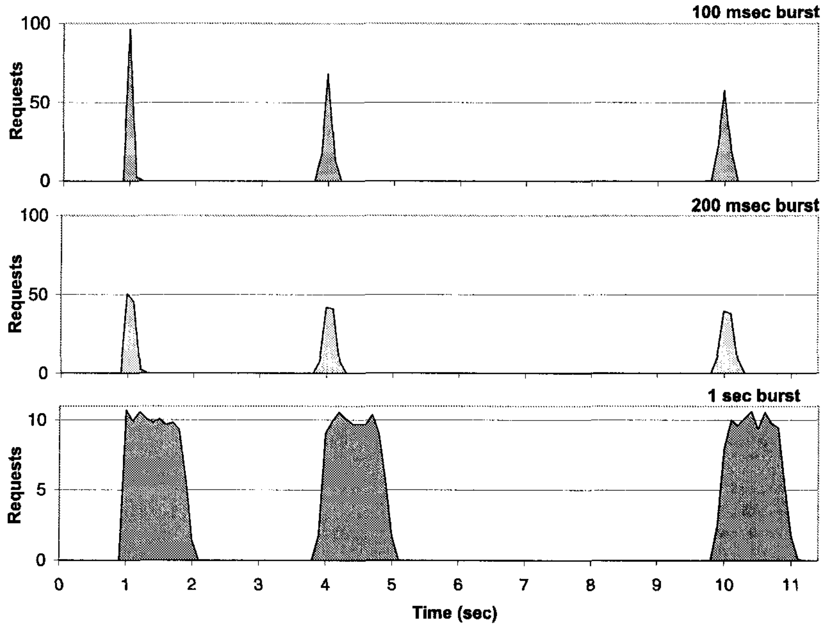


Figure 3.8: Re-synchronization of SYN bursts. A 100 requests burst arrives at a 100 msec, 200 msec, 1 sec intervals, respectively and are dropped.

bursts, however, will not be significantly affected by these low spread-out times, so the retransmissions will be re-synchronized and arrive at the server as bursts of equal intensity.

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

We use *eSim*, a simulation tool, to confirm this effect. Our simulator is based on the Linux implementation of the TCP stack and allows us greater flexibility in specifying server resources than would *ns*. A set of 100 clients are configured to send connection requests to a single server with enough network bandwidth to handle the maximum burst. Each client has an RTO distribution that follows the above $f_{d_i}(x)$ and a network latency jitter that is uniformly-distributed over a ± 5 msec range. We vary the time alignment of the initial client transmissions such that a burst of requests arrives at the server spread over 100 msec, 200 msec, and 1 sec time intervals. The server, in turn, drops all incoming connections. Figure 3.8 shows the arrival pattern at the server and the effects on the shape of retransmission bursts. Clearly, the narrow initial burst arrival is spread out significantly upon retransmission, and the burst intensity is reduced. As the burst duration increases, the effects of the variations in link delay and RTO become inconsequential, and the bursts occur at an equal intensity.

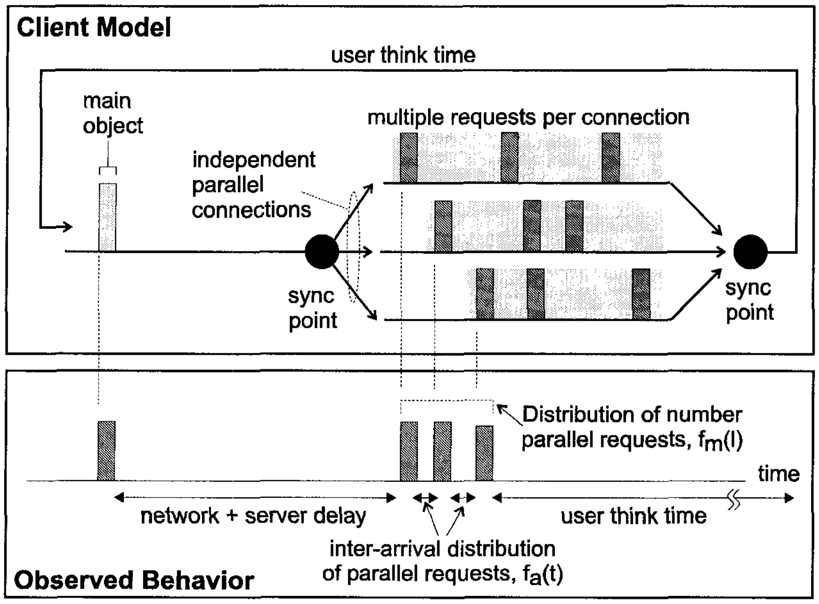

Client Model
user think time
main object
multiple requests per connection
independent parallel connections
sync point
sync point
Distribution of number parallel requests, fm(l)
time
network + server delay
user think time
inter-arrival distribution of parallel requests, fa(t)
Observed Behavior

Figure 3.9: Persistent client model.

3.4 Modeling of Persistent Clients

Creating accurate models of web clients is not an easy task. Several studies have empirically studied the interaction between the client, network, and end-server to characterize the dynamics of underlying traffic [16,50,51]. Unfortunately, such studies often lack the clients' response to different control policies, which is the main ingredient for constructing effective controllers.

We are faced with the question of whether an effective traffic controller can be built without exact knowledge of client behavior. We argue that an optimal controller can be realized by approximating the internal structure of web clients. The model of persistent clients is presented in Figure 3.9; it captures the following four important elements.

E1. Individual clients are independent of each other, and a client's requests are grouped into *visits*. Each visit represents a client accessing a web page and its entire content. Requests within a visit are correlated by the completion of the initial page that contains all the embedded links.

E2. Once the main page is fetched, a batch of l parallel connections with probability distribution $f_m(l)$ are created to request the embedded objects with arrival distribution $f_a(t)$. We do not specify the exact distributions for $f_m(l)$ or $f_a(t)$, but in our subsequent derivations, they are assumed to be independent and have finite means. Moreover, the retransmissions of lost packets from parallel connections are independent as long as none of the connections is aborted.

E3. The expected visit completion time, EV, is the sum of the time it takes to fetch the initial page and the longest finish time of all parallel connections. Formally, consider p^* as the effective timeout probability, $q^* = 1 - p^*$ as the probability of success, m as the expected number of parallel connections, and γ as the mean interarrival times of the parallel connections. Also, consider ET_c as the expected latency for completing a single request (we consider better estimates for ET_c in Section 4.2.2). Then,

$$\begin{aligned} EV = {} & p^* T_{abort} + \\ & q^* \left[ET_c + (1 - (q^*)^m)(T_{abort} + \Psi) + (q^*)^m (ET_c + \Psi) \right]. \end{aligned} \tag{3.5}$$

The second term in Eq. (3.5) estimates the expected delay when the first page is fetched successfully. The term $(1 - (q^*)^m)$ represents the probability that at least one of the m connections times out and $\Psi = m\gamma$ is the approximate overhead of launching m parallel connections. Thus, the last product term in Eq. (3.5) is the expected delay for completing the parallel requests. It is derived by taking the expectation of their maximum completion time.

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

In the absence of packet loss, our model is consistent with earlier ones (Observed Behavior in Figure 3.9) where it is assumed that a client sends a batch of closely-spaced connection requests (*active period*) followed by a relatively long period of user think time (*inactive period*) [16]. Our distributions have similar characteristics to the ones in [16,49,51] with very different distribution parameters. Our model, however, captures the effects of the applied control, which we use to construct an optimal controller.

3.5 Emulating Persistent Clients

The importance of accurate emulation of real users to evaluate the performance of server has long been recognized. However, the advent of high-capacity servers has elevated the complexity of performance analysis to a much higher level as proper analysis involves an accurate emulation of thousands, if not tens of thousands, of simultaneous and independent clients. In particular, accurate emulation requires two important elements: (1) each client is representative of real user behavior similar to our model of client's persistence, and (2) the arrival of new clients—not their requests—is independent of the progress of existing ones. As we noted in Section 2.2.4, existing client emulation tools often lack one of these elements (if not both). In this section, we present an efficient tool that is able to capture both elements and also provide simple mechanisms for measurement collection and future extensibility.

Traditionally, client emulators are built over the popular thread abstraction, where each thread represents a single client. Threading is attractive because straight-line codes (or scripts) are easily parallelized to emulate multiple clients. However, existing threading implementations become problematic when a very large number of clients are emulated because of their high resource and management overhead. Two mechanisms are commonly used to overcome this limitation. One mechanism simply limits the number of threads or clients [37]; a client waits for response from the server before issuing a new request. The authors of [13, 94] have shown that this can significantly impact the performance numbers since it does not sustain a specific request rate. Instead of threads, an event-driven design that is based on non-blocking I/O is used to sustain the request rate [13, 94], but this incurs added complexity.

The heart of the design challenge is scalability. On one hand, client emulators should be powerful enough to model complex concepts such as persistence. On the other hand, thousands of such clients must be emulated in order to stress-test high-capacity servers. In this section, we present *Eve*, an efficient and highly-customizable client emulation tool for creating client models. *Eve* consists of two powerful abstractions. The first is an optimized user-level thread library that simplifies client emulation. This library combines the ease-of-programming of traditional threading implementations with the improved performance of event-driven counterparts. The second abstraction is a distributed shared variable (DSV) architecture that simplifies the management of, and supports the scalability of, clients across multiple machines. This efficient communication abstraction is needed to facilitate client management, data collection, and event synchronization. *Eve* uses a modular design to integrate both abstractions into an extendable tool that can stress-test powerful multi-tier servers.

3.5.1 Design

Eve is designed with future extensibility in mind. While we only focus here on emulating persistent clients, *Eve* is a programming environment that provides powerful mechanisms for specifying

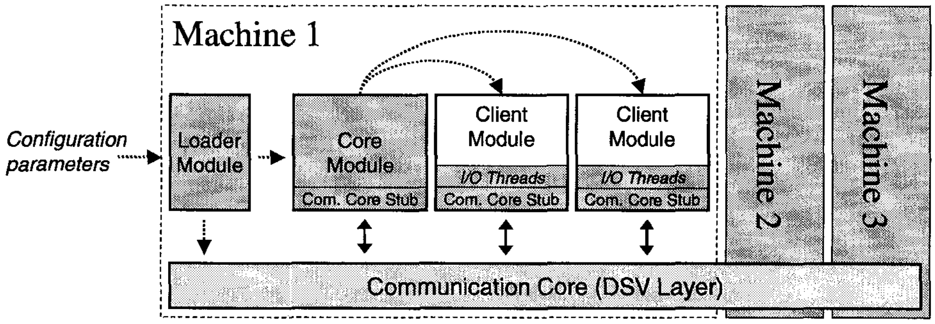


Figure 3.10: Design of *Eve*

and emulating a wide range of client models as well as taking measurements. It is implemented using the C programming language on the Linux operating system (kernel version 2.2.14). With the exception of a single header file, cross-platform porting is straightforward. *Eve* has the capability of running as many instances of a given client model (or a combination of models) as necessary to analyze servers under test. As the need for more resources arises, *Eve* can be configured to add more machines to expand its set of usable hosts.

Figure 3.10 shows a high-level design of *Eve* in which it is decomposed into three basic components:

- **Modules** constitute the basic building blocks of *Eve*. They are connected through a standard interface, and offer a great deal of flexibility in extending and scaling *Eve*'s functionality.
- **Communication Core** glues all modules together and because it is distributed, *Eve* easily scales beyond the boundary of a single machine.
- **I/O-Threads** are used to simplify the creation of a single client. This highly optimized user-level thread library abstracts away all complexity of using non-blocking I/O without losing any performance advantage.

3.5.2 Eve Modules

To allow a high degree of customization, *Eve* implements all of its components, including its core components, as modules. *Eve* modules are implemented as plug-in dynamic link libraries (DLLs) where each DLL is required to implement two interface points, `eve_init` and `eve_exit`, for initialization and clean-up, respectively. Modules are free to export other functions to provide added functionality. A unique identifier is associated with each module and is part of the modules configuration parameters. This identifier allows modules to reference each other's shared variables though the Communication Core (described shortly), to update or obtain measurement data. Furthermore, the identifier is used to load and unload modules using the `eve_load` and `eve_unload` primitives, respectively.

There are two types of modules: client and core. Core modules are used to initialize all client modules, facilitate communication between these modules, and at the end of an experiment, allow for the reporting of measurements. The bare-bone *Eve* (shaded boxes in Figure 3.10) consists of a loader module that initializes the Communication Core, sets up environment variables based on the user-defined configuration parameters, then loads the Core modules. A client model (e.g., one that mimics user shopping behavior) is encapsulated in a separate module and uses the interface points to connect to other modules. Figure 3.10 highlights a generic design of a client module where it shows the interaction between the client and the Communication Core, and other modules.

3.5.3 Communication Core

Eve implements its communication core using a distributed shared variable (DSV) layer (Figure 3.10). It is used by individual clients to communicate among themselves to share data and perform synchronization. Since typical emulation data is mostly shared at the beginning—during the initialization stage—and at the end—during the statistic collection stage—with the occasional data sharing during synchronization, a simplified version of Munin [17] is used to implement this layer. Our design follows a client/server model, where modules (the clients) use a thin stub to establish a communication channel to the DSV repository (the server).

Eve provides two standard calls to access shared and synchronization variables. They are `eve_in` and `eve_out`, which are a restricted version of Linda's `in` and `out` methods [30]. This abstraction hides the majority of the underlying variable management from the programmer while unifying the access method to all shared variables. Release consistency is chosen based on the observation that most statistical values are updated only at the end of a simulation or at relatively coarse progress snapshots. Enforcing stronger consistency would unnecessarily increase overhead.

To allow synchronization, *Eve* uses two other functions `eve_lock` and `eve_unlock`. These provide a simple *mutex* context. A client that tries to lock a variable that is already locked by another client will block until that client variable is unlocked. If more than one client are waiting for a lock, then the highest-priority one will run first. *Eve* supports priority inheritance to prevent priority inversion.

Collecting statistics is simplified by the DSV where measurements can be updated by all partic-

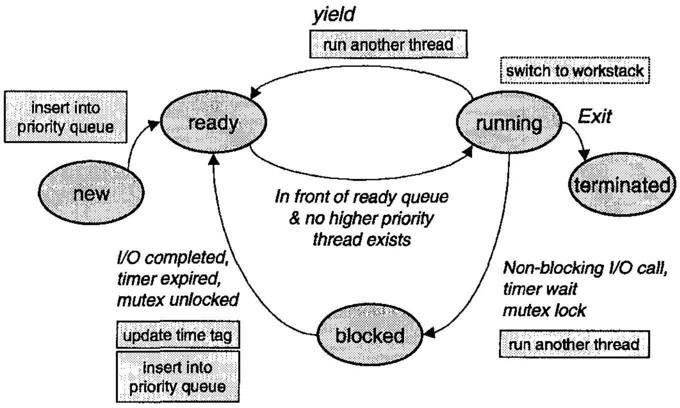

yield
run another thread
switch to workstack
insert into priority queue
ready
running
Exit
terminated
new
In front of ready queue & no higher priority thread exists
I/O completed, timer expired, mutex unlocked
update time tag
insert into priority queue
blocked
Non-blocking I/O call, timer wait mutex lock
run another thread

Figure 3.11: State transition diagram. Transition conditions are represented in italicized text. Transition actions of the scheduler are represented in boxed text.

ipating clients. In our example, we were interested in tracking the number of successful sessions. Therefore, upon completion of a successful session, the client increments the corresponding shared variable before exiting. Additional statistics can also be defined to indicate other variables such as the throughput in number of requests, response time, number of failed connections, etc. What is important, however, is that collecting measurements does not change even in situations where clients are distributed across multiple hosts. In fact, *Eve* successfully hides away most of the complexity of managing distributed clients, scaling clients to stress-test powerful servers, and collecting measurements.

3.5.4 I/O-Threads

The thread abstraction, because of its simplicity, is an ideal solution for emulating a wide variety of client modules. However, one must be careful about the design of the thread architecture when thousands of simultaneous threads must be supported. For example, if clients time out after 30 seconds, to sustain a rate of 200 requests/sec (which can be handled by off-the-shelf web servers), then the worst-case number of needed threads is $200 \times 30 = 6000$ threads, one thread for each outstanding request.

Eve implements a lightweight thread library where each new request or client can be easily handled by a separate thread.[7] With potentially thousands of threads that need to be supported, a natural question is why would our scheme work while others don't? We identify five key design

[7] At the time of creating this multi-threading library, existing user-level libraries did not provide the necessary functionality. Recently, pth [47] has advanced to provide similar functionality.

decisions that enabled us to answer this question:

- **Integrate I/O handling into the design of the thread library to provide greater control over potentially expensive I/O calls:** By transforming blocking I/O calls into non-blocking ones, *Eve* can schedule and execute a different task. Whenever the I/O call is completed, the calling thread is then placed in a ready queue (Figure 3.11). These calls are then a natural place where a thread would yield to another ready thread.

- **Enforce cooperative or non-preemptive multitasking [115] to maximize processor usage time and minimize switch overhead:** Implementing cooperative multitasking was a natural consequence of using non-blocking I/O. By avoiding the use of preemption, we eliminated the need for signals, synchronization (to lock thread control block structures) as well as the need to support reentrant functions. While implementing preemptive multitasking, as opposed to cooperative multitasking, is not difficult, as we will see later, doing so would have prevented other optimizations. One may wonder if cooperative multitasking negatively influences the accuracy of real-time events. While this is true to some extent, usual client models are I/O-heavy, implying a short period between two scheduling operations.

- **Support priority queues to give soft real-time tasks, such as timers, priority over other tasks:** Supporting a large number of threads requires more than just a simple FIFO scheduler, typical of most user-level thread implementations. Since some tasks, especially timeouts, should be handled immediately, priority scheduling was necessary to support these soft real-time requirements [76].

- **Use *time tagging* to timestamp when a thread is ready; this way the thread can compensate for the long wait caused by supporting a large number of threads:** The time delay between two scheduling-slices may be long for a large number of threads. For example, if there are 1000 ready threads and each thread takes 1 msec, then there is an average thread delay of 0.5 sec. This delay can easily degrade accuracy of measurement and is common in most client emulators. To minimize this inaccuracy, *Eve* uses *time tagging* where each thread is tagged with the time when it became ready. A simple call can then be used to retrieve this time and can be easily incorporated into the client code.

- **Minimize resource and memory consumption per thread to support a large number of threads. *Eve* provides a *work stack* for functions that require large stack space.** Memory usage can explode easily when supporting a large number of threads. *Eve* uses a 4KB stack per thread. Smaller stacks can also be used. While we recognize that 4KB may not be enough in some cases, *Eve* introduces a large, shared, *work stack* that can be used during one scheduling-slice of any thread. Since threads are only switched at specific instances, they can safely take advantage of the larger stack space. This is another advantage for cooperative multitasking.

Operation	Cost
Creating a thread	4 usec
Switching between threads	0.63 usec
Average I/O call	25 usec
DSV initialization	340 usec
DSV access (server)	0.62 usec
DSV access (local)	100 usec
DSV access (remote)	210 usec

Table 3.3: Cost of operations in *Eve*

As we show in the next section, the above five features enabled us to maximize efficiency while reducing overhead.

3.5.5 Performance of Eve

We studied two aspects of *Eve*'s performance: basic operation overhead and maximum number of supported clients. We used a simple testbed to study *Eve* in a well-controlled environment. In all testing scenarios a server machine (Intel Pentium-based PC with 1.7 GHz and 512 MBytes memory) was running off-the-shelf Apache 1.3 web server. Two client machines (Pentium-based PCs 600 MHz with 512 MBytes memory) connect to the server through a FastEthernet switch. These machines are used to test the performance of *Eve* in both single and multi-host configurations. Since we only focus on the performance of *Eve*, client requests follow a Poisson-distributed inter-arrival times, allowing our measurements to reach the desired confidence level relatively quickly. Alternatively, using a heavy-tail distribution, such as those proposed in Section 3.4, would require a very long time for measurements to converge [39]. Measurements were repeated until the estimated error was under 5% (with 95% confidence). Also, all requests were made to a single static file to maximize the throughput of the server.

3.5.6 Operation Overhead

Table 3.3 shows the basic overhead for common *Eve* operations. As shown in the table, the cost of both creating and switching between threads is very small, thus enabling support of a large number of simultaneous threads. I/O operation, however, requires more system resources partly because of our implementation (i.e., using `select`). However, even in such a case, it is relatively

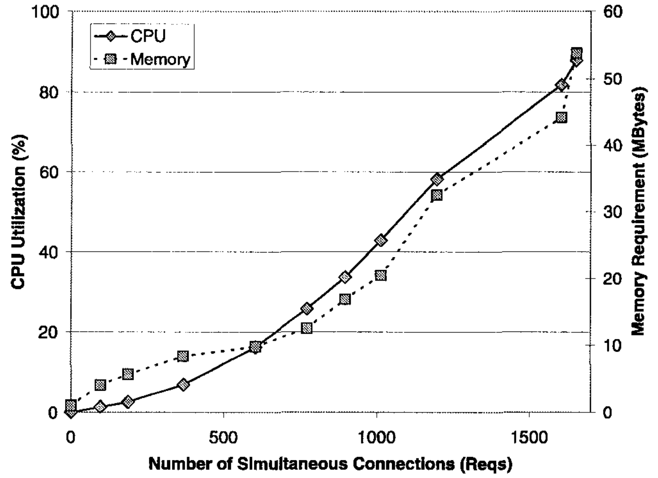


Figure 3.12: Client Load

easy to support 1000 threads.

Accessing shared variables is the most expensive operation because of our use of socket streams. While other implementations, such as local IPC calls, may reduce the overhead for local DSV accesses, it can not dramatically improve the DSV's performance in multi-host configurations. The need for strong consistency is the primary reason behind this overhead. For a multi-host configuration, data updates must be performed at the central server, which unfortunately also reduces the benefits of automatic data caching. We therefore recommend designing clients to limit DSV access to the beginning and end of an experiment and minimally access the DSV during the experiment.

3.5.7 Client Support

We measured the resource requirements in terms of CPU utilization and memory requirements as the number of simultaneous clients is increased. These measurements were collected from the `/proc` file system in Linux. Figure 3.12 shows the resource requirements on a single machine (Pentium 600 MHz with 512 MBytes of RAM). The figure shows two important points:

- The resource requirements grow linearly with the number of simultaneous connections. A linear fit showed the cost of a single connection as 0.05% of the CPU and 29 KBytes of memory. Since the CPU was the bottleneck in our test scenario, we did not evaluate *Eve*'s behavior when memory starts thrashing.
- The number of simultaneous connections is estimated as *offered load* × *average response time*. Therefore, the offered load is inversely proportional to the server's response time. For exam-

ple, if requests have an average response time of 8 sec, then only 200 reqs/sec can be sustained by *Eve*.

Unfortunately, *Eve* does not provide an automatic method for detecting the maximum number of simultaneous clients that a host is able to support. This is basically done using a trial-and-error technique. However, since the design of *Eve* easily scales to multiple hosts, the process of increasing the number of simultaneous requests is relatively straightforward: starting daemon process on remote hosts and changing a single configuration file. Neither the client models nor data measurements and collection needs to be changed.

3.6 Conclusions

We characterized the dynamics of persistent clients in aggregate traffic. In particular, we showed that client's persistence, which is due mostly to TCP's congestion control, has a direct effect on the stability and effectiveness of traffic control mechanisms. We also showed a discovery of re-synchronization of dropped requests. Based on our detailed analysis, we constructed an accurate model of client's persistence and also an efficient tool, *Eve*, to accurately generate such clients for performance analysis. The presented model and client generation tool are the bases for our proposed optimizations and evaluations in the proceeding chapters.

CHAPTER 4

PERSISTENT DROPPING:

An Efficient Control of Traffic Aggregates

The development of persistent clients in Chapter 3 provided an improved understanding of the internal dynamics and the corresponding aggregate traffic of typical Internet clients. This chapter undertakes an analytical approach to improve the way that requests originating from such clients are controlled. The analytical development gives rise to a low-level admission control scheme, *persistent dropping*, that is tailored to the intrinsic behavior of persistent clients. Two efficient realizations of this new scheme are also presented with thorough evaluation of its efficacy.

4.1 Introduction

Client persistence imposes an added challenge to the controllability of aggregate traffic. In this model, clients do not simply go away if their requests are not served. Instead, they keep trying over and over until they either succeed or get fed up. We observe this behavior in the way TCP handles packet loss, i.e., an unacknowledged packet will be repeatedly sent after exponentially increasing timeout periods. This persistence is further increased when clients send their requests in parallel. In Chapter 3, we showed that client persistence is an important problem when controlling high-load situations, such as flash crowd events. If a router or end-server is operating near or at full capacity, then any slight increase in load will trigger dropping of requests. These persistent requests, upon their retransmission, will set off further drops, creating a vicious cycle of drops causing future drops. Repeated dropping also dramatically increases the client-perceived latency as it may require several timeouts before a client successfully establishes a connection. A system that is trapped in such a cycle will require a long time to recover after the load subsides.

Unfortunately, existing rate-based techniques are ill-suited for controlling connection requests. In particular, we observed, that the reaction of the underlying traffic to a rate-limiting policy can, and often will, reduce the effectiveness of the applied control. One way of explaining this is by decoupling aggregate traffic into two elements. The first element describes how existing or ongoing connections react to the applied controls; the second element describes how the arrival of new connections are affected by the applied control. We found that the combination of TCP's reaction to packet loss (first element)—namely, retransmitting after timing out—with the arrival of connections from new clients (second element) has an additive effect that is not accounted for by current traffic controllers. To improve the controllability of aggregate traffic, we advocate, in this chapter, classifying incoming connection requests into new SYN packets and retransmitted SYN packets, then applying specialized controls to each traffic class—a similar concept to [122]. We show that this approach can be used not only to improve the controllability of aggregate traffic, but also to minimize the response time of client requests (i.e., TCP connections) during overload while maintaining fairness to all connection requests.

Through the specialization of control, we are able to focus on the persistence of clients accessing the server. We propose *persistent dropping (PD)*, an effective control mechanism, which minimizes the client-perceived latency as well as minimizes the effective aggregate traffic (includes new and retransmitted connection requests) while maintaining the same control targets as a regular rate-control policy. PD randomly chooses a number of requests based on a target reduction in the effective aggregate traffic arrival rate and systematically drop them on every retransmission. PD is well suited for controlling aggregate traffic as it achieves three goals: (1) it enables routers and end-servers to quickly converge to their control targets, (2) it minimizes the portion of client delay that is attributed to aggregate control by Internet routers and end-servers while maintaining fairness to all packets, and (3) it is both easily implementable and computationally tractable. We emphasize that PD complements, but does not replace, existing control mechanisms that are optimized for controlling already-established TCP connections [32, 56]. We also emphasize that PD does not interfere with end-to-end admission-control policies as it represents an optimization of existing queue management techniques.

This chapter is organized as follows. We first search for the optimal drop policy and propose a PD controller to deal with persistent clients in Section 4.2. We present different implementation of PD in Section 4.3 that emphasize performance, robustness, and accuracy. In Section 4.4, we experimentally evaluate some performance issues. The chapter ends with concluding remarks in

Sections 4.5.

4.2 Persistent Dropping

Consider an Active Queue Management (AQM) technique that drops incoming SYN packets with probability p. Here, we do not consider how other packets are treated, and p is set in accordance with the underlying AQM technique. For instance, if packets are dropped in routers using RED, then p is based on dynamic measurements of queue lengths [56]. We, thus, view p as the percentage of packets that must be dropped regardless of how it is chosen. Given a target drop probability p (or equivalently an effective timeout probability, p^*, as described earlier), our goal is to find the optimal drop policy that minimizes the effective arrival rate, Λ, and connection-establishment latency, EL_h. We base our development on the same network model introduced in Section 3.2, and still do not consider the parallelism of individual clients. This is addressed in Section 4.3.

The optimality is established by first looking at the retransmission epochs of individual connections. As shown in Section 3.2, given packet-loss probability p, estimates for connection-establishment latency and effective arrival rate can be derived. We use these relationships as bases to show that a drop policy that consistently drops retransmitted requests is able to minimize the additional latency that is caused by the applied control as well as minimize the aggregate traffic of all incoming requests.

Traditionally, a control policy that drops aggregate traffic with probability p does not take into account the transmission class of individual connections. Consider here a different mechanism that associates a drop probability p_i with each transmission class i. In order to assign these probabilities we assume that incoming requests are classified into their corresponding transmission classes; we show later how this can be achieved. Let us rewrite the aggregate arrival rate, Λ, in Eq. (3.2) using the per-class drop probabilities p_i's:

$$\Lambda = \lambda_0 + p_0\lambda_0 + p_0p_1\lambda_0 + \ldots + \left(\prod_{i=0}^{n-1} p_i\right)\lambda_0. \tag{4.1}$$

Notice here that the effective timeout probability is $p^* = \prod_{i=0}^{n} p_i$. For a traditional rate control policy, all requests are dropped with an equal probability (or $p_i = p$ for all i), which implies that $p^* = p^{n+1}$ (consistent with the results in Section 3.2.1).

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

$$P\{L_h(x) = t\} = \begin{cases} (1 - p_0) & \text{if } t = T_0 + RTT \\ \prod_{j=0}^{i-1} p_j(1 - p_i) & \text{if } t = T_i + RTT, 1 \leq i \leq n \\ \prod_{j=0}^{n} p_j & \text{if } t = T_{abort} \\ 0 & \text{otherwise,} \end{cases} \tag{4.2}$$

where T_i is the time of the i^{th} retransmission, $T_0 = 0$ is the time of the initial transmission, and T_{abort} is the time before the connection times out. Intuitively, Eq. (4.2) establishes the probability of connecting successfully after $RTT + T_2$ seconds, for example, is the probability of being dropped during the initial transmission with probability p_0, then being dropped again on the second transmission with probability p_1, and finally connecting on the third transmission attempt with probability $(1 - p_2)$. Notice that the minimum connection-establishment latency is $t = RTT$. Based on this, the expected connection-establishment delay, EL_h, can be computed as:

$$\begin{aligned} EL_h &= (1 - p_0 \cdots p_n)RTT + \\ & \quad p_0(1 - p_1)T_1 + p_0p_1(1 - p_2)T_2 + \ldots + \\ & \quad (p_0 \cdots p_{n-1})(1 - p_n)T_n + (p_0 \cdots p_n)T_{abort}. \end{aligned} \tag{4.3}$$

The optimal drop strategy must, thus, minimize EL_h with the constraint of having an effective timeout probability (see Section 3.3.1) that is equal to the one obtained by traditional policies, i.e., $\prod_{i=0}^{n} p_i = p^*$. It suffices to show that if we set $p_0 = p^*$ and $p_i = 1$, for $i \neq 0$, then EL_h is minimized. This can be seen by observing that each term in Eq. (4.3) cancels out except for the last term. The minimum connection-establishment latency is then $EL_h^{g^*} = (1 - p^*)RTT + p^*T_{abort}$, where g^* denotes our optimal policy. Note that since $EL_h^{g^*}$ no longer has a delay component for successful connections, g^* breaks the dependency between the delay for successful connections and packet drop.

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

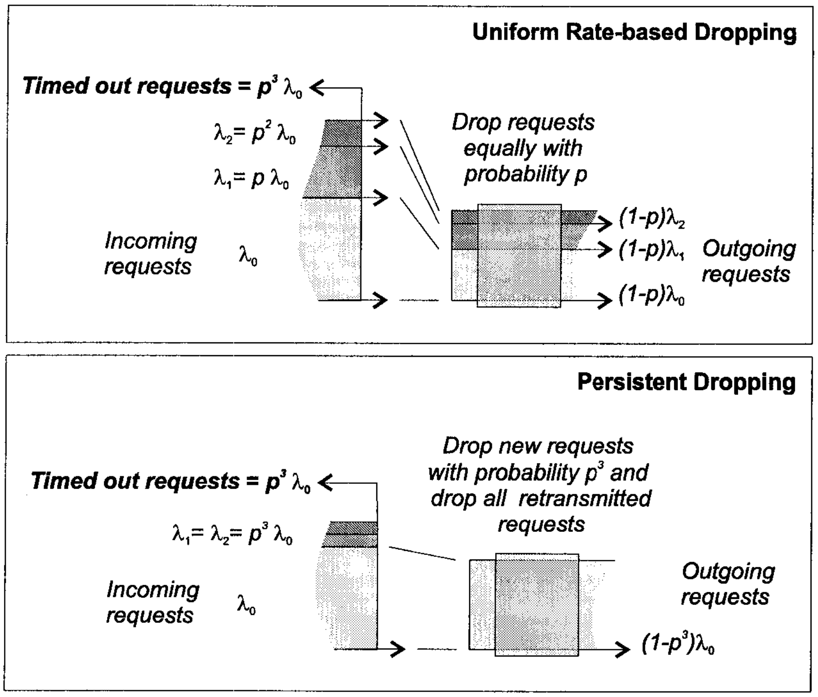

Uniform Rate-based Dropping
Timed out requests = $p^3 \lambda_0$
$\lambda_2 = p^2 \lambda_0$
$\lambda_1 = p \lambda_0$
Incoming requests
λ_0
Drop requests equally with probability p
$(1-p)\lambda_2$
$(1-p)\lambda_1$
Outgoing requests
$(1-p)\lambda_0$
Persistent Dropping
Drop new requests with probability p^3 and drop all retransmitted requests
Timed out requests = $p^3 \lambda_0$
$\lambda_1 = \lambda_2 = p^3 \lambda_0$
Incoming requests
λ_0
Outgoing requests
$(1-p^3)\lambda_0$

Figure 4.1: Illustrative comparison between rate-based dropping and PD. We view the outgoing link as a smaller pipe than the stream of incoming requests. We then show how the two strategies drop incoming requests to fit them into the pipe.

sender's cooperation. This issue is addressed in Chapter 5.

Based on the above discussion, we introduce *persistent dropping* (PD) as the optimal drop strategy that chooses $p^*\lambda_0$ new requests and systematically drop them on every retransmission. An example of PD is illustrated in Figure 4.1, where it shows how this new technique intelligently fills the outgoing link to minimize packet retransmissions. With persistent dropping, the resulting (or the aggregate) drop probability is substantially lower than traditional rate-based techniques. Specifically,

$$\begin{aligned} p^{g^*} &= \frac{p_0\lambda_0 + p_1\lambda_1 + \ldots + p_n\lambda_n}{\lambda_0 + \lambda_1 + \ldots + \lambda_n} = \frac{p_0\lambda_0 + p_0p_1\lambda_0 + \ldots + \left(\prod_{i=0}^{n} p_i\right)\lambda_0}{\lambda_0 + p_0\lambda_0 + \ldots + (p_0p_1\cdots p_{n-1})\lambda_0} \\ &= \frac{(n+1)p^*}{1+np^*}. \end{aligned} \tag{4.4}$$

The fact that this dropping scheme chooses certain connections and consistently drop them on each retransmission does not imply that it is biased against these connections. We have shown that a rate-control drop policy generally causes $p^*\lambda_0 = p^{n+1}\lambda_0$ connections to time out, which is identical (but less efficient) that PD. In this respect, both a PD policy and rate-control policy have the same fairness. Table 4.1 compares the performance improvement of PD over a traditional rate-control

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

Table 4.1: Comparison between PD and random dropping.

policy in terms of mean client-perceived latency, average number of retransmissions, and aggregate arrival rate for the same effective timeout probability. In Section 4.4, we also compare the variance in the latency, L_h, of the two schemes—they can be directly computed using Eq. (4.2).

4.2.1 Applicability to Network of Queues

In most cases, requests must pass through multiple queues as they traverse different links on the network before reaching their destination. Fortunately, the above results also hold in this scenario, namely, when requests pass through a network of queues in series, each using a persistent drop policy g^*, the client's establishment latency and effective arrival rates are minimized. This is illustrated in Figure 4.2 where we assumed for simplicity that all queues have the same drop probability p. We see that for a rate-based drop strategy, the probability of a single request succeeding on a single attempt is $(1-p)^m$, where m is the number of queues that it must pass through. In contrast, the PD policy g^* has a probability $(1-p^{n+1})^m$. To put this in perspective, if $m = 5, p = 0.05$, and $n = 4$, then the probability of a request succeeding is 0.77 and 0.99 for the uniform rate-based and the PD policy, respectively. Using a similar development to our single queue analysis, we can prove that g^* is the optimal drop strategy even when each queue uses a different drop probability.

4.2.2 Applicability to Persistent Clients

The development in the previous sections treated clients' connections as independent entities without considering the correlation between a group of connections originating from the same client (e.g., client visits or sessions as defined in Section 3.4). It is not difficult to verify PD's optimality in the case of correlated connections. Under the assumption that the controller does not distinguish

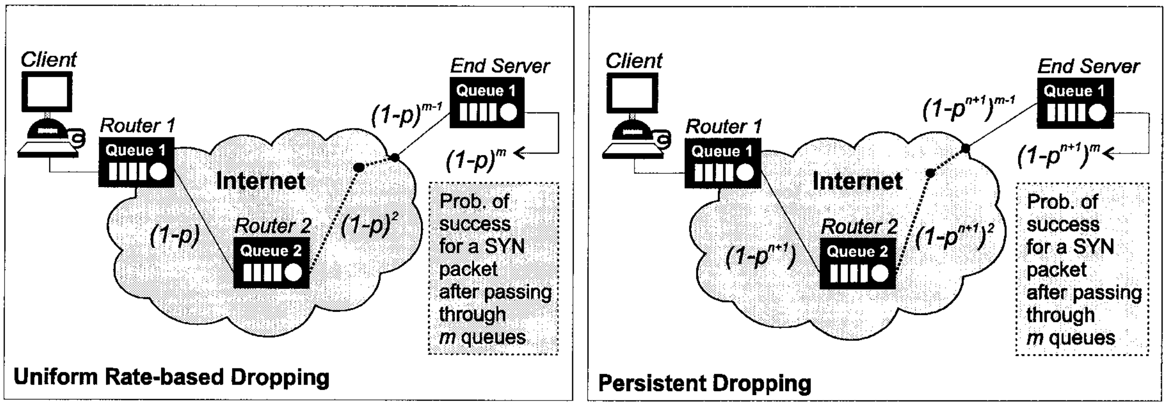


Figure 4.2: Probability of success in network of queues

between SYN packets that belong to an already admitted visit and those that represent new visit, we provide here an intuitive sketch of the proof. Consider $EL_s = E[L_h|$ connection succeeds$]$ as the conditional expectation of the connection-establishment latency for successful connections. This is equivalent to Eq. (4.3), but excludes the last term and divides by $(1 - p^*)$ to compensate for unaccounted timed-out connections

$$EL_s = RTT + \frac{p_0(1-p_1)T_1 + \ldots + (p_0 \cdots p_{n-1})(1-p_n)T_n}{1-p^*}. \qquad (4.5)$$

Assume that once a connection is established, the average time to send the request, get it processed by the server, and receive the reply is ET_s. Estimates for ET_s are derived in [29, 97] as part of determining the expected latency of a TCP connection. We can now substitute the expression of EL_s into Eq. (3.5) using the relationship $ET_c = EL_s + ET_s$ to obtain EV as a function of per-class drop probabilities. Similar to the development in Section 3.4, when $p_0 = p^*$ and $p_i = 1$ for $i \neq 0$, EV is minimized.

4.3 Designing a Persistent Dropping Queue Manager

In the previous section, we have showed the optimality conditions of an admission-like control policy that can be implemented at the TCP level protocol to minimize the portion of the client's delay that is due to queue management in routers or end-servers. Implementation of our optimal drop policy in routers and end-servers relies on the ability to (1) group requests originating from the same client visit and (2) distinguish between new and retransmitted requests. Unfortunately, both present a design challenge, especially since we intend for our technique to operate at the packet-level. In fact, precise implementation requires violation of the protocol layers, similar to

Layer-7 switches (e.g., Foundry, Alteon) to satisfy requirement (1) and need per-connection state information to satisfy requirement (2). However, one must not forget the original environment that this is intended for: large aggregate traffic causing an FCE. We are, thus, interested in constructing approximate implementations that are allowed to be less accurate than an exact implementation, but significantly improves on existing techniques.

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

The selection of a suitable hash function, $h(.)$, is not difficult. In fact, as we show shortly, a simple XOR operation on the input parameters produces the desired uniform hashing [38]. On the other hand, we found that choice of the input parameters to the hash function is the most critical element in our design. Unfortunately, without client-side cooperation, packet-level information provides limited choices in achieving the desired classification. They are summarized as follows. We abbreviate IP source and destination addresses and TCP source and destination ports with src_addr, $dest\_addr$, src_port, and $dest\_port$, respectively.

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

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

H3. $h(src\_addr, dest\_addr, IP\ options)$: One alternative solution is to require clients to encode

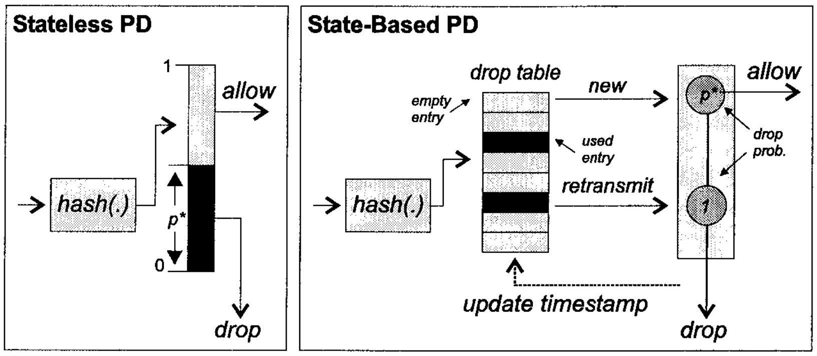


Figure 4.3: Stateless and state-based implementation of persistent drop controller.

their session information using IP options. This will produce the most accurate classification. It is, however, impractical as it requires client stack modification as well as high router overhead to process IP options. We will not investigate this alternative any further.

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

$$h(x_1, x_2, \ldots, x_k) = x_1 \oplus x_2 \oplus \ldots \oplus x_k \times K(t) \bmod R, \tag{4.6}$$

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

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

4.3.1 Stateless Persistent Dropping (SLPD)

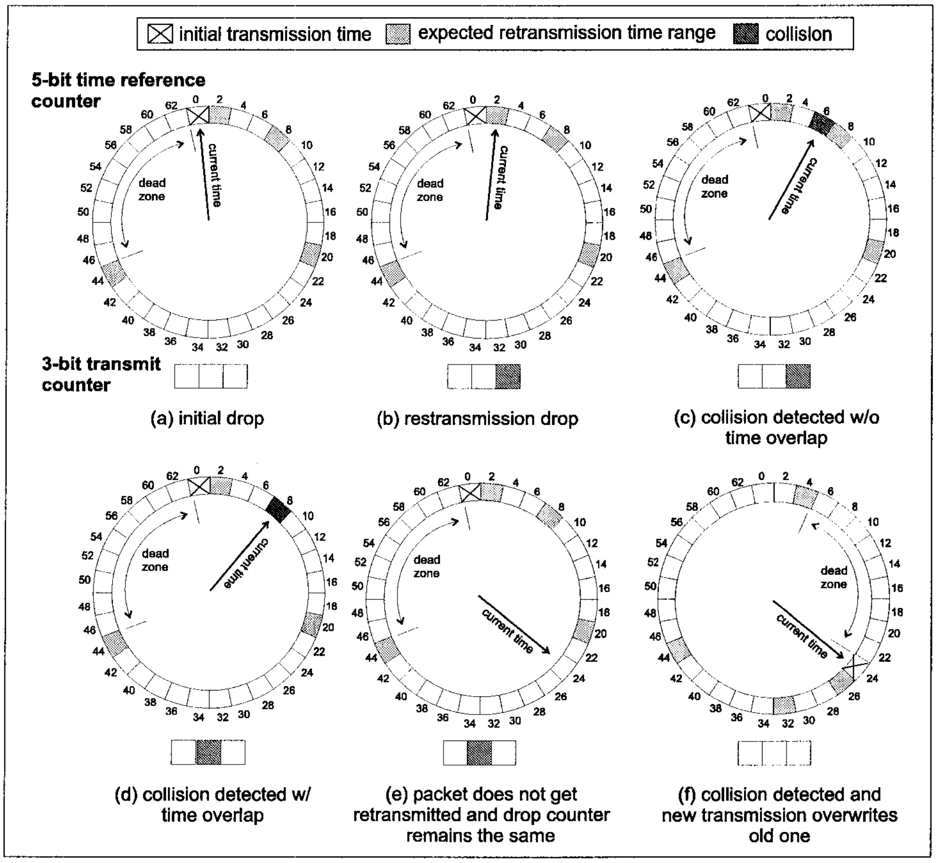
Upon arrival of a new connection, the hash in H1 or H2 is computed and normalized to a number within the range [0,1]. A threshold value, represented by the effective timeout probability, p^*, is used to drop those packets that have a hash value less than p^* and allow the rest to pass through (Figure 4.3). Depending on whether H1 or H2 is used, client- or connection-level persistent dropping can be achieved, respectively. The absence of state makes this scheme very simple to implement and fast to execute. However, this scheme can be unfair as it discriminates against a fixed set of clients. To mitigate this problem we use the term $K(t)$ in Eq. (4.6) to periodically change the function's mapping, hence its dependence on t [21]. The time interval between changes should be on the order of several minutes to minimize the error introduced by changing the set of dropped packets.

4.3.2 State-based Persistent Dropping (SBPD)

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

The basic operation of SBPD is split into two stages (Figure 4.3). The first stage consults the table to see if the request is a new or a retransmitted one. A table entry stores the time of the first drop time. Therefore, any incoming request that is mapped to a used entry is systematically marked as "retransmission" for a T_{abort}^{max} second window from the initial drop time. The window length is chosen based on the timeout value among most OS implementations. If the entry is empty or has an expired time-stamp, the request is marked as "new." The second stage of SBPD decides the control policy. Obviously, a request that is marked as "retransmission" is dropped. However, one that is marked as "new" is dropped with probability p^* and the hash table is appropriately updated.

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


☒ initial transmission time
▨ expected retransmission time range
■ collision
5-bit time reference counter
dead zone
current time
3-bit transmit counter

(a) initial drop

(b) restransmission drop

(c) collision detected w/o time overlap

(d) collision detected w/ time overlap

(e) packet does not get retransmitted and drop counter remains the same

(f) collision detected and new transmission overwrites old one

Figure 4.4: Example of request classification based on the time reference and transmit counters.

scenario for which M should be provisioned. Assuming that our hash function is truly uniform, at most λ_0 requests may need to be dropped (and stored) per second.[1] But requests must be tracked for at least T_{abort}^{max}, so the table size is computed as:

$$M = \lambda_0 \times T_{abort}^{max} \times \sigma, \tag{4.7}$$

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

For individual table entries, we identify three criteria for encoding each entry in our hash table. First, the time or equivalently the age of a dropped request must be stored to identify its corresponding transmission class. Second, hashing collisions must be detected upon their occurrence, to

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

```
COMPUTE HASH        \\Assume the destination IP address and TCP port number are in IP_Addr and TCP_port.
    IP_upper ← ( IP_address & xFFFF0000 ) >> 16
    IP_lower ← ( IP_address & 0x0000FFFF )  \\ Compute the lower and upper 16 bits of the IP address.
    hash ← TCP_port ⊕ IP_lower ⊕ IP_upper  \\ where ( & is bitwise-and), ( >> is shift-right), and ( ⊕ is bitwise-xor)

CLASSIFY   \\ Classify into four states: new transmission, retranmission, undetectable collision, and detectable collision.
    index ← Compute Hash of incoming packet
    entry ← Hash_Table[index]                          \\ Retrieve entry corresponding to index from table.
    time_ref ← lower 5-bits of entry                   \\ Extract time reference and xmit counters
    xmit_cnt ← next 3-bits of entry
    xmit_class ← min { x : $T_{x+1}$ > current_time - time_ref }   \\ Determine retransmission class of packet, -1 if none.
    if xmit_class = 0 or xmit_cnt < xmit_class – 1
        state ← new transmission
    if xmit_cnt = xmit_class – 1 and $T_x$ ≈ current_time - time_ref
        xmit_cnt ← xmit_cnt + 1                        \\ We also write it back to the hash table
        state ← retransmission
    if xmit_cnt = xmit_class and $T_x$ ≈ current_time - time_ref
        state ← undetectable collision
    if $T_x$ < current_time - time_ref
        state ← detectable collision
```

Figure 4.5: Hashing and classification algorithms. The algorithm presented ignores the counter wrap-around issues. So, $current\_time - time\_ref$ should always be assumed to be non-negative.

maintain accurate classification of incoming requests. Third, the size of each entry must be limited (e.g., to 8 bits) to minimize memory requirements. To meet these requirements, we observe that almost all requests will time out within 5 retransmission attempts ($n = 4$), including the initial transmission (Table 3.1). This implies that the reference time counter should cover a range of 45 seconds to properly classify all transmissions in that range. Since retransmissions are on the order of multiple seconds, a two-second resolution is sufficient; it also accounts for the slight variations in transmission times.[2] With this in mind, only 5 bits are required to encode the reference time counter, which covers a range of 0 to $2 \times 2^5 - 1 = 63$ seconds. To detect collisions, we use three bits to account for the number of transmission attempts.

The basic process of classifying incoming requests are exemplified in Figure 4.4 and formally defined in the CLASSIFY function in Figure 4.5. Let $t = t_0$ be the arrival time of a *new* connection request that was dropped and hashed into a hash table entry. Assume that the hash entry was initially unoccupied. Logically, our time reference is a circular counter, and thus, can be represented by a

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

time dial in Figure 4.4(a). All time values on the dial represent time ranges with respect to t_0, where the "X" in the figure marks arrival (and also drop) time of this connection request and the shaded boxes indicate the time periods for its expected retransmissions within a 2-second range ($[T_x - 1, T_x + 1]$). Because this is the first transmission, the transmission counter is cleared. Consider now $t = t_0 + 3$, the arrival time of the request's retransmission (Figure 4.4(b)). Since it will arrive during the first shaded box, it will be classified as a retransmission of the original request; the transmission counter is incremented accordingly. This will repeat until the connection times out.

Classification conflicts may arise when multiple requests are hashed to the same entry. Consider a second request arriving during a non-shaded period (Figure 4.4(c)). We call this a "decidable collision" since—with high probability—this request does not correspond to the original one. On the other hand, if a second request arrives during a shaded period (Figure 4.4(d)) and the transmission counter has already been incremented to reflect that a retransmission has been seen in this period, then this request can correspond to the actual retransmission or some new transmission; we, thus, classify it as an "undecidable collision."

When a collision is detected, a proper action must be taken to maintain proper (future) classification. As mentioned earlier, there are two types of collisions: decidable and undecidable. All undecidable collisions are classified as a retransmission and dropped, because an arriving request during a shaded-area will either correspond to the original request or to a new request, and such requests are indistinguishable from each other. Fortunately, new requests that are inappropriately dropped will be retransmitted during a non-shaded area, which are then classified as decidable collisions. When a decidable collision is detected, we interpret it as new transmission. Two possibilities exist in this case. First, upon arrival during a non-shaded area, if the transmission counter was not incremented during the most recent shaded area (Figure 4.4(e)) or if the counter equals 4 (i.e., its maximum value), then this means that the original request has either aborted the connection attempt or timed out, respectively. The entry can then be updated to reflect the new request (Figure 4.4(f)). Second, upon arrival during a non-shaded area, if the transmission counter was appropriately incremented during the most recent shaded area, then we are almost certain that the request is a new transmission, but cannot be stored in the entry. We, thus, let it pass through the filter. In general, because of the low collision probability, letting "decidable collisions" pass through will not affect the target drop probability or the corresponding delay.

When the hash table is used beyond its design range, the above classification technique can yield too many errors. To protect against such erroneous behavior, we use dynamic monitoring to detect and take corrective actions. Basically, the real drop probability is measured on-line by counting the total number of arrivals and dropped requests. If the measured drop probability is dramatically different from the aggregate drop probability in Eq. 4.4, then a uniform drop probability is used with $p = p^*$ for all incoming requests. This is a fall-back behavior, which is only used in extreme cases.

Finally, periodic maintenance of the hash table entries is required. This is equivalent to garbage-collection where old entries are cleared before the reference timer roles around. During this process, all hash entries are examined for expired values as follows. If the transmission counter does not correspond to the appropriate transmission class at the time of the maintenance (similar to the case in Figure 4.4(e)), then that entry is cleared.[3] Only once every 16 seconds all entries must be checked. This requirement can be verified by observing that at least once for every time the counters role over (64 seconds) we need to check during the "dead zone" (Figure 4.4) of every entry if it has expired. Note that if longer inter-maintenance periods are required, then more bits are needed to encode the reference timer to increase the length of the "dead zone".

4.3.3 Linux Implementation

We implemented working prototypes of SLPD and SBPD in Linux (Kernel 2.4) as filter extensions to `iptables`, Linux's firewalling architecture [86]. Using `iptables`, our implementation can be configured as part of the routing path, when our Linux box is configured as a router, or as a front-end, when it is configured as a regular server.[4] In `iptables`, packets are filtered based on user-defined rules. Typically, a rule may include IP and TCP header information such as source or destination addresses/networks, ToS bits, SYN or RST flags, or TCP source or destination ports. A target function is also associated with each rule; it specifies what should be done to packets that match the corresponding rule. Typical targets include `ACCEPT` to accept the packet and `DROP` to drop the packet. Therefore, when an incoming packet matches a rule, the associated target function is invoked. For example, one can define a rule that matches all packets with the SYN flag set (indicating a new connection request) with a target of `DROP`. This would effectively block any connection attempt to the protected machine.

The architecture of `iptables` is designed to be easily extensible where the target function can be written as a separate kernel module and is free to implement any packet enforcement behavior. We defined two new targets in `iptables` called `SLPD_Filt` and `SBPD_Filt` that are kernel

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

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

modules. These targets have a configurable effective timeout probability, p^*, and hash function, H1 or H2, that can be altered at runtime. Their implementation follows the exact description in this section. To activate either filter, we define a new rule that matches any packet with the SYN flag set and associate either module as its target. This way, new connection requests are dropped according to our optimal drop policy. As mentioned, our implementation dynamically monitors the real drop probability. If the number does not match the expected value, incoming requests are dropped with probability p.

4.4 Evaluation

To evaluate and demonstrate the efficacy of PD, we equipped a Linux server machine with working implementations of the SLPD and SBPD controllers (Section 4.3.3) as well as a rate-based drop (RBD) controller. The latter mimics traditional mechanisms where it uniformly drops all incoming requests with probability p and is used as the baseline for comparison [86]. Our main goal is to subject these controllers to realistic load conditions so that the results we obtain may be applicable to real-world deployment scenarios. We also want to avoid any unnecessary complexity without sacrificing accuracy. The three controllers are compared by studying their effects on the performance of clients during a synthesized high-load scenarios (such as FCEs), which is emulated by generating high client arrival rates to a web server. In each scenario, we also compare the measured results with the predicted ones from our analytic models.

4.4.1 Experimental Setup

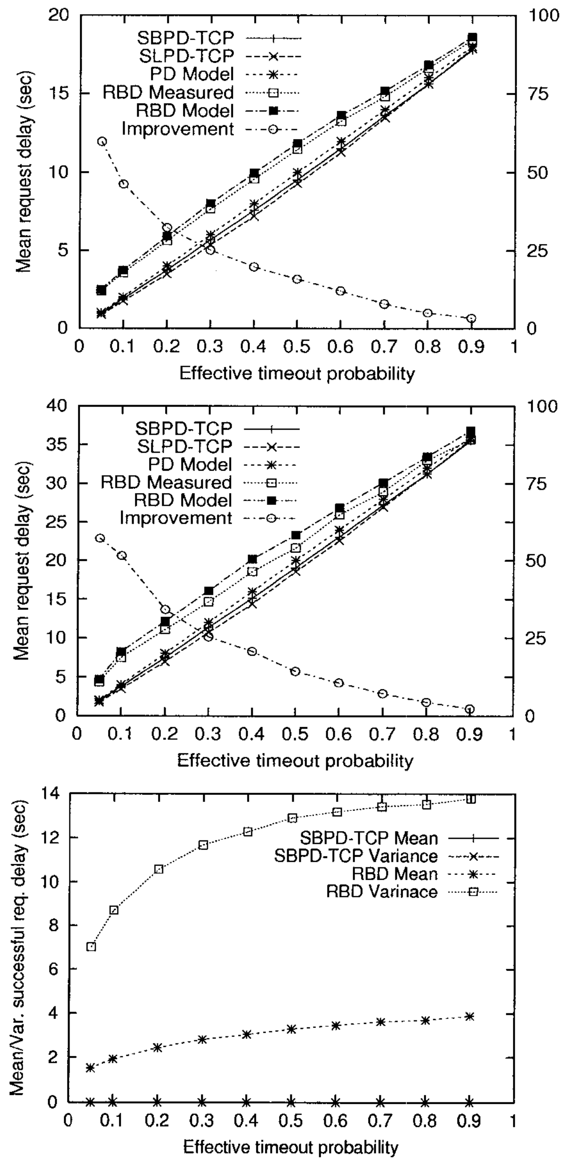
We employ a simple setup where the server machine (a 2.24 GHz Pentium 4 with 1 GBytes of RDRAM) runs Apache 1.3 to receive HTTP requests through a high-speed FastEthernet link. Clients on the other side are generated using Eve, a scalable highly-optimized client emulator. Each of our emulated clients was based on the model described in Section 3.4, where the distributions for the number of parallel connections and their inter-arrival times were based on our estimates for IE 6.0 in Table 3.2. Furthermore, we used IP aliasing to provide each client with a unique IP address, which is necessary for the H1 hashing metric. The arrival of clients (not their requests) followed a Poisson process with mean λ_0, a traditionally-accepted model. Furthermore, each client behaved independently from other clients and, on average, issued 6 (independent) parallel requests. Up to four (500 MHz Pentium III with 512 MBytes of SDRAM) machines were used to generate the desired client arrivals. Finally, an intermediate Linux machine was used as a router to implement one of the three controllers.

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

Our focus in this section is to evaluate the efficacy of PD at the request level and user-visit level based on the H1 and H2 metrics in Section 4.3, respectively, and also to compare stateless and state-based implementations, SLPD and SBPD, respectively. Since PD is intended as a low-level control mechanism (and also due to space considerations), we provide a limited discussion regarding higher-level semantics such as user-sessions. As previously noted, PD is not intended to replace high-level admission control mechanisms, but to improve the control of aggregate traffic in routers, especially during overload.

4.4.2 Connection-Level Measurements

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



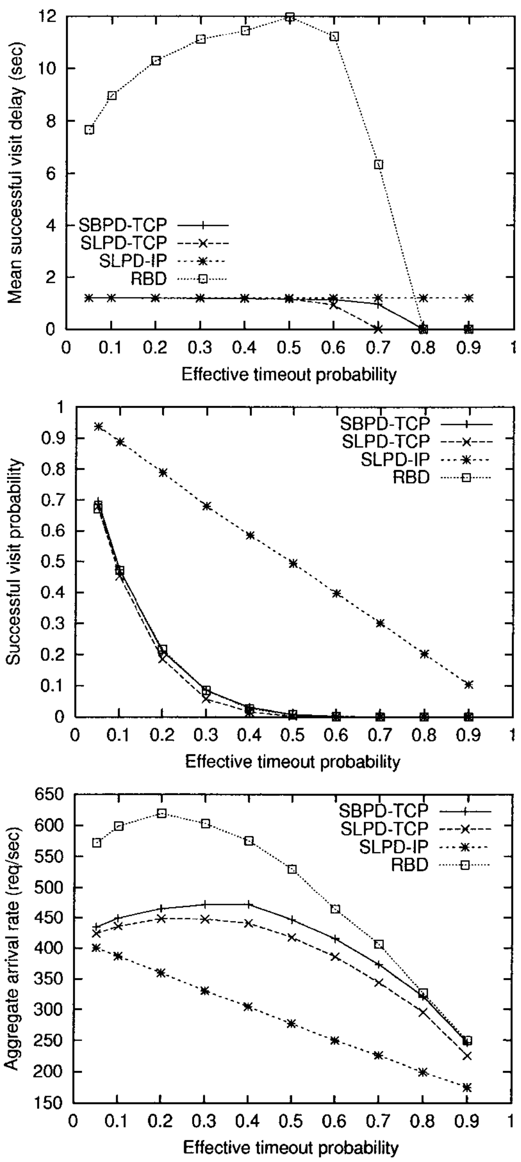


Figure 4.6: Request delay comparison, (top) Delay for λ_0 = 60 clients/sec and T_{abort} = 20 sec, (center) Delay for λ_0 = 80 clients/sec and T_{abort} = 40 sec, (bottom) Mean and variance for the delay of successful requests (same configuration as left).

Figure 4.7: User-visit behavior. In all cases, λ_0 = 60 clients/sec and T_{abort} = 20 sec, (top) mean successful visit delay (a point with zero value implies that no visit was successful), (center) probability of successful visit, (bottom) effective arrival rate.

results were consistent with those presented here.

Two metrics are of particular interest to us: (1) the *mean request delay*, which is computed by averaging the elapsed time before a request is completed or timed out, (2) the *mean and variance of successful-request delay*, which is similar to the first metric but only looks at successful requests; it also looks at the variance of the delay. Figures 4.6(left) and (center) show the benefits of using PD. In the center plot, for example, clients experiencing an effective timeout probability of 0.1 had about a 50% reduction in their mean request delay (due to the reduction in the mean connection-establishment latency) when SLPD-TCP or SBPD-TCP, instead of RBD, is used. This is a dramatic reduction as it implies that a traffic controller that uses RBD to uniformly drop incoming requests with a probability of 0.56, achieves an effective timeout probability of 0.1 and produces 100% longer connection-establishment delays than the one that uses PD (SLPD-TCP or SBPD-TCP). In Figure 4.6(right) we plotted the delay and variance for successful connections only. The figure shows the main benefit of PD, namely, decoupling the effects of the control policy on the delay of successful requests. The greatest impact can be seen on the variance of successful requests since PD produces one of two outcomes: (1) immediately allow a connection to pass through or (2) consistently drop it. We also observed that PD reduced the variability of the underlying aggregate traffic.

Figure 4.6 shows that SLPD-TCP achieves similar performance to SBPD-TCP. The real difference between the two schemes is, however, fairness, which is not reflected in our performance metrics. In SLPD-TCP, packets are dropped based on their header information and the only randomness in the scheme is introduced by the prime multiplier, $K(t)$, in Eq. (4.6). On the other hand, SBPD-TCP has a built-in randomness in every packet it chooses to consistently drop. This, in our opinion, produces better fairness from the client's viewpoint.

We also verified the accuracy of our analytic models. We observe larger, but tolerable, errors in our estimates for smaller values of p^*. However, as p^* increases, T_{abort} dominates the computation of EL_h and thus, improves the accuracy of our prediction. Based on the presented results, our model still accurately predicts the expected delay, even though incoming requests are highly dependent, which seems counter-intuitive. This phenomenon is due to the strict enforcement of the effective timeout probability. Specifically, regardless of the instantaneous arrival rate, a fixed percentage of requests is dropped. Looking back at how the expected delay, EL_h, was derived (Section 4.2), one can observe that once the p_i's are held constant, the delay value becomes independent of the arrival rate. In fact, this type of policy enforcement is implemented by most Active Queue Management

(AQM) techniques where a constant drop probability is enforced based on the average (not instantaneous) length of the underlying queues [56]. Furthermore, the effects of dependent traffic are apparent in other metrics, such as mean user-visit delay and probability of a successful visit (to be discussed shortly).

4.4.3 User-Visit Behavior

While the mean request delay provides a good indication of the performance of the underlying drop policy, it does not give a complete picture. Looking at the performance metrics that are associated with user-visits and the corresponding aggregate traffic better reflects what a typical client experiences in real systems. They also show the effects of dependent traffic more clearly than looking at individual requests by themselves. In the context of user-visits, we use three metrics to compare the performance of the drop polices: (1) the *mean successful visit delay*, which measures the cumulative time for a successful visit as described in Eq. (3.5), excluding the aborted visits, (2) the *probability of a successful visit*, which reflects the sensitivity of dependent traffic to packet drops, and (3) the *effective arrival rate*, which looks at the change in arrival rate as the drop probability is varied.

Figures 4.7(left) and (center) plot the expected delay and success probability for the various drop policies. They also compare a stateless PD that uses a client-level hashing metric (H1), referred to as SLPD-IP. Our analytical predictions for the expected user-visit delay were consistent with the measured values and omitted to reduce graph clutter. The figure clearly shows the advantage of PD, especially on the mean visit delay due to its additive nature (Eq. (3.5)). We note that while the delay seems to be decreasing as $p^* > 0.6$, it is only an artifact from having user-visits with fewer parallel connections that are actually succeeding. Eventually, all visits are aborted by the client and is represented by a zero-valued point in the figure.

Figure 4.7(center) shows how user-visits are very sensitive to connection-level and random dropping policies since a visit is successful only if none of its requests times out. This sensitivity is reduced when client-level dropping (SLPD-IP) is performed, which is apparent in the linear relationship between success probability and the effective timeout probability. In effect, SLPD-IP is performing a form of low-level admission control, which maximizes the performance of the controller. Unfortunately, SLPD-IP has the least fairness among our PD implementations as it can target entire clients. As mentioned earlier, unless care was taken to deal with NAT and proxy servers, SLPD-IP may unintentionally block a large number of clients.

Figure 4.7(right) shows how the aggregate traffic changes among the different policies. Two important points should be observed. First, because the source traffic model is highly dependent, the aggregate traffic, Λ, decreases as the effective timeout probability, p^*, is increased. Our analytical model assumed independent traffic sources and is, thus, not suited for predicting Λ in this case. Second, for any given p^*, we can see the dramatic improvement in using any of the PD policies compared to a rate-based drop policy. From that perspective, our estimate for Λ highlights the relative (not absolute) improvement in using PD over a rate-based drop policy.

4.4.4 Limitations of the Study

There are still three specific limitations to our study that are worth mentioning. First, we have not discussed how a traffic controller would adjust p^* based on the measured arrival rates or router queue lengths. We believe that PD can be easily integrated into existing AQM techniques, which already have built-in adaption mechanisms [32, 56]. Because PD reduces the variability of aggregate traffic, it will improve the stability and responsiveness of such mechanisms. Second, we have assumed that clients have unique IP addresses. This provided SLPD-IP with a clear advantage over the other schemes as it mimicked application-level admission control policies. For this reason, we believe that its performance numbers are overstated, but nevertheless, still performs well when controlling large aggregate traffic as classification errors can be better tolerated.

4.5 Conclusions

Based on our earlier model of persistent clients, this chapter derived the optimal drop strategy for controlling aggregate traffic. Furthermore, we showed that persistent dropping yields the lower bound that an AQM technique can achieve in reducing the effects of packet drop on client-perceived delay and on the effective arrival rate. We presented two working implementations of persistent dropping based on hash functions that can be deployed in routers or end-servers. Persistent dropping can be considered as a low-level admission control policy. No application-level support is required for the correct operation of persistent dropping. In particular, when connection-level classification is performed (H2), persistent dropping does not violate any end-to-end semantics and, at the same time, achieve the same control targets as the traditional rate-based control. Furthermore, the improvement in the connection-establishment latency does not interfere with higher-level admission control mechanisms. On the other hand, client-level classification (H1) does violate the end-to-end argument, and it is presented here to show the full potential of an intelligent dropping mechanism in routers. One can argue that connection-level controls should be avoided in routers and left to the end-servers. We addressed this exact issue by showing that in some high-congestion cases, such as FCEs, routers are forced to drop new connection requests. Our technique provides an optimal way to achieve quick convergence to the control targets with minimal intrusion on successful connections. Finally, while our technique seems less effective in controlling or defending against DDoS attacks, it is indeed not more vulnerable than traditional rate-based techniques. The vulnerability of our scheme is only apparent in the choice of the hash function. This can be easily overcome by using more secure hash functions that an adversary cannot exploit. All that a DDoS attack can do is to increase the amount of traffic, which may force the controller to use a larger p^* value. This is no different than traditional control mechanisms.

CHAPTER 5

ABACUS FILTERS:

Controlling Bursty Service Requests

To deal with client's persistence, we introduced PD, a low-level admission controller. PD focused on minimizing aggregate traffic and connection-establishment latency. However, it relied on implicit notification when denying requests, namely let the corresponding connections time out. This chapter extends the notion of low-level admission control in two dimensions. First, it explores different ways of performing an explicit (reject) notification to further reduce connection-establishment latency. Due to the lack of universally supported techniques, this chapter advocates extending ICMP to provide a universal support of request rejection. Second, this chapter also presents an efficient traffic predictor to deal with the inherent burstiness of request arrivals. Both techniques are combined to create the *Abacus Filter*, an effective mechanism for controlling request arrivals to Internet services.

5.1 Introduction

During periods of heavy load, an Internet server must ensure bounded client-perceived latencies for arriving requests. As mentioned, NOSCs operate at the lower network layers and are the first-line of defense that must be tailored to match the intrinsic behavior of the underlying traffic. Thus, for new connection requests, it must be sensitive to delays incurred due to request dropping. Because of clients' persistence, commonly-used traffic control or shaping techniques such as Token Bucket Filters (TBFs) [109] are not effective in reducing the burstiness of arriving requests nor do they provide the necessary control over the connection-establishment latencies, and thus, cannot bound overall client-perceived delay. Particularly, the dropping of "out-of-profile" packets alone cannot deal with delays due to TCP backoffs and retransmissions, and, in some cases, can introduce instabilities in the system.

Following the same general goals in Chapter 4, we continues to investigate the shortcomings of traditional control mechanisms, but with the focus on bursty request arrivals. Furthermore, we complement persistent dropping by providing a self-adapting technique to different traffic load conditions. This technique can be easily tuned to balance between maximizing aggregate throughput and limiting connection-establishment latency—two often conflicting goals.

In order to develop control mechanisms that can provide the desired limits on client-perceived delays, we need to provide greater power to the server during periods of heavy load. In particular, we need to provide a mechanism by which the server can "reject" a client request. This way, a client whose requests will never be served is *explicitly* informed of this as soon as possible to minimize his connection-establishment latency even if it an indication of connection failure. The *reject* mechanism performs this action of informing a client that his/her request will not be served, and that s/he should not try to reconnect again. In a sense, the reject mechanism provides an extension to admission control that operates at fine granularity and is performed at the lowest networking levels. We elaborate on the actual technique in Section 5.2. A server will only resort to such extreme measures when it is certain that its overload will be sustained throughout the period of retransmission.

The goal of this chapter is to limit the average total delay experienced/perceived by all clients. We use a simple economic formulation to describe our basic goal. First, assume that each request incurs a cost, $h \times T_k$, where h is some constant and T_k is the delay for request k. A request will also incur a fixed cost r if rejected. Our objective function is to minimize the total cost, which can minimize the average client-perceived delay, given an appropriate setting of h and r. We use this economic formulation to create an analytic model of the expected throughput and client-perceived delay of the system. We also describe in Section 5.4.2 how the choice of h and r relates to the minimization of rejected packets and client-perceived delay.

Based on our model, we propose the *Abacus Filter* (AF), a hybrid token bucket implementation, tailored to the intrinsic behavior of SYN packets. Our approach maximizes the number of successfully-served requests, while keeping client-perceived delay in check. AFs estimate the amount of available future service capacity and only admit a portion of any request burst that, with high probability, will eventually be serviced. This way, bursty traffic or a sudden jump in network load does not affect the overall delay of successful connections.

This chapter is organized as follows. We first motivate connection-level controls and identify enforcement mechanisms in Section 5.2. We then characterize the impacts of bursty arrivals on traditional control mechanisms in Section 5.3. We formulate our analytic model and derive a predictive heuristic in Section 5.4. We present our new control mechanism in Section 5.5, and empirically evaluate some performance issues in Section 5.6. The chapter ends concluding remarks in Section 5.7.

5.2 Enforcing Traffic Control

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

Rejection of requests, at first glance, seems relatively easy to accomplish. RFC 1122 [24] defines that a host receiving an "`ICMP destination port unreachable`" message must abort the corresponding connection. This can, therefore, be used by servers to reject clients. Unfortunately, not all operating systems adhere to this specification. In particular, Microsoft OSs completely ignore these messages, as shown in Table 3.1, which details the reaction of various OSs to different control mechanisms. Furthermore, many servers are designed to limit the rate at which ICMP packets are sent out, following the specification in RFC 1812 [10]. So, the simple ICMP-based technique is not currently a promising reject mechanism.

Alternatively, the server can send a reset packet (which has the RST flag set in the TCP header) whenever it wants to reject a connection. This is similar to a mechanism used in some routers and firewalls to indicate queue overflow and avoid overload. A host receiving a reset packet must abort the corresponding connection as described in RFC 793 [103]. The use of reset in this context does violate its original intent, but does not conflict with TCP standards for correct client-application behavior, i.e., will not break existing client applications. Again, with the exception of Microsoft OSs, sending a reset achieves the desired effect. In this case, the Microsoft OSs did not wait for a timeout. Instead, they attempt to reconnect immediately when a reset is received. When the reconnect attempt also results in a reset, the client repeats the attempt, for a total of 4 tries (3 for

Windows 2000) before giving up.[1] In this respect, a client is informed of the rejected request, but at the expense of several round-trip times as well as increased network load. Finally, we are unaware of any OS implementation that would limit the rate at which reset packets are sent.

The above two approaches show that there is no universal method for rejecting a client. One can argue, however, that higher-level mechanisms, such as sending back an HTTP 500 status code, can achieve the desired goal. Unfortunately, implementing such mechanisms at the application level incurs almost as much server overhead as accepting the request, while implementing this anywhere below the application level (e.g., in the kernel) suffers from two major drawbacks. First, it requires a separate TCP stack implementation to mimic the process of completing the three-way handshake, accepting the requests, and then sending the reply that rejects the requests. This not only violates protocol layering, but also consumes more resources as indicated in Table 3.1. Second, a different reject mechanism and the corresponding stack implementation would be required for each high-level protocol, which limits its applicability to future protocols. So, a low-level reject mechanism that can regulate requests prior to connection establishment is the best approach to providing a universal reject mechanism that will be handled by all client applications gracefully. As we will show that such a feature is crucial for bounding client-perceived delays, particularly with highly-loaded servers, this will hopefully motivate the adoption of a universal low-level reject mechanisms for TCP connection requests. We have proposed an extension to ICMP that provides the required functionality in Appendix A. An ICMP is chosen, as oppose to modifying TCP, to allow for backward compatibility and incremental deployment. For the remainder of this chapter, we assume that such a reject mechanism is available. Prematurely aborted connections and manual reconnect attempts by human clients require a separate study that can incorporate their unpredictable behavior.

5.3 Limitations of Token Bucket Filters

Every Internet server implicitly implements connection dropping through the nearly universal use of fixed-size protocol queues. This is, in a sense, a control mechanism that limits the load on Internet servers to some degree by dropping requests when queues are full. Rate-based mechanisms, such as TBFs, perform explicit dropping to achieve more configurable objectives: they regulate the arrival of requests such that it conforms to some average arrival rate and limit the

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

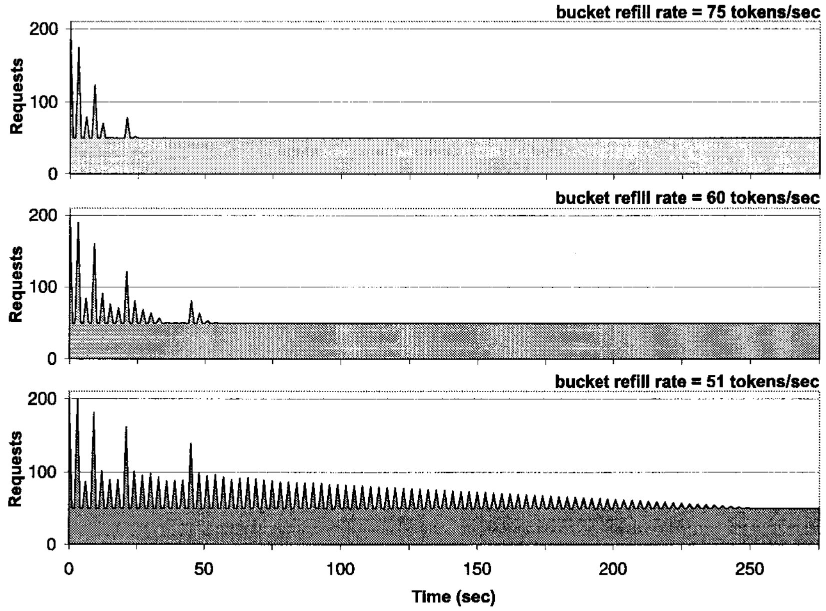


Figure 5.1: Burst convergence for TBFs with different acceptance rates r_t. A one second burst of 150 requests arrives on top of regular traffic arriving at a constant rate $\lambda = 50$ reqs/s.

maximum burst size admitted [71]. Unfortunately, uncontrolled drops directly impact both client- and server-perceived performance. An overloaded server that temporarily drops incoming requests will increase network load by causing an additional surge of retransmitted requests to appear at a later time. Furthermore, clients will experience exponentially longer delays when their connection requests are dropped.

A Token Bucket Filter (TBF) is a simple and well-known mechanism used for rate control [71]. In its general form, the TBF can be seen as a bucket of tokens, and is described by two parameters: a refill rate, r_t, and a bucket size, W. When a packet arrives at a particular TBF, a token is removed from the bucket and the packet is accepted into the system.[2] If the bucket is empty, then the packet is not accepted. The bucket is replenished with tokens at rate r_t, but is limited to a maximum of W tokens available at any time. This ensures an average acceptance rate of r_t while maintaining the ability to handle some burstiness in the arrival stream. The behavior and simplicity of TBFs make them useful for regulating general traffic flows. As they can provide good approximations of application- or system-level queues (such as the TCP listen queue),[3] TBFs can be used to match the number of requests entering the system to the server capacity by setting W to the maximum queue

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

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

length and r_t to the average processing rate of the system.

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

Figure 5.1 shows an example of this, where retransmitted bursts trigger additional dropping of new incoming requests. In this experiment, we simulate the effects of TBFs on bursty request arrivals using *eSim*. The clients are assumed to follow the retransmission behavior and distributions discussed in Chapter 3. We use a TBF to regulate the admission of new requests, which arrive at a constant average rate, except for a single initial burst. This unrealistic load arrival is used to determine the burst response of the system as the system becomes critically-loaded. We will look at more realistic arrival distributions later. For now, we vary the token refill rate, setting it closer to the traffic arrival rate, and set the bucket size, W, such that it can be completely refilled in 3 seconds. The intuition behind this setting is that the TBF matches the processing capacity of the server and the admitted requests should not be queued for more than 3 seconds; otherwise, the client will assume packet loss and retransmit them anyway, wasting network bandwidth and server resources. We also vary the initial burst length and the average arrival rate.

Based on this simulation, we observe that the difference $(r_t - \lambda)$ between the token refill rate and the average arrival rate greatly influences the stability of the system. As the system operates closer to full capacity (i.e., λ approaches r_t), it becomes critically-loaded, and any increase in the apparent arrival rate will trigger drops. The retransmissions of the original burst of requests that were dropped, due to their re-synchronization, will arrive at the server at roughly the same time, triggering further bursts. This will result in more synchronized drops of the retransmitted requests as well as many new requests. As a result of the cycle of retransmissions triggering new drops, which trigger further retransmissions, the instability may last for a very long time, continuing until all retransmissions time out or are slowly absorbed by the system. These behaviors are formally analyzed in Section 5.4.

Alternatively, we can try to avoid these stability issues altogether by rejecting requests when the token bucket is empty, instead of employing the drop-policy analyzed above. This will eliminate such instabilities, since the rejected clients do not retransmit their requests. Unfortunately, the added stability can be achieved at the cost of poor acceptance rates and low server utilization. This is most apparent with bursty arrivals, where the rejection policy is very sensitive to the choice of bucket size (Section 5.6). A small or moderate size bucket will provide the necessary traffic control and limit admitted bursts, but would reject a significant number of requests, resulting in poor server utilization. A very large bucket would permit more client connections, but translates into long queues in the system, effectively increasing service times and defeating the purpose of the traffic control.

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

5.4 Optimizing Traffic Controls

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

5.4.1 Network Model

In developing our analytic model, we need to specify carefully the parameters of the whole system. We can assume, without loss of generality, that senders adhere to a TCP-Reno style congestion control mechanism [68]. However, this alone does not suffice, as it identifies neither the arrival distribution of request aggregates nor the correlation between different requests in the presence of packet drops. The distribution of network traffic has been studied extensively [72, 80, 102] and is shown to exhibit a self-similar behavior. These results also extend to connection-level analysis due to the aggregation of independent ON/OFF traffic streams representing different user sessions [111].

Other studies [16,35,93] have measured the correlation between multiple users and across different requests or sessions originating from a single user. Unfortunately, self-similarity and user sessions do not follow traditional queuing models, complicating our analysis. In particular, accurately modeling a real system with many simultaneous clients, connecting to a server with a complex request arrival distribution and individual retransmission patterns will lead to mathematical intractability. We, therefore, continue to assume the network model in Chapter 3 where incoming requests are *i.i.d.* and can be classified into $n+1$ transmission classes, where $T_0 = 0$, T_i is the time before the i-th transmission and T_{abort} is the time before the connection aborts. The values for the T_i's and T_{abort} are measured in Table 3.1. Furthermore, we make the following five assumptions.

A1. Only new requests, not retransmissions, may be rejected. If a request is dropped, but not rejected, it continues to be retransmitted until it is admitted by the system or times out. This can only be done if requests can be classified into different transmission classes, which we relax later.

A2. A rejected packet will not be retransmitted again. We, thus, assume that clients have implemented out ICMP extension as specified in Appendix A.

A3. We ignore link-delay effects in our analysis. We further assume that delay jitter and RTO variations are negligible. This essentially assumes perfect re-synchronization, but does not impact the correctness of our analysis, since jitter and initial RTOs are typically distributed over a very narrow range, as reasoned about in Section 3.3.2.

A4. The system can be observed at discrete time intervals of T_0. Thus, when referring to time t, we are implying the interval $[t, t+T_0)$. The assumption of perfect re-synchronization in A3 allows this, since all requests dropped within the current time interval will be retransmitted in a single future time interval, as long as we select T_0 such that d_0 is an integer multiple of T_0.

A5. We assume equal treatment in dropping requests. During each observation interval, $[t, t+T_0)$, all arriving requests have the same drop probability, regardless of the number of times they have been retransmitted (i.e., regardless of the transmission class). Note that this is only for drops, not rejection (see A1).

The system presented here is used to develop an analytic model to describe the effects of traffic control on incoming connection requests. As we shall show in Section 5.6, despite the seemingly restrictive assumptions, analysis based on these assumptions can be used to develop control mechanisms that perform well under realistic load conditions, and provide effective control for real-world traffic.

5.4.2 Analytic Control Model

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

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

$$A(t) = \lambda(t) + \sum_{i=0}^{N-1} X_i(t - d_i), \tag{5.1}$$

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

$$X_i(t) = A_i(t)\left(1 - \frac{W(t)}{A(t)}\right)^+, \tag{5.2}$$

where operation $(y)^+$ is defined as $\max(0, y)$ to enforce a floor value of 0, $A_i(t)$ represents the number of requests belonging to the i-th transmission class that arrive during the interval $[t, t+T_0)$, and $W(t)$ is the number of tokens available at time t (the beginning of the observation period). The term $(1 - W(t)/A(t))^+$ defines the drop probability for any request during the time interval $[t, t+T_0)$, assuming an equal treatment of all incoming requests (Assumption A5). Since a dropped request belonging to the i-th class will be retransmitted after d_i, we can write $A_i(t)$ in terms of $X_i(t)$ as:

$$A_i(t) = \begin{cases} \lambda(t) & \text{for } i = 0 \\ X_{i-1}(t - d_{i-1}) & \text{for } 0 < i \leq N. \end{cases} \tag{5.3}$$

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

$$\begin{aligned} X_i(t) &= P_d(t)X_{i-1}(t - d_{i-1}) \\ &= P_d(t)P_d(t - d_{i-1})X_{i-2}(t - d_{i-1} - d_{i-2}) \\ &= \left(\prod_{k=0}^{i} P_d(t - \sum_{l=1}^{k} d_{i-l})\right) \lambda(t - \sum_{l=1}^{i} d_{i-l}). \end{aligned} \tag{5.4}$$

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

$$W(t) = \min\left((W(t - T_0) + T_0 r_t - A(t - T_0))^+,\ W\right), \tag{5.5}$$

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

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

It is important to note that our economic formulation assumes that all TCP connections are given an equal weight or priority. In reality, this may not be the case. Connections that belong to paying customers, for instance, may have a higher rejection cost than those that belong to non-paying ones. From that perspective, both delay and rejection costs may vary across different types of connections. The difficulty here is that a particular connection's real cost, a high-level application notion, is not directly available at the lower TCP-stack level, where traffic regulation (e.g., using TBF) is actually being performed. Therefore, we assume that if connections incur different costs, they should be classified first by an independent mechanism (possibly something similar to Layer-7 switching[4]) and that independent traffic regulators with different costs should be used for each class.

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

$$c(t) = c(t - T_0) + \left(\sum_{i=0}^{N} hd_i X_i(t) \right) + rX_N(t). \tag{5.6}$$

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

$$c(t) = \sum_{j=0}^{t/T_0} \sum_{i=0}^{N} hd_i X_i(jT_0). \tag{5.7}$$

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

$$c(t) = c(t_1) + \sum_{j=1}^{\frac{t-t_1}{T_0}} \sum_{i=0}^{N} hd_i X_i(t_1 + jT_0), \tag{5.8}$$

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

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

$$c^g(t) = c^g(t_1) + \left(\sum_{j=1}^{\frac{t-t_1}{T_0}} r\lambda_r(t_1 + jT_0) \right) + \sum_{j=1}^{\frac{t-t_1}{T_0}} \sum_{i=0}^{N} hd_i X_i(t_1 + jT_0), \tag{5.9}$$

for $t \geq t_1$, where $X_i(t)$ is the same as defined in Eq. (5.4).

Eq. (5.9) can be illustrated by the following example. Consider a system that is under-loaded until time t_1, so $c(t_1) = 0$. Now, let a burst of 100 SYN packets arrive immediately after t_1; if all of the packets in the burst are rejected, then the cost at $t > t_1$ is just $100r$. Here the third term in Eq. (5.9) is 0 since no packets will be retransmitted. However, if only a portion is rejected, then the

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

total cost is the cost of rejecting this portion (second term of Eq. (5.9)) plus the cost of the delay incurred by those packets that were dropped and then retransmitted at a later time (third term in Eq. (5.9)). Note that the second term also includes the effects of a retransmitted request inducing further drops by the system.

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

5.4.3 Predictive Control Policy

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

We look at a predictive heuristic that decides the control action by approximating the analytic cost function. Basically, when no tokens are available, the first requests, up to a threshold, λ_{thresh}, are dropped, and subsequent requests rejected. The predictive heuristic approximates the cost computation for future time t by looking only at the new requests that may arrive in the next time interval, and using this to select λ_{thresh}. Future retransmitted requests are still included in the heuristic. We can then simplify Eq. (5.9) as:

$$c^g(t) = c^g(t_1) + r\lambda_r(t_1 + T_0) + \sum_{j=1}^{\frac{t-t_1}{T_0}} \sum_{i=0}^{N} hd_i X_i(t_1 + jT_0), \tag{5.10}$$

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

$$c^g(t) = c^g(t_1) + r\lambda_r(t_1 + T_0) + H(t_1, t), \qquad \text{for } t \geq t_1. \tag{5.11}$$

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

$$c^{g'}(t) = c^g(t_1) + r\lambda_r'(t_1 + T_0) + H'(t_1, t), \qquad \text{for } t \geq t_1. \tag{5.12}$$

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

$$c^g(t) - c^{g'}(t) = H(t_1, t) - (H'(t_1, t) + r), \qquad \text{for } t \geq t_1. \tag{5.13}$$

Using the above formula, we can now find λ_{thresh}. Note that rejection of all requests will make the cost very high. As we reduce the number of rejected requests, the cost decreases. However, at some point, once we let too many requests in, and some begin to time out, the expected cost reaches its minimum and begins to increase. At this point, the cost difference in Eq. (5.13) will, for the first time, become positive. To find λ_{thresh} that minimizes cost, we can simply search for this occurrence:

$$\lambda_{thresh} = \min_{t \to \infty} \{\lambda(t_1 + T_0) : H(t_1, t) > H'(t_1, t) + r\}, \tag{5.14}$$

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

The predictive heuristic presented here uses an approximation of the analysis from Section 5.4.2, to determine the control policy that will minimize expected costs. This is really a greedy mechanism, that can, at best, provide a locally optimal cost under the assumption of no new arrivals in the future. Furthermore, as the cost incurred for a timed-out request is always greater than the cost for a new request that is rejected, this heuristic will switch to a reject policy at the threshold where an additional drop of a new request seems to cause some previous connection attempt to time out. In addition, this method is limited by the accuracy of its estimates of $\lambda(t)$ and $X_i(t)$. We will in Section 3.5.5 see how well this control mechanism performs using realistic traffic models that do not follow the assumptions we have outlined earlier.

5.5 Abacus Filters

A direct implementation of the predictive control mechanism developed above faces two critical challenges. First, the computation of λ_{thresh} at the beginning of each observation period, with its recursive variable dependencies, can inflict a very high computational overhead. This is especially counter-productive in servers that are already heavily-loaded. Second, we need an accurate measure

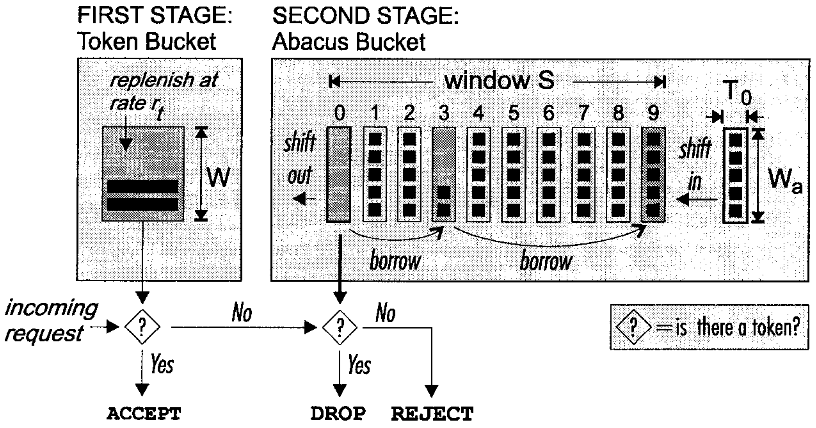


Figure 5.2: Abacus Filter architecture

of the rate of new requests in order to compute $X_i(t)$ values. As the computations are recursive, errors may accrue, degrading the accuracy of the estimates. In this section, we develop an approximate design that captures the benefits of our analytic model, but without its computational costs or the need for precise measurements. This new filter design, called *Abacus Filter (AF)*, is a two-stage hybrid token bucket implementation. The first stage limits the service load, as in a regular token bucket filter. The new second stage determines the appropriate control policy by estimating the number of packets that can be deferred for processing at a future time.

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

We can overcome the former by assuming the worst-case scenario that no tokens will be carried forward into future time periods. Therefore, the expected number of tokens, W_a, available during a future time interval to handle retransmission bursts is determined solely by the token refill rate, r_t, and the arrival rate of new requests, λ, over the time interval, T_0. Hence, $W_a = (r_t - \lambda)T_0$ represents the estimate of future excess capacity available to handle retransmission bursts for any future observation period, which is defined according to the guidelines in Section 5.4.1.

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

Combining these ideas, we introduce the AF, a control mechanism that incorporates two stages: the first stage is a regular token bucket, while the second stage decides the traffic control policy. The novelty lies in this second stage, which consists of a series of buckets, each of which has size W_a (Figure 5.2). Each of these buckets represents the number of retransmissions that can be accepted over an interval of T_0, the *time granularity* of the filter, which corresponds to the observation period in A4. Together, these buckets represent the future retransmission burst handling capacity over a *time window, S*. Rather than replenishing the second-stage filter one token at a time, once every T_0 seconds the token buckets are shifted, dropping the first bucket and shifting-in a new bucket at the end. This models a discrete time progression over the time window S. The new bucket is filled with $W_a = (r_t - \lambda)T_0$ tokens, the estimate of excess resources available in the future based on an average new request arrival rate of λ.

When a burst of requests arrives, the first stage determines the packets that can be accepted immediately. The remainder are passed to the multi-bucket second stage for making a control policy decision. If tokens in these buckets are available, the packets are dropped and allowed to retransmit; otherwise, they are rejected. The AF looks for tokens in future buckets that represent the time the dropped packets will be retransmitted. An example is shown in Figure 5.2, where a burst arrives at an initialized AF. Consequently, the filter first looks d_0 or 3 seconds into the future, then d_1 or 6 seconds after that, and so on. Only if the entire series of buckets is exhausted, the particular request is rejected. This "borrowing" of tokens, somewhat akin to operating an abacus, performs the same function as the $A_i(t)$ values in our analytic model, which account for the future arrivals of request retransmissions. This way, future request arrivals contending for the same resources will be appropriately controlled.

The borrowing pattern depicted in Figure 5.2 assumes that all dropped requests belong to a single transmission class, i.e., new requests that are dropped. Processing capacity that will be consumed by the first retransmissions of these dropped requests, as well as all subsequent retransmissions for the portion of the retransmissions that is expected to be dropped again, is accounted for by the removal of tokens in the future buckets. At a future time, e.g., 3 seconds later, when the retransmissions arrive, a portion of these will be dropped and retransmitted again. However, the future capacity needed to handle the subsequent retransmissions has already been accounted for by the token borrowing during the initial drop of the requests. Hence, the portion of retransmissions not immediately accepted can be simply dropped without borrowing. The borrowing process is used only for new arrivals that are not accepted by the first stage token bucket.

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

In addition to the token bucket parameters, W and r_t, two basic parameters determine the operation of the AF: *time granularity*, T_0, and *window size*, S. The value of T_0 has to be chosen such that it is much larger than the average jitter of the link delays experienced by arriving packets and also large enough to accommodate variations in d_0, as outlined in Section 5.3. The window size, S, limits the future borrowing scope of the AF. This value must be selected to minimize total cost in the system. Since for each second of delay for a request, cost is increased by h, while an immediate reject incurs cost r, intuitively, one should reject a request if its expected delay exceeds r/h, and drop it otherwise. This is enforced by setting window size $S = r/h$. Since r/h can be arbitrarily large, we show in Section 3.5.5 that setting $S = \max\{T_i : T_i \leq r/h\}$, where T_i is the time of the i-th retransmission ($T_i = \sum_{l=1}^{i} d_l$), does indeed minimize total cost.

5.5.1 Implementation

The actual implementation of an AF is fairly straightforward. We extend the regular token bucket filter by adding the AF second stage to select the drop or reject policy. The AF's second stage is implemented as a circular array, where the elements represent the number of available

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

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

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

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

Figure 5.3: Borrowing algorithm of the Abacus Filter

tokens in each bucket. Each time a bucket is removed and a new one appended, an index indicating the element corresponding to the current bucket is increased. The array element associated with the removed bucket is reused for the newly-appended one, and is set to the value $W_a = (r_t - \lambda)T_0$. As the average arrival rate of new requests cannot be known *a proiri*, and can change over time, λ is actually an estimate based on the weighted-time average of the observed request arrivals over the most recent observation periods. To determine the number of requests that can be dropped, the filter looks at the appropriate array entries in the future according to the borrowing algorithm in Figure 5.3, using a simple modular arithmetic to handle wraparound of the index.

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

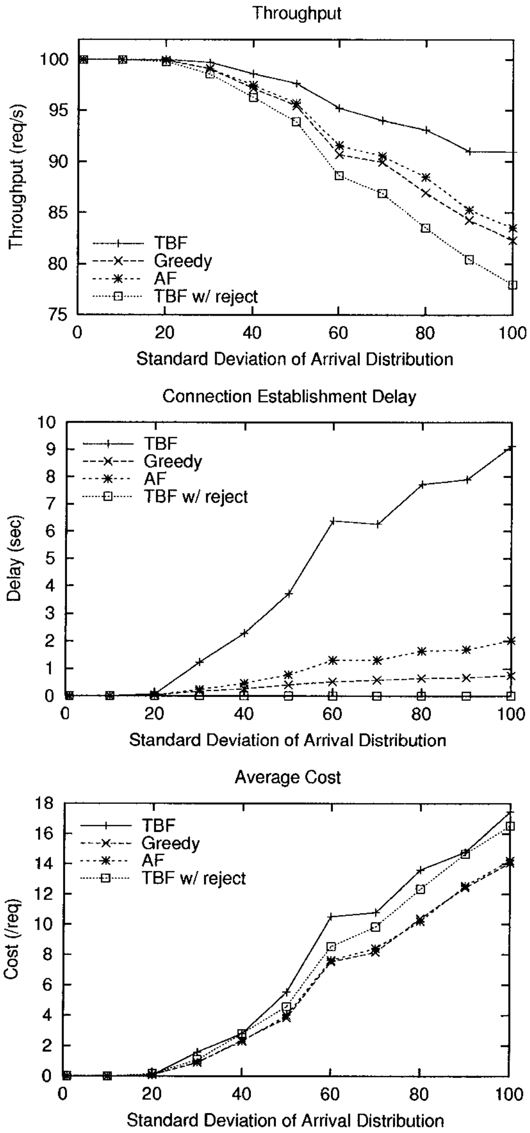


Figure 5.4: Effects of burstiness

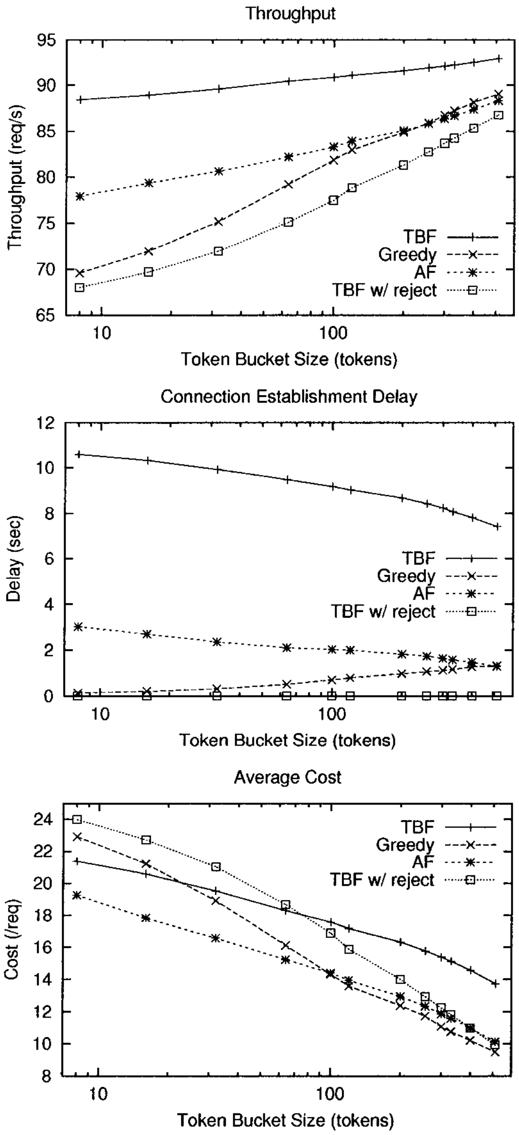


Figure 5.5: Effects of bucket size

5.6 Evaluation

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

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

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

AF is a direct implementation of the Abacus Filter as proposed in Section 5.5. The AF is characterized by four parameters: bucket size, W, refill rate, r_t, window size, S, and time granularity, T_0. In all experiments, we set $T_0 = 1$ sec, sufficiently large to overshadow the effects of RTO and delay jitter, yet small enough to maintain a high level of accuracy in estimating server capacity.

Predictive Greedy (**Greedy**, for short) is based on the predictive control policy that was developed in Section 5.4.3. Our implementation does not distinguish between transmission classes, so that packets are dropped/rejected indiscriminately across all service classes. This introduces inaccuracies in its predictions since our development in Section 5.4.3 relies on the fact that only new requests can be rejected, and retransmitted ones are always dropped if they cannot be accepted. As we shall show, this negatively affects the filter's overall performance.

In creating the simulated environment, our main goal is to subject these filters to realistic load conditions so that any results would be applicable to real-world deployment scenarios. We also want to avoid any unnecessary complexity without sacrificing accuracy. We, therefore, employ a simple setup where a server receives incoming requests through a high-speed link. Clients on the other side of the link also have enough resources to generate the desired request arrival distribution. Each incoming request to the server must first pass through one of the filters described above, which is located at the entry point of the server (the lowest level of its communication protocol stack). To eliminate external effects from our measurements, we make two assumptions about the system under test. First, we assume that the client-to-server network path is bottleneck-free, with negligible propagation delay (however, with a jitter distribution that follows the one in Section 3.3.2). Second, we assume that a request, once accepted, completes the connection handshake immediately, with no delay. The latter models a typical server, where SYN requests are handled in the operating system, and hence unaffected by processing delays and application queues. Therefore, the results presented in this section only reflect the effects of the underlying control mechanisms (i.e., one of the four filters) on the first half of the connection-establishment handshake. This way, one can apply the results in a wider range of operational scenarios.

Because we need to conduct a large number of experiments to cover the wide range of variable parameters, we evaluate the filters using an event-driven simulation, ensuring that long-duration simulations may be performed in reasonable time (simulating 54 days of non-stop testing in less than 6 hours). A self-similar traffic model is used to generate the arrival distribution of new requests. Self-similarity is crucial in describing the bursty behavior of current network traffic. Specifically, we use the *Multi-fractal Wavelet Model (MWM)*, proposed by Riedi [106], as our underlying request generation model. While other approaches [49, 101] also produce self-similar traffic, the MWM-based approach is shown to have better flexibility and accuracy in modeling a wide range of real network traffic conditions. Traffic generation relies on four basic parameters: the mean, variance, and *Hurst* parameter of the arrival distribution, and the number of wavelet scales [106]. In our simulation, we use a mean of 100 reqs/sec and a Hurst parameter of 0.8, a commonly-measured value in real network traffic [80, 106]. The number of wavelet scales is set to 10 to generate 2^{10} data points, each of which represents the number of new client arrivals in a 1-sec interval. Finally, the variance is, in a sense, an indicator of the burstiness of the trace, so we use this parameter to vary the burstiness of the generated traffic.

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

We want to characterize the behavior of the four filters when the underlying system is critically-loaded. By this we mean that the average arrival rate of new requests is close to the server processing rate, which is reflected in the token refill rate. Thus, we set $r_t = 120$ tokens/sec, reflecting a server that has 20% greater capacity than the mean request-arrival rate of 100 reqs/sec. Configuring the system to be under-utilized or over-utilized would yield predictable results: TBF would work well

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

in under-utilized scenarios, while TBF w/ Reject would be best in over-utilized cases. In what follows, each experiment is set to simulate 600 seconds of incoming requests, and is repeated 30 times, amounting to roughly 1.8 million new arrivals for each plotted point.

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

5.6.1 Effects of Load Characteristics

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

Figure 5.4 shows the throughput, connection-establishment delay, and average cost for all four filter designs. It shows that as the burstiness of incoming requests is increased, it negatively impacts the performance of all four filters. Furthermore, the TBF provides a higher throughput for bursty arrivals, but performs poorly in controlling client-perceived delay. The above figure also highlights the importance of using the AF or Greedy filter, namely, as the burstiness of traffic is varied, the expected delay remains in check. Of course, throughput is limited by the fixed system capacity, so when traffic becomes increasingly bursty, fewer requests will succeed (as more will be rejected), which will, in turn, decrease the overall percentage of accepted requests. The figure also shows that a reject policy (TBF w/ Reject) provides throughput only 5% and 8% less than the AF and Greedy filters, respectively, but with a cost about 15% greater.

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

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

5.6.2 Effects of Configuration Parameters

The second step in our evaluation is to study the effects of the main configuration parameters on the performance of the filters: bucket size (W), rejection cost (r), and window size (S). We use a similar setup to that in Section 5.6.1, configure the token bucket size to $W = 120$ tokens, use AF and Greedy window sizes of 50 seconds, and set the costs of rejection and delay to $r = 75/req$ and $h = 1/(sec \cdot req)$, respectively. We ran a separate test for each parameter, varying it while fixing the others.

Figure 5.5 shows our three performance metrics as the bucket size is increased from 8 to 512 tokens. Note that W is doubled at each successive point, and the graph is plotted using a log scale. The figure shows that as the bucket size increases, so does the performance of the filter. Larger buckets allow the filters to accept larger bursts into the system with fewer retransmissions, rejects, and timeouts, increasing performance. Of course, we cannot arbitrarily increase bucket size, as this would overload the server, increase application queue lengths, and worsen service times. However, since we are only looking at the connection-establishment performance, such application queueing effects are not apparent in our metrics. It is interesting to see that the performance of the filters improves linearly with an exponential increase in bucket size. This is due to the bursty behavior

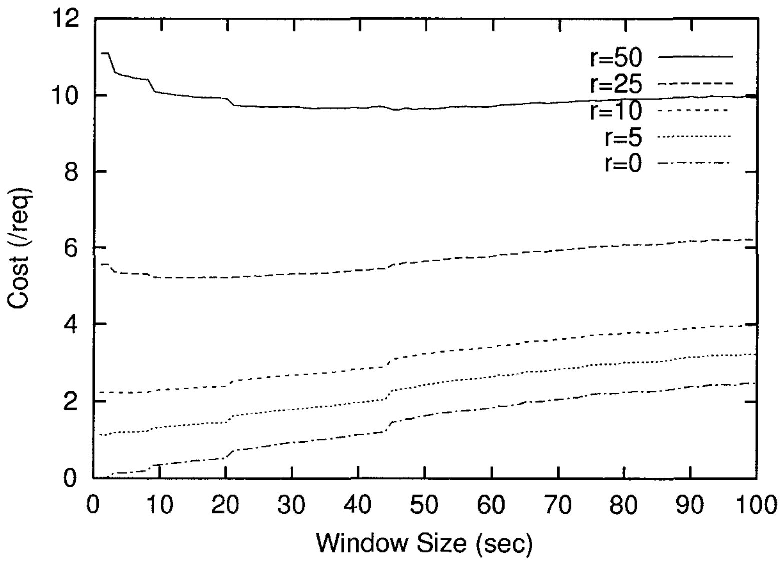


Figure 5.6: Relationship between Abacus window size and rejection cost

of self-similar traffic across multiple time scales. Even with the highly bursty request arrivals, it is surprising that an increase in bucket size produces only a logarithmic improvement in overall performance of the filters.

In Section 5.4.2, we showed the important role that the rejection cost plays in the derivation of the optimal control policy. A small rejection cost implies that it is more desirable to reject incoming requests than increasing the average connection delay. A large r indicates that it is worth sacrificing delay to avoid losing clients through rejects and timeouts. Recall that the cost of a timed-out connection request is the total delay cost plus the rejection cost, since it suffers the same effective result as a reject, but with a longer client notification time. To evaluate the performance impact of r, we first establish the relationship between the AF window sizes, S, and the costs h and r. Since it is the relative values of r and h that matter, i.e., ratio r/h, we fix $h = 1/(sec \cdot req)$ and vary both S and r. Figure 5.6 shows the performance, in terms of total cost, of the AF as S is increased. Each line in the figure represents different r values. Two observations are directly made from the figure:

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

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

Using the above relationship to set the AF window size to minimize cost, we evaluate the impact of rejection cost on the performance of the underlying filters. Here we set S as described earlier and maintain $S_G = 50$ sec. Unlike the AF window size, the Greedy window size need not vary with r/h. As mentioned in Section 5.4.3, the value of S_G should be set to at least T_N. Figure 5.7 shows the performance impact of the rejection cost on the four filters. Overall, as the rejection cost is increased, the throughput of the TBF and Reject filters remain unchanged, but the differences between the average costs increase dramatically. Both the AF and Greedy filters, which explicitly account for costs, are significantly affected, and are biased towards rejecting requests when r is low. This changes as the cost of rejecting a clients becomes greater than the delay costs of retransmission. However, no matter how high the rejection cost is, they never switch to a drop-only policy. This is because the cost of a timed-out request is always greater than that of a rejected packet, so it is always preferrable to limit the expected number of timed-out requests by rejecting some arrivals. This is directly reflected in the computation of λ_{thresh} for the Greedy filter and in the borrowing behavior of the AF. Over a wide range of rejection cost values (0 to 500), the average percentage of clients that time out is 5.5% for TBF, 0.01% for Greedy, 0.04% for AF, and 0% for Reject (as it does not allow retransmissions). Finally, when the rejection cost is very high, TBFs, the only filters that do not use reject, incur the lowest costs. This may indicate that both the Greedy and AF tend to predict more timeouts than really do occur, and reject more requests than they should. Of course, with varying traffic conditions, the actual crossover points in these graphs will change greatly, but for a large range of reasonable rejection-cost values, AFs seem to work very well.

Overall, our experiments demonstrate the effectiveness of the AF in controlling client-perceived delays. More remarkably, the simple design is able to find a balance between a lenient control policy (as with a TBF) and a strict one (as with a TBF w/ Reject) in response to changing request load conditions. It, therefore, shows the greatest advantage over the TBF under realistic, highly-bursty loads, while the latter performs well only under predictable loads. The Greedy filter, based on our analytic derivations, also achieves good results, but at a much higher computational cost. We note that this is just one of many possible implementations of our analytic results from Section 5.4, so with different assumptions and simplifications, it should be possible to derive filter designs that yield improved performance. In particular, improved estimation of arriving request distributions (e.g., using SYN caches [81]) and the request transmission classes is key to improving these control mechanisms. Specifically, this may reduce the apparent over-estimation in both the AF and the Greedy filter of future requests that would time out, which results in a larger number of requests

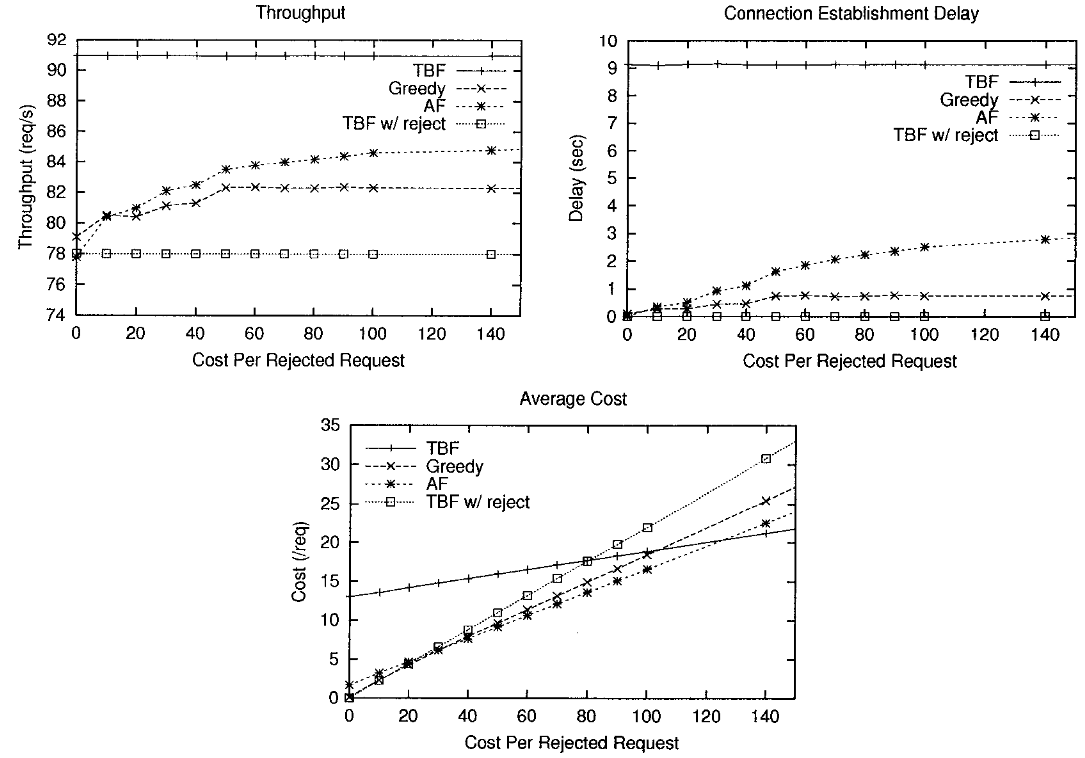


Figure 5.7: Effects of rejection cost

rejected than is absolutely necessary to minimize costs.

5.7 Conclusions

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

CHAPTER 6

MODEL AND CONTROL OF MULTI-THREADED INTERNET SERVICES

Better understanding of the internal dynamics of Internet services is a key element in the design of effective NOSCs. In this chapter, we take an analytical approach to answer key questions in the design and performance of rate-based techniques for providing QoS support to Internet services that are based on the multi-threading or multi-processing abstraction. Key to our analysis is the integration of the effects of concurrency into the interactions between these multi-threaded services—an important factor that is commonly ignored by previous research. In our new model, we first develop powerful, yet computationally-efficient, mathematical relationships that describe the performance (in terms of throughput and response time) of multi-threaded services. We then apply optimization techniques to derive the necessary rate-limits given specific QoS objective functions.

6.1 Introduction

With the prevalence of the Internet, more complex Internet services are continually being developed and deployed. This wave of new services has created unprecedented demands on both the network and end-servers. Unfortunately, the increased demands by end-users can easily saturate network links and/or overload end-servers. The notion of Quality-of-Service (QoS) is introduced to address the problem of not having enough resources to keep everyone happy. When resources are scarce, the QoS mechanism will ensure proper division and allocation of network and server resources. Supporting QoS has been addressed extensively in the literature both in the network and end-systems, for example, in [6, 9, 14, 26, 105, 120]. Without explicit OS support, such mechanisms have limited capabilities in enforcing strict service guarantees and are often restricted to only pro-

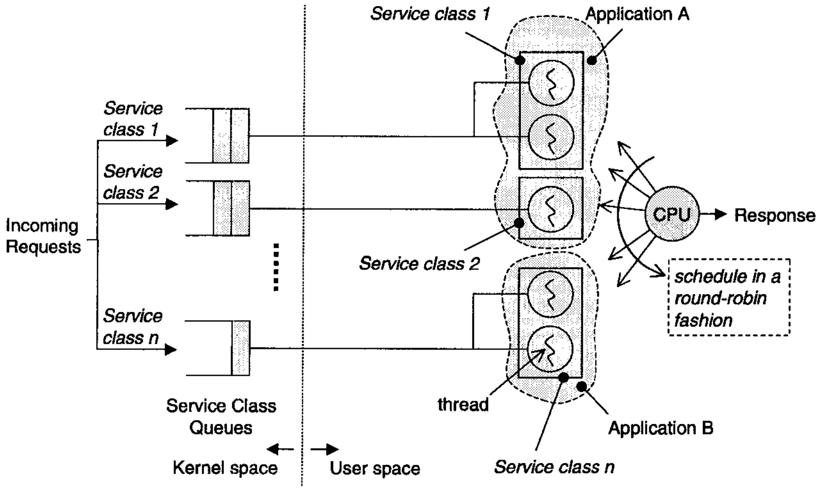


Figure 6.1: System model. Application A uses two service classes to give preferential treatment for requests in service class 1 than requests in service class 2. Application B uses one service class to enforce a certain QoS to all incoming requests. A controller (not shown) can then adjust the acceptance rate to different service classes.

viding proportional QoS differentiation. In this chapter, we closely examine and evaluate the extent to which NOSCs can provide QoS differentiation.

Because NOSCs are used to regulate the arrival of requests to running services, the first step is to build accurate models that capture the interactions between such services. Here, we focus on the effects of the acceptance rate on the performance of the different services being hosted on a single machine. Two design principles motivate the use of rate limits to provide QoS differentiation to multiple services: (1) increasing arrival rate will allow a running service to accept more requests into service, which allows the overlapping of long blocking I/O operations of one request with non-blocking operations of another, and (2) server capacity can be divided in proportion to the acceptance rate. Unfortunately, the extent to which rate-based QoS differentiation is effective depends heavily on the degree of interaction between the running threads, which further depends on the nature of the workload of incoming requests. This chapter carefully examines each of the two design principles with the goal of providing deeper understanding of internal dynamics behind this QoS mechanism.

When a service is allowed a higher acceptance rate, the resulting increase in concurrent processing of requests are apparent in the increase in service's throughput. There is, however, a *saturation point* beyond, which reflects the minimum rate that maximizes the service throughput. We focus in this chapter on locating this point as part of measuring the impact of increased concurrency on client-perceived delay and server performance and preventing the buildup of system queues.

When multiple services use rate limits to improve their own performance, the interaction between them become more complex (and less predictable) due to the non-trivial dependencies between shared resources. In fact, when different types of workloads (e.g., I/O-heavy and CPU-heavy) are sharing the system, a marginal improvement in the QoS of one service can cause a dramatic decrease in the QoS of the other services. Based on our measurements and observations, we show that a rate-based approach is ill-suited for providing service-level QoS support. On the other hand, it can be used effectively to provide QoS guarantees to different client groups.

In this chapter, we take an analytical approach to characterize precisely the interactions between multi-threaded services in an Internet server. Crucial to the correctness of our analysis is developing an accurate model that reflects the operation of the server. We introduce the multi-threaded round-robin (MTRR) server model to capture the multi-threading and process-sharing abstractions of real systems. The MTRR model assumes that the server is shared by a number of threads where an application is composed of one or more threads; each thread is given control of the underlying system for a time-quantum before the next thread is scheduled in a round-robin fashion. The MTRR model is an extension of traditional round-robin servers, which are used in the analysis of polling and time-shared systems [73, 75, 118]. Unlike traditional approaches, our model incorporates the performance benefits of increased concurrency into the interaction between the running threads. Using this MTRR model, we are able to derive powerful, yet efficient, relationships that describe the internal dynamics of a typical multi-threaded server. Furthermore, these relationships allow us to address three important issues in the design and performance of rate-based QoS differentiation: (1) better predict the impact of a given arrival rate on client-perceived delay than traditional models, (2) estimate the expected performance of services for any rate allocation, (3) find the rate allocation, if any, that guarantees certain response times to different client groups (e.g., paying customers are given preferential treatment over the non-paying ones).

This chapter is organized as follows. We analyze, in Section 6.2, the benefits of concurrency in multi-threaded applications. We then establish the server and application models for our analysis in Section 6.3. Section 6.4 provides a detailed analysis of MTRR server to provide basic relationships governing the performance of single and multi-class services. We then provide, in Section 6.5, a computationally-efficient method for determining the optimal allocation that meets various QoS objectives. We use real measurements on a typical Web server in Section 6.6 to evaluate the correctness and effectiveness of our derivations. Finally, in Section 6.7 we conclude the chapter with our final remarks.

6.2 Quantifying Concurrency Gains

Understanding the effects of increased concurrency is a key element in analyzing typical Internet services. The basic idea is that increasing the number of concurrent threads or processes of a particular server improves the performance of that service. In particular, the performance gain due to increased concurrency is normally split into three regions as shown in Figure 6.2: (I) a linear increase region due to overlapping blocking operations of some threads with non-blocking operations of the other threads, (II) flat or no-gain region due to threads contending for the bottleneck resource, and (III) sudden (or exponential) drop region due to memory thrashing. In this section, we establish this behavior for different workloads on a real system. This will set the stage for exploring the impact of concurrency on the controllability of multi-threaded Internet services.

We define $G_k(m)$ as the *speedup function* that expresses the potential performance gain (or loss) when m threads[1] run concurrently. Because the expected speedup is workload-dependent, the function needs to be profiled for each specific workload—denoted by the subscript k. The speedup function expresses the change in throughput rather than the change in response time. This is because increasing concurrency does not reduce the actual amount of work that each request needs. Instead, it increases the efficiency of the server, which can be captured by the improved throughput. To profile $G_k(m)$, we must first measure the maximum service throughput, $\hat{\mu}_k(m)$, when m threads run concurrently. This is done by limiting the application to have a maximum of m concurrent threads (for $m = 1, 2, \ldots$) and configuring the arrival rate to be high enough to keep all threads busy processing incoming requests. The speedup function is, then, the throughput gain when m threads are allocated compared to when a single thread is allocated. Specifically,

$$G_k(m) = \frac{\hat{\mu}_k(m)}{\hat{\mu}_k(1)}.$$

To illustrate the general characteristics of concurrency improvements, we configured a server machine (a 2.24 GHz Pentium 4 with 1 GBytes of RDRAM) to run Apache 1.3 and receive HTTP requests through a high-speed FastEthernet link. Up to three Linux-based machines are used to gen-

[1] we use the terms "threads" and "processes" interchangeably in this chapter

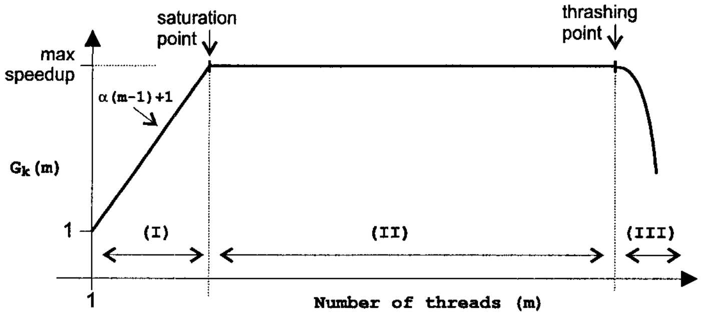


Figure 6.2: Shape of the speedup function, $G_k(m)$. The function increases at a linear rate in region I up to the saturation point. The function, then, flattens out in region II and suddenly drops after the collapse point in region III.

erate the desired requests. Our load generator, Eve (Section 3.5), follows the same design principles provided by SPECWeb99 [37], a widely-used tool to evaluate the performance of Web servers, to test static and dynamic workloads. The primary difference between the two load generators lies in our ability to sustain an arrival rate regardless of the progress of on-going requests. In contrast, SPECWeb99 sends a fixed maximum number of requests; once the maximum is reached, a new request is sent only after the completion of a previous one. We profiled $G_k(m)$ for three workloads: purely static, purely dynamic, and mixture of the two—mixed for short. Each workload adheres to the specification provided by SPECWeb99; in general, the requested files follow a Zipf distribution [16] regardless of whether they are statically or dynamically generated.

Figure 6.3 shows $G_k(m)$ for the three workloads, with the abscissa drawn in log-scale. The figure also shows the 95% confidence interval. The first two regions outlined earlier are clearly depicted by the figure, where the linear region is reflected by the sub-linear growth in the log scale. The combination of having a fast machine with large memory and running processes with small memory footprints prevented reaching the collapse point. This was the case even when a very large number of processes run simultaneously.

The width of the linear increase region (i.e., region I) in Figure 6.3 and its slope depend heavily on the type of workload. We approximate the speedup function in region I by a simple linear function:

$$G_k(m) = \alpha_k(m-1)+1 \quad \text{for } m = 1, \ldots, m_k^t,$$

where m_k^t reflects the saturation point (defined later), and the slope, α_k, reflects the *speedup rate*, or alternatively, the efficiency of concurrency for workload k. In the ideal case, where each additional

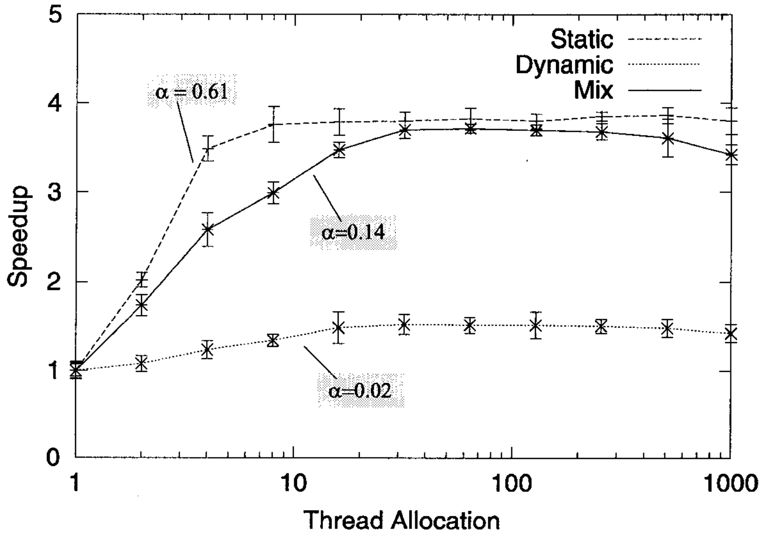


Figure 6.3: Speedup function, $G_k(m)$, for static, dynamic, and mixed workloads.

thread behaves as an independent server, $\alpha_k = 1$. This is seldom the case, and therefore, $\alpha_k \leq 1$. The mixed workload, for instance, had a speedup rate $\alpha_{mix} \approx 0.14$ and a linear increase region of $m^t \approx 23$ threads. If the workload is purely CPU-based or purely I/O-based, then one expects little performance gain since blocking and non-blocking operations are not overlapped. In that case, $\alpha_k = 0$.

The transition point between regions I and II, which we call the *saturation point* ($\hat{m}$), is primarily due to threads contending for the bottleneck resource—usually the disk. When a single class is being controlled, increasing the number of allocated threads beyond the saturation point provides no performance advantage to the hosted service. But when multiple services are being controlled, recognizing the saturation point becomes more crucial since adding more threads to one service class (or equivalently increasing the maximum allowed arrival rate) reduces the second class' share of the system. This may cause the second class to increase its thread allocation (or its maximum allowed arrival rate) and create a vicious cycle between the two classes rendering the underlying control mechanism ineffective. It is necessary, thus, for any dynamic control mechanism to adjust the maximum allocation based on the observed throughput; when no throughput gain is observed, then no further actions should be taken. This issue is explored closely in the remainder of the chapter.

6.3 Modeling Multi-threaded Services

The complexity of today's servers presents a real challenge in building analytic models that fully describe the dynamics of the underlying server. Our goal is, thus, to create a model that is simple enough to allow for mathematical tractability, yet accurate enough to reflect realism. Specifically, the created model must capture the effects of concurrency as well as the basic interactions between the various running services. In this section, we give a detailed specification of our system by describing the computing model, which details the assumed operation of a typical multi-threaded server, and the workload model, which specifies the arrival and service-time distribution of incoming requests.

6.3.1 Computing Model

Our computing model is based on a general understanding of the typical operation of current time-sharing OSs and Internet services. We focus here on mechanisms that are common to a wide range of OSs and services instead of restricting our model to reflect a single implementation. Using a general model, unfortunately, reduces the accuracy of our predictions when applied to a specific implementation. Nonetheless, our approach enable us to make broader conclusion on the fundamental dynamics governing an Internet server. We use an MTRR server to model a general computing environment where a single processor is shared by multiple threads. Threads are assumed to be the smallest allocatable unit of work and are distributed among n service classes, $\{S_1, S_2, \ldots, S_n\}$. Specifically, each service class S_k is allocated m_k^0 threads and has an independent buffer of size B_k to hold the requests that cannot be processed immediately. We use the term "service classes" as opposed to just "services" to capture the situation where a single service is configured to differentiate between multiple client populations (Figure 6.1). An example of this is Apache's *Virtual Host (VH)*, where, for instance, clients from network `192.168.10.x` are serviced using one VH and clients from the remaining IP address-space are serviced using another VH. Thus, our Apache service is said to have two service classes. In contrast, if an application does not differentiate between clients, the entire application is represented by a single service class. Using the notion of service classes, therefore, allows us to capture QoS differentiation between different applications and also between client groups within a single application.

Beside having threads as a shared resource, dependencies between service classes arise due to two possible interactions: (1) they share a bottleneck resource such as a disk and (2) they are organized as a series of stages where an incoming request must be processed by multiple services in a particular order [19, 120]. In this chapter, we restrict our analysis to single-stage services and focus on the dependencies due to resource sharing. We, thus, make the following assumptions for the internal operation of an MTRR server.

A1. A request is assigned to a working thread. Multiple requests can be processed simultaneously by running multiple threads that time-share the system. We assume that all threads are homogeneous,[2] even though they can be assigned to different service classes. This is in line with actual OS operation as system threads can be created and removed easily with little overhead.

A2. A thread is either running, ready, or blocked waiting for a new incoming request. Basically, a ready thread is waiting for its share of the server to continue processing a request, and a blocked thread is waiting for a new request. We do not consider alternate states in which a thread is waiting for other operations to complete such as blocking for I/O. These are captured by the speedup function.

A3. All threads are of equal priority. Service priorities have been studied in both queueing and real-time systems [76, 121]. Including service priorities in our model will, unfortunately, complicate our analysis and is, thus, omitted from our model.

A4. Threads (in the ready state) are scheduled (by the underlying OS) in a round-robin fashion, each for Q seconds or until the thread finishes processing the current request, whichever happens first. The task of servicing all ready threads once is called a *service round*. We do not consider the effects of hierarchical priority queueing, which is commonly used to age long-running threads. Since all requests are relatively short-lived and all threads have the same priority, a strict round-robin algorithm can be accurately assumed.

A5. Switching between different running threads is done instantaneously with no overhead. Similar to A2, we capture this overhead in the speedup function. Our decision is motivated by the fact that switching overhead is load-dependent. That is, as more threads are running, switching between threads will depend on whether the threads need to be swapped out of memory or not. The speedup function allows us to include load-dependent overheads in our analysis.

A6. The system has a fixed (finite) number of threads, m^{max}. This corresponds to the maximum

[2] That is, we do not mix different types of threads such as application-level and kernel-level threads.

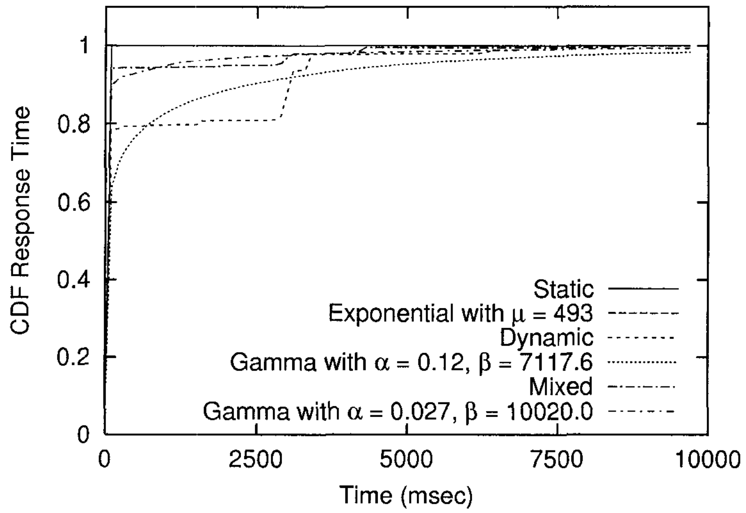


Figure 6.4: CDF and best-fit function of service time for the three workloads.

number of threads that a typical OS can support. Not all threads need to be allocated, but, the total number of threads that are allocated to all service classes cannot exceed this limit.

One final point to make is that our analysis does not consider any particular server resource as the bottleneck resource. Instead, the server is limited by the rate at which it can process requests and this rate is defined by the service-time distribution and speedup function of incoming requests.

6.3.2 Workload Model

In an Internet server, the workload model captures the arrival of requests and service that each request requires. Both were studied extensively in the literature [7, 11, 15, 49, 111]; in general, they have been observed to follow heavy-tail distributions: for request arrivals, it is due to the ON-OFF nature of client browsing behavior, and for the required service, it is due to few requests having very long completion times. In fact, we have observed similar behavior during when we profiled the distribution of the service time, $F_k(t)$ for our three workloads: static, dynamic, and mixed. To profile $F_k(t)$, we configured Apache to run using a single process and with the minimum listen queue length. Having a single process eliminated the effects of concurrency from our measurements, and having a short listen queue minimized the queueing delay component in our measurements. Because we expected the distribution to be heavy-tailed, we ran the experiment long enough to collect over a million sample measurements [39]. We combined these values to create the cumulative distribution function (CDF) of the response time for successful connections. The CDF for the three workloads are shown in Figure 6.4. The figure also shows the best-fit functions using standard distribution-fitting techniques [63]. The effects of long and frequent I/O operations can be seen in the step-like growth of the dynamic plot. In contrast, the static workload is well-behaved and can be approximated using an exponential distribution (it overlaps the measured lines). As we show later, I/O-heavy workloads can severely impact the controllability of running applications.

Unfortunately, heavy-tail distributions are hard to analyze even with very simple computing models. In order to provide better understanding of the dynamics of multi-threaded services, we assume that requests arrive following a Poisson process and require exponential service times. We evaluate our model, in Section 6.6, using realistic load distributions.

We distinguish between *service time* and *processing time* of an incoming request. The former reflects how much work that each request brings to the system, whereas the latter reflects how much time it spends in service as it shares the system's resources with other requests. Three parameters are, thus, associated with each service class S_k:

λ_k: the mean request arrival rate of a Poisson arrival process. When a maximum arrival rate is enforced using a NOSC, we use the term λ_k^{max} to reflect the mean arrival rate of the controlled arrival process.

$1/\mu_k$: the mean service time of each request. As mentioned, the real service time depends on the allowed concurrency due to overlapping of blocking and non-blocking operations, resource sharing, and context switching. Therefore, it is equal to the processing time only when the system is allocated a single thread.

$G_k(m)$: the speedup function as defined in Section 6.2. Even though we use the subscript k to denote the service class—not the workload—the characterization of $G_k(m)$ is unchanged. For example, if $G_1 = G_2 = G_{static}$, it implies that both service classes have static workloads. We assume that $G_k(m)$ only operates in regions I and II (Figure 6.2). The point where $G_k(m)$ collapses is hard to predict *a priori*, but not very difficult to detect (See Chapter 7). For the purpose of our analysis, we assume that detection/prevention from server thrashing is handled by a separate mechanism. Therefore, we define the speedup function as follows:

$$G_k(m) = \begin{cases} \alpha_k(m-1)+1 & \text{for } 1 \le m \le m_k^t \\ \alpha_k(m_k^t - 1)+1 & \text{for } m > m_k^t, \end{cases} \tag{6.1}$$

SYMBOL	MEANING
α_k	Speedup increase rate
B_k	Buffer length of service class k
C	$(m^* - 1)\alpha + 1$
$G_k(m)$	Speedup of service class (or workload) k with m concurrent threads
K_k	$m_k^0 + B_k$
λ_k	Mean arrival rate for service class k
m_k^t	Saturation point for service class (or workload) k
m_k^0	Thread allocation for service class k
μ_k	Mean service rate for service class k
$\hat{\mu}_k(m)$	Maximum service rate for service class k with m concurrent threads
p_i	Probability of having i requests in the system
Q	Scheduling quantum
$\bar{Y}$	Mean processing time

Table 6.1: Explanation of commonly-used symbols. Note that subscript k is dropped for the single class case.

where α_k is the constant reflecting the efficiency of concurrency and m_k^t is the saturation point of service class k.

6.4 Analysis of MTRR Server

Analysis of an MTRR server requires an extension of some of the existing results from queueing theory, which did not model concurrency gain. We use a two-step process to develop our results. In the first step, we derive the expected performance of an MTRR server for a single service class. We then generalize our results to capture the interactions between n service classes. We also estimate the errors introduced by our model simplifications.

6.4.1 Case 1: Single Service Class

The analysis of the single service class MTRR is similar to the ones presented in [73, 75]. What is different here is the introduction of state-dependent service rates through the speedup function to include the effects of concurrency gains. We first consider an idealized model where the scheduling quantum is infinitesimal, i.e., $Q \to 0$. With this assumption, we are able to model the MTRR server using Continuous-Time Markov Chain (CTMC) [121]. Later we will estimate the error introduced

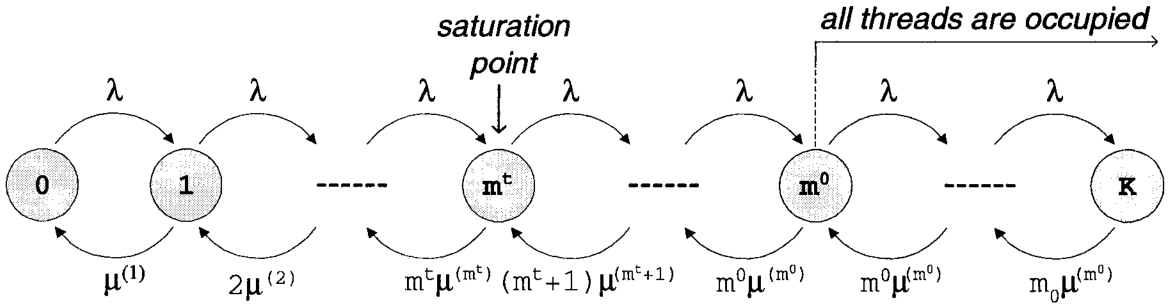


Figure 6.5: Markov Chain representation of MTRR server.

by this assumption.

Figure 6.5 shows the basic representation of the CTMC of the single-class MTRR server. The state here represents the number of requests in the system, and $\mu^{(i)}$ represents the state-dependent service rate, not the per-service class parameter, μ_k, described earlier. Therefore,

$$\mu^{(i)} = \frac{\mu}{i} G(i), \qquad (6.2)$$

where μ and $G(i)$ are the parameters describing the single service class under study. We drop the subscript k as there is only one service class.

We start by writing the steady-state probabilities for the CTMC, which are based on the local balance equations:

$$p_i = \begin{cases} \frac{\lambda}{i\mu^{(i)}} p_{i-1} & \text{for } i = 1, \ldots, m^0 \\ \frac{\lambda}{m^0 \mu^{(m^0)}} p_{i-1} & \text{for } i = m^0 + 1, \ldots, K, \end{cases} \qquad (6.3)$$

where m^0 is the number of allocated threads, and K is the maximum number of requests that can be admitted into the system, which includes requests in queue and in service. Specifically, $K = B + m^0$. Using Eqs. (6.1) and (6.2), we rewrite the probabilities as follows:

$$p_i = \begin{cases} \frac{\lambda}{[(i-1)\alpha+1]\mu} p_{i-1} & \text{for } i = 1, \ldots, \hat{m} \\ \frac{\lambda}{[(\hat{m}-1)\alpha+1]\mu} p_{i-1} & \text{for } i = \hat{m} + 1, \ldots, K, \end{cases} \qquad (6.4)$$

where α represents the speedup rate as described in Section 6.2 and $\hat{m} = min(\hat{m^t}, m^0)$. Notice the change of indicies in Eq. (6.4) from m^0 to $\hat{m}$ since $\mu^{(i)}$ remains unchanged for $i \geq \hat{m}$. Let us define $\rho = \frac{\lambda}{\mu}$ and also $\Psi_\alpha(i)$ as follows:

$$\Psi_\alpha(i) = \begin{cases} 1 & \text{for } i = 0 \\ \prod_{k=1}^{i}(k\alpha + 1) & \text{otherwise.} \end{cases} \qquad (6.5)$$

Now using simple substitution, we can rewrite each p_i as a function of p_0.

$$p_i = \begin{cases} \frac{\rho^i}{\Psi_\alpha(i-1)} p_0 & \text{for } i = 1, \ldots, \hat{m} \\ \frac{C^{\hat{m}}}{\Psi_\alpha(\hat{m}-1)} \left(\frac{\rho}{C}\right)^i p_0 & \text{for } i = \hat{m}+1, \ldots, K, \end{cases} \tag{6.6}$$

where $C = (\hat{m} - 1)\alpha + 1$. Since $\sum_{i=0}^{K} p_i = 1$, we can express p_0 as follows:

$$p_0 = \left[1 + \sum_{i=1}^{\hat{m}} \frac{\rho^i}{\Psi_\alpha(i-1)} + \frac{\rho^{\hat{m}} - \frac{\rho^K}{C^{K-\hat{m}}}}{\Psi_\alpha(\hat{m}-1)(\frac{C}{\rho} - 1)} \right]^{-1} . \tag{6.7}$$

Using these steady-state probabilities, one can numerically compute the expected number of requests in the system, $\overline{N} = \sum_{i=0}^{K} i.p_i$. Little's formula, $\overline{N} = \lambda(1 - p_K)\overline{W}$, can then be used to compute the total response time, $\overline{W}$, which includes both the queueing and processing delays. The term $(1 - p_K)$ is used to account for the probability that an arriving request will find a full queue and is dropped.

Given specific values for the system parameters, computing the various results is straightforward and can be achieved in $O(K)$ operations. This can be seen by observing that p_0 can be computed in $O(\hat{m})$ since $\Psi(i)$ does not need to be recomputed for every i, but rather, it can use the value from the previous iteration, i.e., $\Psi(i) = \Psi(i-1)(i\alpha + 1)$, for $i > 1$. Using Eq. (6.4) and the fact that p_0 need only be computed once, computing the response time can be done in $O(K)$.

Our formulation of the MTRR server along with the introduction of the speedup function constitutes a superset of several well-studied systems. For instance, when $\alpha = 0$, we observe no speedup. This reduces to a Generalized Processor Sharing (GPS) without priorities [71]. If we further add the restriction of $m^0 = 1$, then only a single thread is allowed to run. The system is further reduced to $M/M/1/B$ server. Finally, if $\alpha = 1$, it implies ideal speedup or, effectively, m^0 servers running in parallel. The system then becomes $M/M/m^0/B$.

Comparison with Discrete Quantum Values

The development so far assumed an idealized case of $Q \to 0$. Here we want to give a general idea of the expected error that is introduced by this assumption. For simplicity, we consider the worst-case scenario where the service is heavily-loaded, i.e., m is always equal to m^0. We also assume no speedup, i.e., $G(m) = 1$. When $Q \to 0$, the mean processing time, $\overline{Y}$, is just

$$\overline{Y} = \frac{m^0}{\mu}. \tag{6.8}$$

Now, let Q be a positive real value—typical values are 0.01 sec. We want to derive an approximate expression for $\overline{Y}$. We consider the processing of a request by one of the m^0 threads that always run during any service round (assumption A4). Let X be an exponential random variable reflecting the service time of an arriving request. Upon admission of the request into service at the beginning of a service round, its corresponding thread must first wait for its service turn before it starts execution. When the thread is scheduled, if it completes servicing the request in less than a time quantum (i.e., $X \leq Q$), its processing time is just the sum of X and the queueing delay before it starts service. On the other hand, if $X > Q$, then we expect that after Q seconds, the remaining threads must run before the beginning of the next service round, where the given thread must wait for its turn to run again. This process repeats until the request is finished.

The time that a thread must wait for m other threads to be serviced, either before or after it is scheduled, can be computed as follows:

$$\begin{aligned} E[V|m] &= E[\sum_{i=1}^{m} min(X,Q)] = m.E[min(X,Q)] \\ &= m\left(\int_0^Q xf_X(x)dx + \int_Q^\infty Qf_X(x)dx\right) \\ &= m\frac{1-e^{-\mu Q}}{\mu}, \end{aligned} \tag{6.9}$$

where $f_X(x)$ is the probability density function (pdf) of X.

During each service round, we assume that the order of scheduling threads is completely random. That is, for any given thread, its probability of being scheduled at the k-th position is $1/m^0$. We can now compute $\overline{Y}$ using the so-called *regenerative formulation*:

$$\begin{aligned} \overline{Y} &= \sum_{k=0}^{m^0-1} \frac{1}{m^0}\left\{E[V|k] + \int_0^Q xf_X(x)dx + \int_Q^\infty \left(E[V|m^0-k-1] + \overline{Y} + Q\right) f_X(x)dx\right\} \\ &= \frac{1}{\mu}\left[\frac{m^0+1}{2} + \frac{m^0-1}{2}e^{-\mu Q}\right]. \end{aligned} \tag{6.10}$$

When $Q \to 0$ in Eq. (6.10), we see that the results are consistent with Eq. (6.8). Furthermore, the error between the two equations is

$$\begin{aligned} \%Error &= \frac{\frac{m^0}{\mu} - \frac{1}{\mu}\left[\frac{m^0+1}{2} + e^{-\mu Q}\frac{m^0-1}{2}\right]}{\frac{m^0}{\mu}} \\ &= \frac{m^0-1}{2m^0}\left[1 - e^{-\mu Q}\right]. \end{aligned} \tag{6.11}$$

In the case of the mixed workload, where $\mu = 50$ reqs/s, the expected error is approximately 19%. We emphasize, however, that this is a worse-case scenario. In our experiments, we found that our derivations are within 10% of real measurements for a wide range of configuration parameters.

We note that while using finite Q values to determine $\overline{Y}$ better approximates the real system behavior, it is only mathematically tractable when $G(m) = 1$. When $G(m) > 1$, this method incorrectly reduces the processing times as it underestimates the number of service rounds required for completing a single request.

6.4.2 Case 2: Multiple Service Classes

The basic operation of a multi-class MTRR server is similar to an MTRR server with a single service class. But there are n service classes, $\{S_1, S_2, \ldots, S_n\}$, and each service class has a separate workload parameters: $(\lambda_i, \mu_i, G_i(m))$. Furthermore, each service class S_i is configured with a fixed number of threads m_i^0 such that the sum of all m_i^0's does not exceed the system-wide thread limit m^{max} (Assumption A6). We, thus, would like to characterize the response time of incoming requests given arbitrary thread allocation and workload configurations.

With the introduction of multiple services into our model, the analysis must consider two interdependencies between service classes. The first, which we refer to as *direct interdependencies*, is due to thread time-share the system. The second, which is *indirect interdependencies*, is caused by the possible sharing of a bottleneck resource such as the disk. We start by looking at how these interdependencies affect the analysis, and hence the performance, of servers running multiple services.

Identifying Workload Interdependencies

The distinction between direct and indirect interdependencies is important in the analysis and control of multi-class servers. Direct interdependencies have predictable behavior that can be accurately captured by an analytical model. An ideal time-sharing system is a good example where a thread will run for its entire scheduling quantum, Q, without blocking. Unfortunately, this is seldom the case for web servers especially with the growing popularity of per-user customization. Therefore, when requests require many I/O operations, the bottleneck resource is shifted from the CPU to the memory or to the disk; it becomes much harder to predict the impact of one service class on the other ones.

In some cases, precise understanding of indirect interdependencies may not be necessary. This

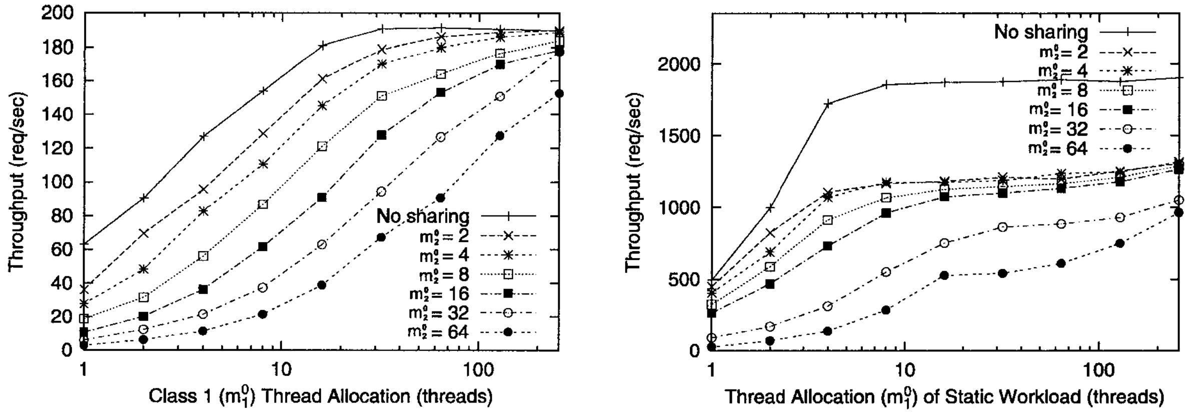


Figure 6.6: Workload interdependencies. (right) homogeneous workloads with both services are designated the mixed workload, and (left) heterogeneous workloads with one service designated the static workload and other designated the dynamic workload.

occurs when requests from different service classes have similar resource requirements. The load on all of the resources (including the bottleneck one) will, thus, be proportional to the number of requests that are being concurrently processed in each service class. We refer to these workloads as *homogeneous*. For example, a server that wants to provide client-side differentiation can be configured with several service classes, one for each group of clients. We, therefore, expect that these service classes will have similar service rates, μ_i, and speedup functions, $G_i(m)$, but with possibly different arrival rates, λ_i. Alternatively, when very different workloads need to be managed on the same system, e.g., a web server and an FTP server, each incoming request may have very different resource requirements. In this case, we refer to the workloads as *heterogeneous*.

We studied the multi-class server in the context of our workload categorization. Here we quantify the impact of increasing the concurrency (i.e., thread allocation) of one service class on the performance of the other running services. We base our comparison on the effects of changing the thread allocation rather than changing the arrival rate since the former provides a more direct bases for comparison. Studying workload interdependencies based on the change in the arrival rate would unnecessarily add a level of indirection as the arrival rate has to be translated into a specific resource requirement, which is expressed in our model by the number of concurrent threads.

We use a similar set up in Section 6.2, but now, we run two independent Apache services. Each service can be configured to receive requests for one of our three workloads: static, dynamic, an mixed. In particular, we had two test configurations. The first configuration reflects the homogeneous workload, where incoming requests to both Apache services are for the same workload;

the second configuration reflects the heterogeneous workload, where one service is designated one workload and the other service is designated a different workload. Finally, we measure the maximum throughput as a function of the number of threads that are allocated to each service class.

Figure 6.6 reflects the throughput gain as the thread allocation of the first service class is increased while the allocation of the second class is held constant. Each line represents a different allocation for the second service class; the "No sharing" line indicates that there is only a single class running on the server. We only show two test configurations in the figure. Other configurations of both homogeneous and heterogeneous workloads where tested and yielded consistent results.

The homogeneous (mixed) workload behaved as expected, where the throughput of a service class is proportional to its used thread. Specifically, we can express the service rate of any service class as a function of the number of the threads that are running:

$$\mu(m) \approx \frac{m}{\sum_{i=1}^{n} m_i} G(\sum_{i=1}^{n} m_i)\, \mu, \tag{6.12}$$

where m_i is the class-i threads that are running, or equivalently, the number of requests that are being concurrently processed by service class S_i.

The heterogeneous workload, on the other hand, did not exhibit the same behavior. Here we fixed the number of threads that are allocated to the server with the dynamic workload ($SRV_{dynamic}$ for short) and increased the thread allocation of the server with the static workload (SRV_{static} for short). Based on our measurements, we observed three unexpected phenomena:

P1. Even when $SRV_{dynamic}$ is assigned a single thread, its impact on the performance on SRV_{static} is significant. In fact, we observe an artificial ceiling that limited the maximum performance of the SRV_{static}.

P2. When the thread allocation of $SRV_{dynamic}$ is increased, but still below its saturation point ($\leq$ 16 threads), SRV_{static} incurs a small decrease in performance.

P3. After the thread allocation of $SRV_{dynamic}$ is increased beyond its saturation point, the SRV_{static} has a much greater performance drop. In both cases (P2 and P3), the performance drop is not proportional to the thread allocation of the two servers.

Because controlling the arrival rate will indirectly control the concurrency of running services, the above example shows an important result, namely, without precise understanding of the resource requirements of different—and heterogeneous—workloads, using rate-based controls to provide

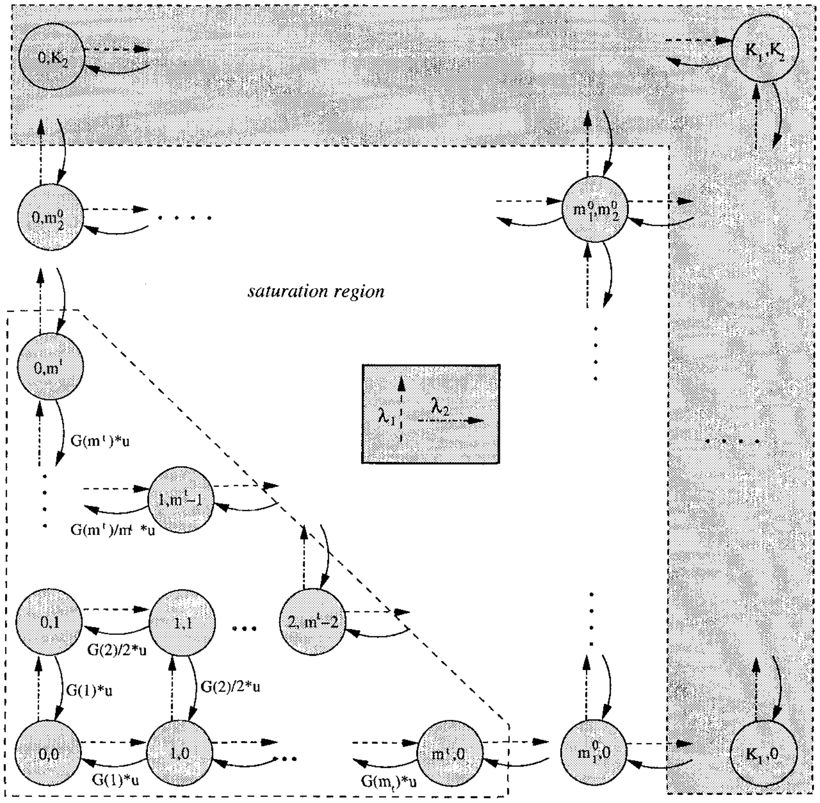


Figure 6.7: Example of CTMC for multi-class system with $n = 2$.

QoS differentiation is not an effective approach. Fine-grain resource management must be used to be able to provide effective QoS guarantees [14, 117, 120]. Unfortunately, these techniques require substantial changes to the application or the OS. We will, however, show that if services are configured with homogeneous workloads to provide client-side differentiation, then rate-based controls can be used as an effective tool.

Analyzing Homogeneous Workloads

Focusing on homogeneous workloads, we can characterize the system using a unique speedup function, i.e., $G_i(m) = G(m)$ for all i. We also expect that all service classes have equal service rates, i.e., $\mu_i = \mu$ for all i. As mentioned earlier, we assign $\mathbf{M} = \{m_1^0, m_2^0, \ldots, m_n^0\}$ as the maximum allowable number of threads for service classes $\{S_1, S_2, \ldots, S_n\}$. We also assign $\mathbf{L} = \{\lambda_1, \lambda_2, \ldots, \lambda_n\}$ as the mean arrival rate for each service class. Finally, each class is also assigned a waiting queue of size B_i, and thus, the maximum number of class-i requests that can be admitted into the server is $K_i = B_i + m_i^0$. We still assume that the time quantum, Q, is infinitesimal in order to simplify our derivation, i.e., $Q \rightarrow 0$.

We define the system state as $\pi = (r_1, r_2, \cdots, r_n)$ in which r_i is the number of class-i service requests in the server, which includes requests in the queue. We can, then, model our system as a CTMC similar to Section 6.4.1. Furthermore, the transition probability rates of the CTMC can be computed as follows:

1. For arrival of new requests:

$$p_{\pi=(\cdots,r_i,\cdots),\pi=(\cdots,r_i+1,\cdots)} = \lambda_i, \; i = 1, \ldots, \; n. \tag{6.13}$$

2. For departure of existing requests:

$$p_{\pi=(\cdots,r_i,\cdots),\pi=(\cdots,r_i-1,\cdots)} = \frac{\mu \hat{m}_i}{\sum_{j=0}^{n} \hat{m}_j} \cdot G(\sum_{j=0}^{n} \hat{m}_j), \tag{6.14}$$

where $\hat{m}_i = \min(r_i, \hat{m}_i^0)$. The first term here accounts for the processor sharing inside the MTRR server while the second term represents the speedup due to concurrency.

Figure 6.7 plots the CTMC for the case of two service classes. The states in the left-lower trapezoid represent that the system's speedup is not fully utilized, while the states in the right-upper rectangular imply that all allocated threads are occupied. The steady-state distribution of system state can be obtained from the transition matrix, P, by standard matrix operations. In fact, it is the eigenvector of P with the summation of each element equal to 1. However, the complexity increases rapidly with the number of service classes and possible states, K_i's.

In order to simplify our analysis and get a simpler expression for steady-state distribution of π, we truncate all the states with $r_i > m_i^0$ in Figure 6.7, which basically, eliminates the queue from each service. The motivation for this is based on our observation (shown later), where the expected number of requests inside the system is much less than m_i^0 for a wide range of loads given that m_i^0 is set high enough to take full advantage of the concurrency gain (e.g., when $m_i \geq 23$ threads). Thus, truncating those states should only introduce negligible error. Of course, this error will increase as the system becomes heavily-loaded. We address the analysis of heavily-loaded systems in the next section.

The structure of our Markov chain along with our simplification allow us to solve for the steady-state distribution of π by only solving the local balance equations:

$$P(\pi = (r_1, \cdots, r_i, \cdots, r_n))\frac{\mu r_i}{\gamma} G(\gamma) = P(\pi = (r_1, \cdots, r_i - 1, \cdots, r_n))\lambda_i. \tag{6.15}$$

where $\gamma = \sum_i r_i$. This can then be expressed in the following form:

$$P(\pi) = \frac{(\gamma!)\prod_i \lambda_i^{r_i} \cdot P(\pi = \mathbf{0})}{\prod_i (r_i! \mu^{r_i}) \prod_{k=1}^{\gamma} G(k)}. \tag{6.16}$$

Furthermore, the probability that the server is idle, $P(\pi = \mathbf{0})$, can be computed as follows:

$$P(\pi = \mathbf{0}) = \left[\sum_{\pi \neq \mathbf{0}} \frac{(\gamma!)\prod_i \lambda_i^{r_i}}{\prod_i (r_i! \mu^{r_i}) \prod_{k=1}^{\gamma} G(k)} \right]^{-1}. \tag{6.17}$$

Thus, instead of solving the eigenvector of a large matrix, the steady-state distribution can be solved readily by Eq. (6.16). Finally, the average response time of the class-i requests, $\overline{Y}_i$, can be computed by the Little's formula and Eq.(6.16) as

$$\overline{Y}_i = \frac{\sum_{r_i=1}^{m_i} (\sum_{\forall \pi = (\cdots, r_i, \cdots)} P(\pi)) \cdot r_i}{\lambda_i}. \tag{6.18}$$

6.5 Providing QoS Guarantees

Providing QoS guarantees is motivated by the need to protect certain—possibly high-priority—service classes from others that overload the server. As we have shown in Section 6.4.2, this is very difficult without explicit OS support, where strict resource limits are allocated to each service. In this section, we describe the extent at which rate-based control can be used to guarantee specific service delays. We continue to focus on homogeneous workloads; our proposed technique is aimed for providing QoS guarantees to different client groups, each represented by a separate service class.

The development in Section 6.4.2 has outlined the dependency between the performance of all service classes. It also derived equations for determining the expected response times for underloaded systems. In many cases, however, it is desirable to provide worst-case QoS guarantees. Here, the maximum arrival rate for each running service is constrained such that no matter how high it increases, it will not affect the QoS objective of the other—possibly protected—service classes.

The procedure for determining the rate limits requires two basic steps. In the first step, a table is constructed to reflect the performance of the system from the view point of a single service class, k. The table is two dimensional, where one dimension varies the maximum arrival rate of that service class, λ_k^{max}, and the second dimension varies the combined number of busy threads for all of the other service classes, $\gamma_k = \sum_{j=0, j\neq k}^{n}$. Because the system is shared by multiple classes, the idea here is to find the performance of class k when a certain portion of the system, represented by γ_k, is used by the other classes. Each entry in the table holds three values: the resulting response time,

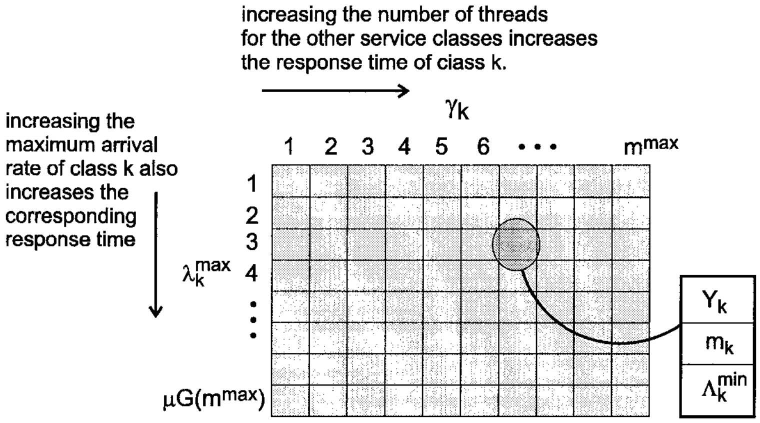

increasing the number of threads
for the other service classes increases
the response time of class k.
γk
1 2 3 4 5 6 ••• m^max
increasing the maximum arrival rate of class k also increases the corresponding response time
1
2
3
λk^max 4
μG(m^max)
Yk
mk
Λk^min

Figure 6.8: Rate table for service class k for finding the resulting response time, expected number of busy threads, and number of arrival rate for other service classes.

$\bar{Y}_k$, the corresponding number of running threads, m_k, and the minimum arrival rate for the other classes that can result in γ_k being busy, Λ_k^{min}. Figure 6.8 shows an example such table.

Computing each entry is relatively straight forward. Since we are assuming that γ_k threads of the other service classes are always busy, the multi-dimensional formulation in Section 6.4.2 will reduce to the single class case in Section 6.4.1, but with a different state-dependent transition rates μ_i. Going back to Figure 6.5, we can define μ_i as follows:

$$\mu^{(i)} = \frac{\mu}{i+\gamma_k} G(i+\gamma_k). \tag{6.19}$$

Here μ is the same for all service classes as the workload is assumed to be homogeneous. The ratio $\frac{\mu}{i+\gamma_k}$ reflects the fact that each thread has to share the system with $i + \gamma_k - 1$ other threads. The steady-state probabilities can be similarly defined:

$$p_i = \begin{cases} \frac{\lambda_k(i+\gamma_k)}{i\mu G(i+\gamma_k)} p_{i-1} & \text{for } i = 1, \ldots, m_k^0 \\ \frac{\lambda_k(m_k^0+\gamma_k)}{m_k^0 \mu G(\gamma_k + m_k^0)} p_{i-1} & \text{for } i = m_k^0 + 1, \ldots, K_k. \end{cases} \tag{6.20}$$

The remaining derivation is similar to the one in Section 6.4.1 and is omitted. The computed $\bar{Y}_k = (\sum_{i=0}^{K_k} i.p_i)/\lambda_k^{max}$ will, thus, reflect the worse-case response time when all γ_k threads are busy. Similarly, the expected number of busy threads for class k is given by the following:

$$m_k = \min(m_k^0, \bar{Y}_k \times \lambda_k^{max}).$$

Finally, the speedup function can be used to compute the last entry of the table, namely, the minimum rate that result in γ_k thread being busy. Specifically,

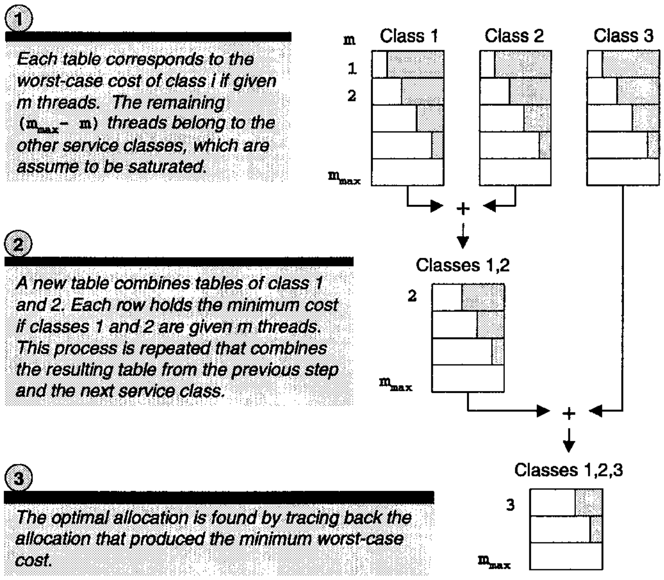
$$\Lambda_k^{min} = \mu G(m_k + \gamma_k) - \lambda_k^{max}$$

Once the table is constructed, the second step is to perform a series of lookups to determine the required rate limits. In general, there are two degrees of freedom: (1) the maximum allowed response time, and (2) the maximum allowed arrival rate, λ_k^{max}. For example, given a response time objective of 500 msec, λ_k^{max} may have a range of allowed values. Clearly, as λ_k^{max} is increased, the maximum arrival rate for the other service classes, Λ_k, decreases. Once both values are selected for class k, the remaining services are allocated the arrival rate, Λ_k^{min}, equal to the value in the third table entry. This value can be considered as the maximum arrival rate that must distributed among $n - 1$ service classes. A similar lookup to the one above is performed. However, because we have already constrained the value of class k, the table size is shrunk to a maximum arrival rate of Λ_k^{min} and a minimum thread allocation of m_k. This process is repeated until all services are allocated the appropriate rate limits.

It is important to note that this allocation must be enforced at all times to guarantee the desired QoS. This is the case even when some services have arrival rates that are below their allocated limit. If an alternate enforcement policy is used, where Λ^{max} is divided among incoming requests using instantaneous measurements (e.g., using WFQ), then it is possible for service queues for a non-preferred service to build up such that when the arrival rate for the preferred service increases, that service will not see the performance benefits of our rate allocation algorithm.

The above derivations shows that a rate-based scheme can only provide QoS guarantees when the total arrival rate is below the maximum allowed rate. The assumption was that the running services had an unconstrained number of threads. There are, however, situation when thread allocation can be easily manipulated. In this case, our model can be directly modified to find the thread allocation M that enforces the required QoS objectives.

Similarly, we consider a system that is heavily-loaded such that all classes are overloaded except for a single one, class k, here the term "overloaded" implies that all threads are continuously busy. The same technique as above can be used to determine $\bar{Y}_k$, the worst case response time of class k. The expression for $\bar{Y}_k$ can now be used to determine the thread allocation that can meet the desired QoS objective. We express the QoS here using the notion of holding cost. Formally, let $h_k(t)$ be the cost of a request in service class k as function of its response time t. Using $h_k(t)$ gives us flexibility in defining different QoS objectives. For example, it allows us to assign weights to service



Each table corresponds to the worst-case cost of class i if given m threads. The remaining (m_{max} - m) threads belong to the other service classes, which are assume to be saturated.

A new table combines tables of class 1 and 2. Each row holds the minimum cost if classes 1 and 2 are given m threads. This process is repeated that combines the resulting table from the previous step and the next service class.

The optimal allocation is found by tracing back the allocation that produced the minimum worst-case cost.

Figure 6.9: Dynamic programming algorithm for finding the allocation that minimizes the worst-case cost of the system. The illustration is limited to three service classes.

classes, which will result in providing preferential treatment to those classes with higher weights. The holding cost function can be arbitrary, however, with the restriction of being a monotonically non-decreasing function of t.[3]

We extend the notion of cost to any thread allocation $\mathbf{M}$. We first define the worst-case cost for service class k in $\mathbf{M}$ as $c_k(\mathbf{M})$. It is computed by assuming that all service classes except for class k are overloaded, and then

$$c_k(\mathbf{M}) = h_k(\bar{Y}_k),$$

where $\bar{Y}_k$ is just the worst-case response time as computed above. The sum of these costs $c(\mathbf{M}) = \sum_j c_j(\mathbf{M})$ is defined to be the cost for allocation $\mathbf{M}$. The allocation that minimizes the worst-case cost is thus

$$\mathbf{M}_{min} = \min_{\mathbf{M}}\{c(\mathbf{M})\}. \tag{6.21}$$

To efficiently compute $\mathbf{M}_{min}$, we first observe that our definition of γ_k in Eq. (6.19) does not distinguish between the different thread allocations to the overloaded service classes. This allow us to use dynamic programming to solve for $\mathbf{M}_{min}$, where in each step we group all overloaded classes together and then find the allocation that minimizes the cost of the non-overloaded classes.

[3] This restriction avoids the situation where requests with long response times have lower cost than those with short response times.

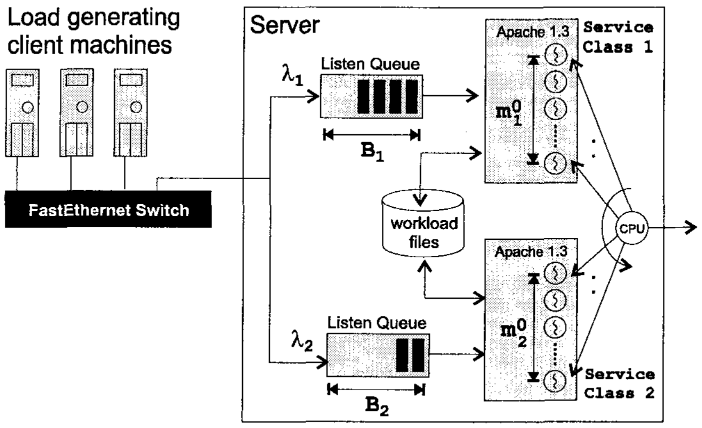


Figure 6.10: Experimental Setup.

The basic algorithm is outlined (graphically) in Figure 6.9, where the algorithm is divided into n steps. In the first step, n tables are created, one for each service class. Each table contains the expected cost for any thread allocation to its corresponding service class, given that all other service classes are overloaded. We only need m^{max} entries to capture this expected cost. The next step combines the tables for classes 1 and 2 into a new table by finding the minimum cost for each allocation given all possible combinations from the tables for classes 1 and 2. Each additional step then combines the resulting table from the previous step with the table of an additional service class. At the end, the final table will contain the minimum cost and by tracing back the allocation that produced it, M_{min} can be determined.

6.6 Evaluation

A realistic server environment is used to evaluate our proposed scheme. We would like to verify the correctness of our derivations with respect to our original assumptions and also demonstrate that the our allocation algorithm can correctly configure the rate limits. We used a similar experimental setup to that in Section 6.1. However, we configured a second Apache server as shown in Figure 6.10 to act as a separate service class. Three parameters describe each service: the arrival rate, λ_i, the listen queue length, B_i, and the thread allocation, m_i^0. In the presented experiments, we set $B_i = 128$ requests for $i = 1, 2$. We have evaluated the system for different buffer lengths; in all cases, our results were consistent with those presented here.

Our evaluation is based, primarily, on response-time measurements at the client machines. This response time is the total wait time before a request completes, and is the summation of three mostly

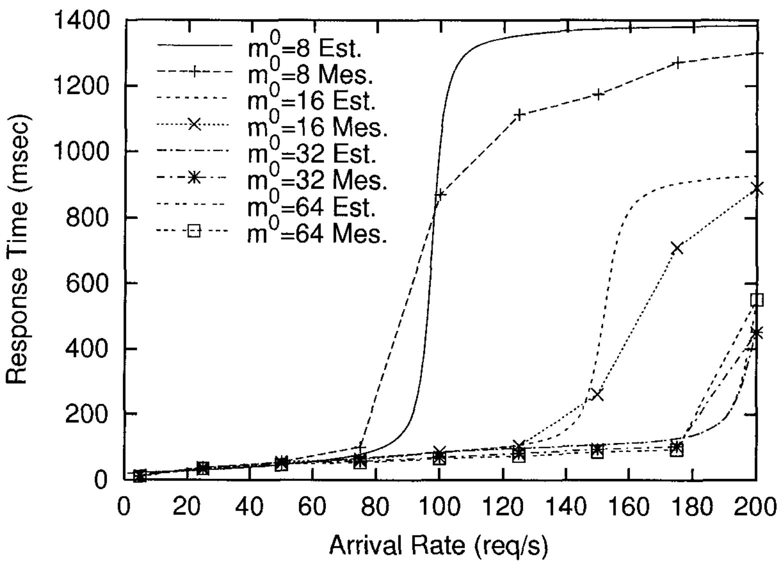


Figure 6.11: Single class measurements

independent components: connection-establishment latency, propagation delay, and processing delay. However, we are only interested in the effects of arrival rate (and thread allocation) on the processing-delay component. Thus, by keeping the first two components constant, we are able to obtain an unbiased view of the performance of the different rate allocations. We take two measures to minimize the variation in these two components. First, we made sure that the client-to-server network path is bottleneck-free. We also reduced the connection-establishment timeout such that any packet drop during that phase will not skew our results. This mimics a server that is using persistent dropping to control connection arrivals. We estimated that the error introduced by the first two components to be less than 2 msec. Finally, because we need to conduct a large number of experiments to cover the wide range of variable parameters, we limit each run to 5 minutes and each experiment was repeated 20 times.

To better reflect the behavior of real request arrivals, we configured requests to arrive following a Weibull distribution as measured in Chapter 3. This is different from our assumption of Poisson arrival, which we used to build our analytical models. Our evaluation is split into two experiments: the first validates the correctness and robustness of our derivation and the second measures the effectiveness of our optimal allocation policy. In all cases, we assume that workload is homogeneous, and hence, we only focus on the extent that the thread abstraction can be used to provide client-side QoS guarantees. The heterogeneous workloads were evaluated in Section 6.4.2. Finally, we only present the results for the mixed workload as it is considered a realistic representation of real server

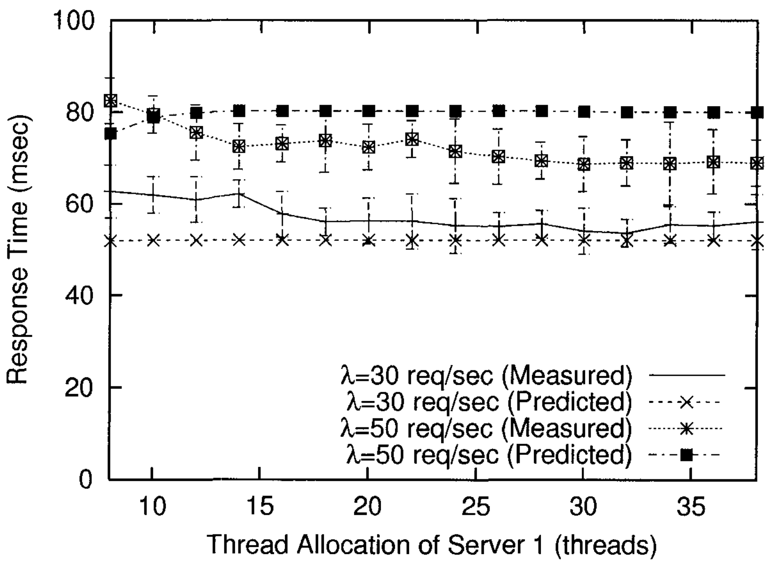


Figure 6.12: Multiple-class measurements

workloads.

6.6.1 Experiment 1: Model Validation

We compared our predicted values of the response time with the real measurements for the two analyzed cases: single- and multi-class server configurations. For the single class case, we used a single Apache service and varied the configuration parameters across two dimensions: arrival rate, λ, and thread allocation, m^0. This is shown in Figure 6.11, where each line represents the response time for a fixed allocation as the arrival rate is increased.

The figure shows that our derivations can accurately predict the expected performance of the underlying server for a wide range of configuration parameters. There is some over-prediction for some values of λ. This, we believe, is due to the system being critically-loaded. In particular, we can split the graph into two distinct regions: underloaded and overloaded. These are presented by the upper and lower parts of the S-shape of each line, respectively. The transition region between the underloaded and overloaded regions is very narrow and occurs when $\lambda \approx \mu G(m^0)$, where the system is critically-loaded.

Our analysis clearly explains the bimodal behavior of system queue occupancy. Namely, when the arrival rate is slightly below the saturation point, incoming requests are admitted almost immediately into service with little queueing delay. However, a slight increase in arrival rate can cause the delays to increase many folds simply because the system cannot keep up with incoming requests which causes queues to fill up. But since queues have limited capacity, the service delay is limited by the maximum length of such queues. The bimodal behavior raises an interesting design decision issue when configuring a web server, namely, when the system is underloaded, only a small queue is necessary to avoid request dropping. The length of the queue depends on the burstiness of arriving requests. However, once the system is overloaded, longer queues do not provide any performance advantage, but they increase the response time of accepted connections.

For the multi-class case, we used two Apache services to test the client-side QoS differentiation. Here we did not perform any intelligent classification to differentiate between incoming requests. Instead, we assigned each Apache service with a unique IP port and configured clients for each server to use the corresponding port number.[4]

When two services are used, our parameter space becomes four-dimensional. However, because our derivation is only applicable to the underloaded services, we focus on a narrower range of parameter values; the overloaded case is addressed in the next section. Figure 6.12 shows response time when the arrival rates for both classes are held constant and the allocation of the first service class is increased. As expected, because of the bimodal behavior of system queues, the response time remained relatively constant for the whole range. Moreover, the results still capture an important element for a multi-class service, i.e., identifying a realistic lower bound of response time for any thread allocation. In the next section, we look at the opposite view, i.e., an upper bound for QoS guarantees.

6.6.2 Experiment 2: QoS Guarantees

In Section 6.5, we described an algorithm to determine the rate allocation that can provide worst-case QoS guarantees. We showed two versions, where the first version used rate-limits to guarantee the necessary QoS objective and the second used thread-allocation to achieve slightly more flexible guarantees. In this subsection, we validate the efficacy of each version in providing QoS guarantees.

Rate-cased Controls: To show the effectiveness of our allocation algorithm, we used two Apache services as described earlier, but here we did not constrict their thread allocations. Specifically, each service is configured with the allowed maximum of 1000 worker threads.[5] We varied the arrival rates over two dimensions, where each dimension represents a separate service. For different

[4] In reality, more sophisticated techniques are used to distinguish between the different client groups (e.g., Level-7 switching).

[5] Configuring Apache with a higher number of threads would not change the results since the number of busy threads is much smaller that this maximum value.

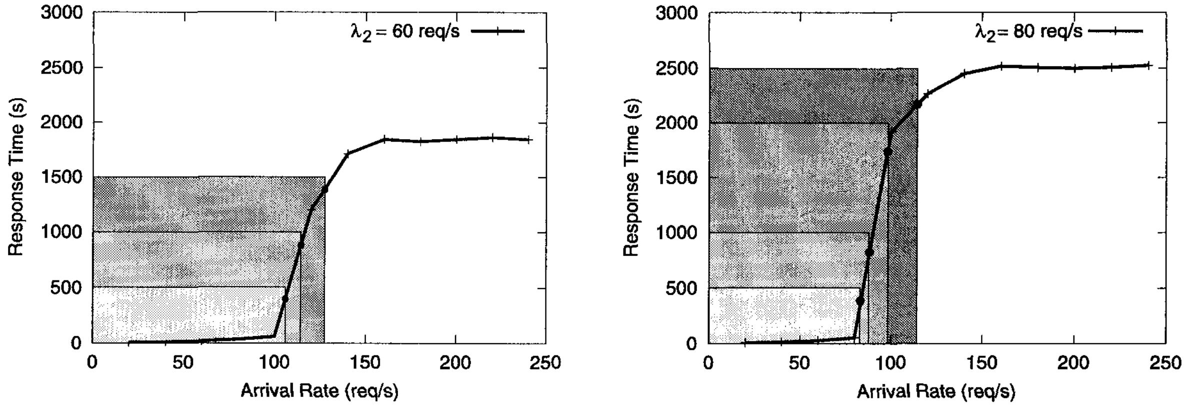


Figure 6.13: Rate-limits that provides response time guarantees for a high-priority service: (left) low-priority service is allowed 60 req/s, and (right) low-priority service is allowed 80 req/s.

QoS objectives, we then compared the rate limits that we obtain from our allocation algorithm with the corresponding values from the measurements.

Figure 6.13 compares the measured and predicted values for different QoS objectives and different λ_1^{max}, the maximum arrival rate of the first Apache service. The shaded boxes reflect the rate limits that should be enforced to guarantee the required response times. The point where a vertical line of a shaded box crosses the measured line reflects the value for the real worst-case response time. In Figure 6.13(right), for instance, a 500 msec response time requires a rate limit of 80 req/s. This translates to a 430 msec actual worse-case response time. Overall, the figure shows that our algorithm is relatively accurate at determining the necessary rate limits.

Thread-based Controls: To fully verify the correctness of the thread-allocation algorithm, we must, unfortunately, test all possible thread allocations, which is computationally-prohibitive even with only two service classes. Here, we look only at a single step of the algorithm, namely, given a thread allocation for a low-priority service class, we want to predict the thread allocation for a high-priority service class that can (statistically) guarantee a maximum response time. We show that for each QoS requirement, our predictions are close to the measured values. With this, we can conclude the robustness of our algorithm in the general case.

Figure 6.14 shows the required number of threads for the high-priority service when its arrival rate is $\lambda_1 = 100$ reqs/s and the low-priority service is allocated a fixed number of threads. For instance, in the top plot where the low-priority service is allocated 8 threads, if a 1 second delay guarantee is required, then the high-priority service should be allocated at least 10 threads. The

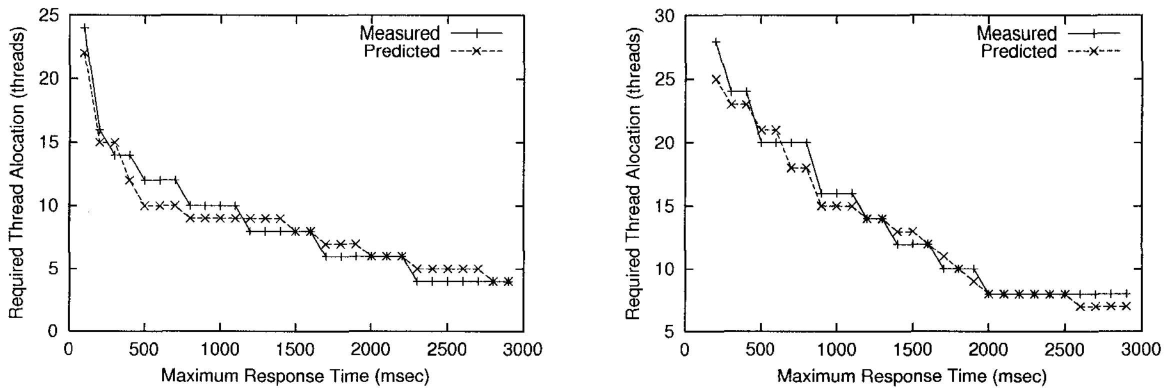


Figure 6.14: Thread allocation that provides response time guarantees for a high-priority service: (left) low-priority service is allocated 8 threads, and (right) low-priority service is allocated 16 threads.

figure (for the measured and predicted lines) is computed by first assuming that the low-priority service is overloaded. A table that holds the thread allocation vs. worst-case response time for the high-priority service is then created. Finally, an inverse table lookup is used to determine the minimum allocation that meets the response time requirement.

The figure shows that our equation-based optimization is able to predict, with high accuracy, the thread allocation that achieves the minimum cost. One can see that if a similar process is used to create the initial tables in Section 6.5, then the resulting prediction will be close to the optimal value. We note that in this experiment we implicitly assumed a linear cost function where $h_k(t) = t$. Other cost functions can still be used.

Overall, the above results show that our models are very robust. They capture the expected performance of a multi-threaded server as well as identify those instances where the model fails. Our approach can be used to improve the performance of existing QoS techniques.

6.7 Conclusions

In this chapter, we have provided a formal analysis of the performance of rate-based QoS differentiation. We derived computationally-efficient equations that describe the expected performance of single-staged multi-class services. We also presented an efficient optimization algorithm for determining the rate limits—if they exist—that guarantees a maximum response-time QoS objective. Through empirical validation in real server environments, we showed that the derived results are applicable to real-world systems.

The results presented in this chapter are essential to the design of any efficient rate-based QoS differentiation mechanism. Three important conclusions can be drawn from our study. First, based on the shape of the speedup function, we argue that dynamic adaptation based on the response-time measurements only is not sufficient to guarantee the stability of the control mechanism. The controller must continuously monitor the saturation point, which may shift with changing workloads. Second, indirect interdependencies between services that arise from non-trivial sharing of system's resources can yield unpredictable performance interactions. We have shown that even with a small arrival rate dedicated to I/O-heavy workloads, the performance of other running services can be affected significantly. Therefore, without accurate understanding of resource requirements, rate-based mechanisms alone cannot provide the necessary QoS guarantees, or even QoS differentiation, to running services. Finally, when similar or homogeneous services are being hosted on a single server to provide client-side differentiation, the rate-based mechanisms can be used to provide effective and predictable statistical QoS guarantees.

CHAPTER 7

ADAPTIVE PACKET FILTERS

The previous chapters have established fundamental results for designing effective NOSCs. Individually, each chapter has looked at specific sub-problem and a proposed specific solutions for dealing with it. This chapter presents a general architecture for implementing different NOSC designs. The proposed architecture focuses on three design goals: flexibility, extensibility, and low overhead.

7.1 Introduction

Sudden and drastic changes in demand, content, and service offerings are characteristic of the Internet environment. For many web-based Internet services, peak demands can be 100 times greater than average load [116]. Over-design alone cannot realistically absorb such surge; if servers are not prepared to deal with extreme load, their Quality-of-Service (QoS) will seriously suffer. Many servers collapse under overload, an unacceptable option as clients increasingly depend on the availability of Internet servers.

In this chapter, we propose Adaptive Packet Filters (APFs), a highly-customizable architecture for realizing different NOSCs implementations. An APF allows arbitrary number of packet filtering rules to be loaded into a *filter controller* and accepts feedback from appropriate load monitoring scripts or modules, where a rule basically defines how incoming traffic should be classified and enforced (similar to what we proposed in Chapters 4 and 5). Depending on the configured filters and monitoring inputs, the APF will automatically enforce filters that avoid overload. APF can be introduced into existing server infrastructures in two different flavors: backend- or frontend-based overload protection. In the former deployment scenario, APF resides on the servers and filters

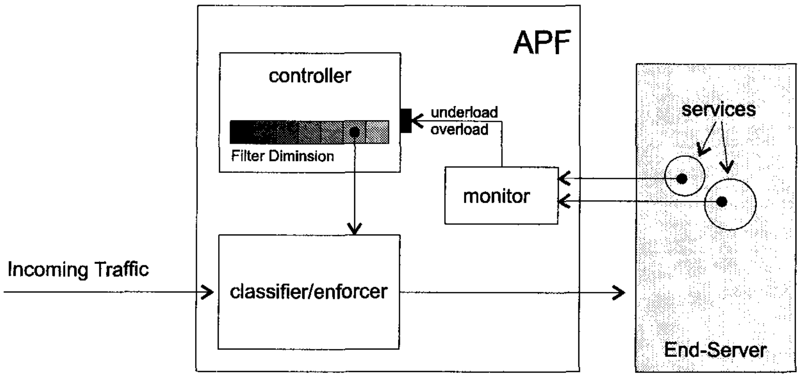


Figure 7.1: APF Architecture

incoming packets based on local load information. In the front-end scenario, an APF device routes traffic to backend devices while throttling packet flows in response to load measurements on the back ends.

APFs can be seen as an enhancement of the classic network-based approaches [43, 55, 71] by allowing arbitrary load-inputs—not just local queue lengths—to affect traffic policing. In fact, we show that fair-queuing-like QoS differentiation can be built easily using the new APF abstraction. Alternatively, one may view APFs as an extension to firewalling [92, 108] since they, too, can be configured to enforce arbitrary packet-filtering policies. What distinguishes APFs from traditional firewalls is that they react to configurable load inputs by dynamically enforcing more or less restrictive packet filters (see Figure 7.1).

APFs are designed with adaptation in mind: APFs can learn the server's (or cluster's) request-handling capacity autonomously and divide this capacity among clients and services as defined in APF rules. APFs' differential treatment of incoming traffic protects servers of arbitrary capacity from overload and sudden request spikes. This allows service providers to increase their capacities gradually as demand grows. Implementing APFs in server OS's and/or router OS's eliminates the need for overdesign in anticipation of transient request floods.

This chapter is organized as follows. We present our design rationale and challenges in Section 7.2 and discuss its implementation in Section 7.4. Section 7.5 studies APF's behavior in a number of typical server overload scenarios. The chapter ends with concluding remarks in Section 7.6.

7.2 Overall Design

The design of APFs can be divided into two main parts. The first describes the component that are used to enforce QoS differentiation and autonomic adaptation to overload, and how they interact with each other. The second describes the configuration of each component that achieves its design objective. We are thus interested in answering how filters should be designed to best control incoming requests and how adaptation should be performed to best match the behavior of running services. In this section, we focus on the first part; the next section builds on results from previous chapters to emphasize the design choices that we made to maximize the performance of APFs.

APFs require basic packet classification where each packet, once classified, will be treated according to a policy associated with its traffic class. Traffic classes are defined by information contained in the packet headers and can represent both server-side applications or client groups. For instance, server-side applications can define traffic classes by matching IP destinations and destination ports. Alternatively, client populations can be defined by specifying sets of IP source address prefixes. Traffic classes can further refine incoming traffic by, for example, looking at packet types such as such as, TCP, TCP-SYN, UDP, IPX, and ICMP. They can also integrate with existing network QoS architectures such as DiffServ when defining traffic classes based on incoming packets' code. Access to the HTTP service can, thus, be captured by configuring the frontend or the server's own packet filtering layer to match packets to the HTTP service's IP with destination port number 80 and the TCP-SYN flag set.

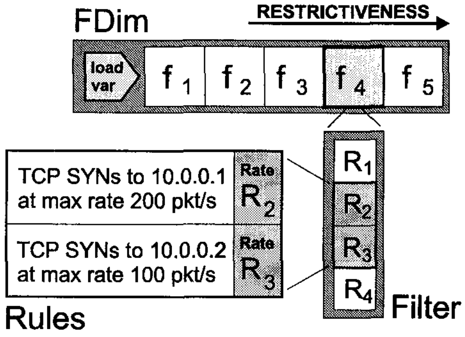
The notion of traffic classes is originally used in firewall configuration and was proposed by Mogul *et al.* [92]. A *rule* combines a traffic class and a policy, where the policy specifies whether packets of the traffic class should be accepted, dropped, rejected, or shaped to a particular rate. Our implementation of ICMP reject in Appendix A or persistent dropping in Chapter 4 are examples of two traffic policies. If multiple rules match an incoming packet, the most restrictive policy will be applied to the packet. This definition of rules is consistent with that of most modern firewall implementations. Therefore, each individual rule defines QoS requirements for exactly one traffic class. Thus, QoS differentiation between competing traffic classes can only be achieved by installing rule combinations. A *filter* represents a rule combination. For example, a filter of two rules could be set up to admit twice the packet rate from IP address X as from IP address Y. Furthermore, to ensure QoS differentiation, the rules of a filter are enforced as a unit, i.e., all or none of its rules are enforced.

One can view filters as a method for specifying explicit traffic controls. Tuning such filters to match the corresponding server's true processing capacity is not a straightforward task as shown in Chapter 6. Inadequate control policies can easily cause underutilization of network servers or can degrade the effectiveness at protecting the server. For example, if one restricts incoming traffic to a moderately-loaded Internet server, one risks dropping requests which the server could have handled easily. However, installing only permissive filters to accommodate the common (lightly-loaded) case would be a naive conclusion. Too permissive filters will fail to defend the server from load surges, and therefore, filters must be chosen *adaptively*.

We introduce *filtering dimensions* (FDims) to allow for adaptive packet filter selection. Each FDim is a linear arrangement of filters, $(f_1, f_2, \ldots, f_n)$. Only one filter of an FDim is enforced at a time. A switch from one filter to another is an *atomic* operation, meaning that the old filter is uninstalled and the new filter is installed without processing any network packets in between. Figure 7.2 shows an expanded view of an FDim. As shown, a load variable is associated with each FDim to drive the filter selection process. It is fed by external load-monitoring software and only read by the APF.

To provide effective overload protection, each FDim configuration must satisfy the following constraints: (1) filters are ordered by increasing restrictiveness, (2) the least restrictive filter does not police incoming traffic at all, and (3) the most restrictive filter drops all incoming traffic. Multiple FDims may be installed simultaneously, each tied to its own load variable. This pays tribute to the fact that different types of overload may be caused by different network services, which must be policed separately. Support for multiple FDims is thus particularly useful when an APF-enabled device controls accesss to different services, each of which is located on its own separate backend server. It is important to note that since different FDims are functionally independent, their corresponding load variables must be tied to *independent* load events. Independence here is important to avoid unpredictable interactions between multiple FDims, leading to possibly inaccurate load enforcements. An example of such interactions was discussed in Chapter 6 and will be revisited in Section 7.5.

APFs are conceptually similar to CBQ [55]. There are, however, three key differences between the two. First, APFs are inbound controls. Therefore, they remain effective QoS controls even if they are only installed on the network servers. Second, APFs are not tied to any particular link scheduling or packet scheduling architecture. Finally, APFs may police incoming packets based on load measurements other than just link utilization. CBQ cannot adapt incoming packet rates to



FDim
RESTRICTIVENESS
load var
f 1
f 2
f 3
f 4
f 5
TCP SYNs to 10.0.0.1 at max rate 200 pkt/s
Rate R 2
TCP SYNs to 10.0.0.2 at max rate 100 pkt/s
Rate R 3
Rules
R1
R2
R3
R4
Filter

Figure 7.2: Relationship between rules, filters, and FDim

match the network servers' capacities.

7.3 Effective Configuration of APFs

In this subsection we look at how APF components can be configured to maximize its effectiveness. Specifically, we look at four design decisions: (1) how traffic filters should be designed to better deal with the intrinsic behavior of arriving traffic, (2) how the adaptation mechanism should be fine-tuned to deal with the possible interactions between running services, (3) what constitutes good load variables, and (4) how can QoS differentiation be enforced?

7.3.1 Traffic Enforcement

As shown in Chapters 4 and 5, the traffic enforcement mechanism can have a significant impact on the behavior of arriving requests due to the persistent nature of incoming requests. In particular, the re-synchronization of dropped requests can (and in our experience often does) reduce the stability of the underlying control mechanisms. For instance, referring back to Figure 5.1, one can see that a temporary burst may induce an oscillatory behavior in request arrival, which may in turn cause an oscillation in the traffic adaptation mechanism. That is, during peak arrival, the adaptation mechanism may move to a more restrictive policy (or filter) to avoid overload, and vice versa.

It is, thus, necessary for our adaptation mechanism to appropriately account for this possible behavior. This considerably complicates the design of the adaptation controller as it must now account for the possible correlation between the applied control (i.e., traffic filter) and future request arrivals. This is further complicated by the fact that incoming requests are inherently bursty. Without a steady arrival distribution (which is seldom the case), it is unclear (from a traditional control optimization perspective) how the stability of the underlying controller can be verified.[1]

Instead of complicating the design of the controller, we choose to take advantage of the results in Chapter 4 where we have shown that persistent dropping can virtually eliminate the effects of clients' persistence from incoming traffic. Since PD is already designed as a low-level filtering mechanism, it fits well with the overall architecture of APF. Specifically, we use PD to decouple the applied control from future arrival (this is from the perspective of the adaptation mechanism). Therefore, each time a new filter is installed, it is effectively changing the drop probability of the persistent dropping filter that is assigned to each traffic class without erasing the state of the previous filter.

It is important to note that the filtering mechanism in APF is not limited to PD. APF can be configured with different mechanisms that can be designed to handle different types of traffic. This flexibility is important to better handle the changing nature of Internet services. However, for the remainder of the chapter, we assume that traffic filtering is engineered around the persistent dropping mechanism.

7.3.2 Adaptation to Overload

FDims is a basic abstraction that allows adaptive filter choices with respect to overload. In the APF framework, QoS differentiation and overload defense depend on the controller's ability to determine the best filter for incoming traffic. The following control mechanism automatically determines the least restrictive filter in each FDim that prevents their respective monitoring variables from reporting overload.

1. Load measurements are integral numbers between 0 and 100, with 0 representing no load and 100 representing the highest load.

2. The APF associates configurable thresholds with each load variable as to what constitutes an overload and an underload.

3. When an overload is detected on some load variable, v_i, the next most restrictive filter for its corresponding FDim, F_i, is applied. If this change triggers the application of the most restrictive filter for that FDim, the FDim is flagged as severely overloaded. The network administrator can retrieve this flag for reconfiguration purposes.

[1] We note here that similar to any control mechanism, stability is an important, if not the pivotal, issue in the design of effective (and predictable) controllers.

4. When an underload is detected on load variable, v_i, the next less restrictive filter for its corresponding FDim, F_i, is applied.

5. The APF is configured with a configurable inter-switching delay, s_i, for each FDim, F_i to stabilize adaptation behavior. The APF keeps track of how long each filter was applied against incoming load. This is an indication of a filter's effectiveness, and therefore, important for system management.

The above describes the overall adaptation mechanism. There are still two issues that must be resolved to optimize the overall server performance subject to the "no-overload" constraint.

Issue 1: Determining the Appropriate Switching Time

The filter switching delay, s, is possibly the most important design parameter that affects both the stability and convergence of the underlying control mechanism. Fundamentally, the value of s should be at least the average waiting time of an arriving request, which includes both the queueing delay and service time. This is because if the APF controller switches to a more restrictive filter, all the monitoring information is incorrect until the residual effects of the previous filter is dissipated. Fast switching may thus cause large oscillations, reducing the stability and effectiveness of the control mechanism.

One method for determining s is to use load profiling as described in Chapter 6. This would thus require off-line analysis, which in some cases is not an attractive solution to system administrators. APF, thus, tries to estimate s dynamically based on the oscillatory behavior of the FDims. We note though that s can always be manually configured when off-line capacity analysis is available.

The dynamic configuration of s begins by assuming a minimal switching delay (our implementation assumes a 2 sec delay, which seemed reasonable based on current server capacity). As required by the APF specification, our implementation signals whenever the most restrictive filter of an FDim is installed. We observed in our initial implementation that these signals occurred frequently with long-lived requests. As mentioned before, this is due to the fact that the effects of a previous overload are still present as the controller decides to switch to the next most restrictive filter. Residual load results from the servers' own request queuing mechanisms (which can be both in the kernel- and application-level queues). With potentially very large combined queues, if it takes a very long time to serve each request, it will take very long to drain the server of residual load. To solve the residual load problem, we adapted our implementation by having the controller learn the time, t, that it takes for the load variable to indicate the first underload after the onset of an overload. This time t indicates how much an FDim's load variable lags behind the selection of a new filter. So, instead of strictly enforcing the minimal switching delay to be s, it is adjusted dynamically to $\max(s, t)$. We did not overwrite the value of s with t to avoid the possibility of very short switching times that may increase the overhead of APFs due to frequent filter switching. Thus, in systems where the effects of filter switches are delayed significantly, the load-controller waits for a longer time before increasing (or decreasing) the restrictiveness of inbound filters which completely eliminated the difficulties in handling residual server load. Furthermore, in systems where the effects of filter switching are immediate, the minimal switching time, s, is used to reduce the operational overhead of APFs.

7.3.3 Issue 2: Convergence and Minimizing Filter Dimensions

As expected from any control mechanism, APFs will eventually begin to oscillate between the application of a few adjacent filters because the policies of one filter are too restrictive, and too loose in the other. This behavior is a natural consequence of the controller's design and is non-problematic unless the controller oscillates between vastly different filters. In this case, the FDim is configured too coarse-grained and needs to be refined.

Using very fine-grained FDims for accurate QoS differentiation is not without problems either. It requires an accurate estimate of the server's request processing capacity. Otherwise, fine-grained filters would require large FDims to cover a large range of possible server request processing capacities. Unfortunately, such large FDims converge slowly towards filters that protect the server from overload. We tackle this trade-off between fine-grained control and fast convergence in the design of an automated FDims generation and reconfiguration called the *FDim-Manger (FM)*. The FM is a user-level management tool that is also responsible for generating and configuring FDims. The FDim-Manager speeds up response to sudden overload by keeping only a fixed, small number of filters per FDim. At the same time, it provides fine-grained rate control by dynamically reducing the grain around the FDim's most frequently-used filter. This filter, called the *focal point*, is estimated by computing a weighted average of all filters' shaping rates with their respective weights being the time for which they were enforced by the APF.

Whether or not the FM uses a finer filter quantization around the focal point depends on the load-controller's stability (absence of oscillations covering more than 20% of all filters). When oscillations are large, indicative of a shift of the focal point, the FM automatically shifts to a coarser quantization grain. This speeds up the process of finding the new focal point. Focal point shifts are generally the result of a change in the servers' per-request processing requirements. A family of *compressor functions* [112] is used to adapt the FM's grain.

7.3.4 Load Monitoring

At the core of APFs is load monitoring. It is simply the trigger to any adaptation: If the load on the system is high, APF will move to a more restrictive policy, and vice versa. There are many challenges to providing accurate and fast monitoring of server load. This is due to the fact that server load consists of many components that are often difficult to measure without major modifications to applications and OSs. Take, for instance, the listen queue length as an indication of server load. It is often argued that listen queues resembles wait queues in queueing systems and can thus be a good indication of server load (i.e., when the queue is building up, the system is experiencing high load). As it was shown in Chapter 6, this is not necessarily the case for multi-threaded services—the prevailing service model—since threads act as a virtual application-level queue and the listen queue is only the tail-end of this virtual queue. Thus, if a service is configured with a large number of threads (which is often the case), then a build up in the listen queue may only occur well after the server is severely overloaded.

The above example is meant to emphasize that even the most obvious and ubiquitously used load indicators are alone inadequate to measure the server load. For this, we have designed APFs to accept a collection of load indicators. Such indicators are combined in a fuzzy-logic style to provide a general sense of how well the server is performing. This was found to give us great flexibility to explore a variety of different monitoring variables. In Section 7.4, we discuss in more detail the load indicators that we implemented.

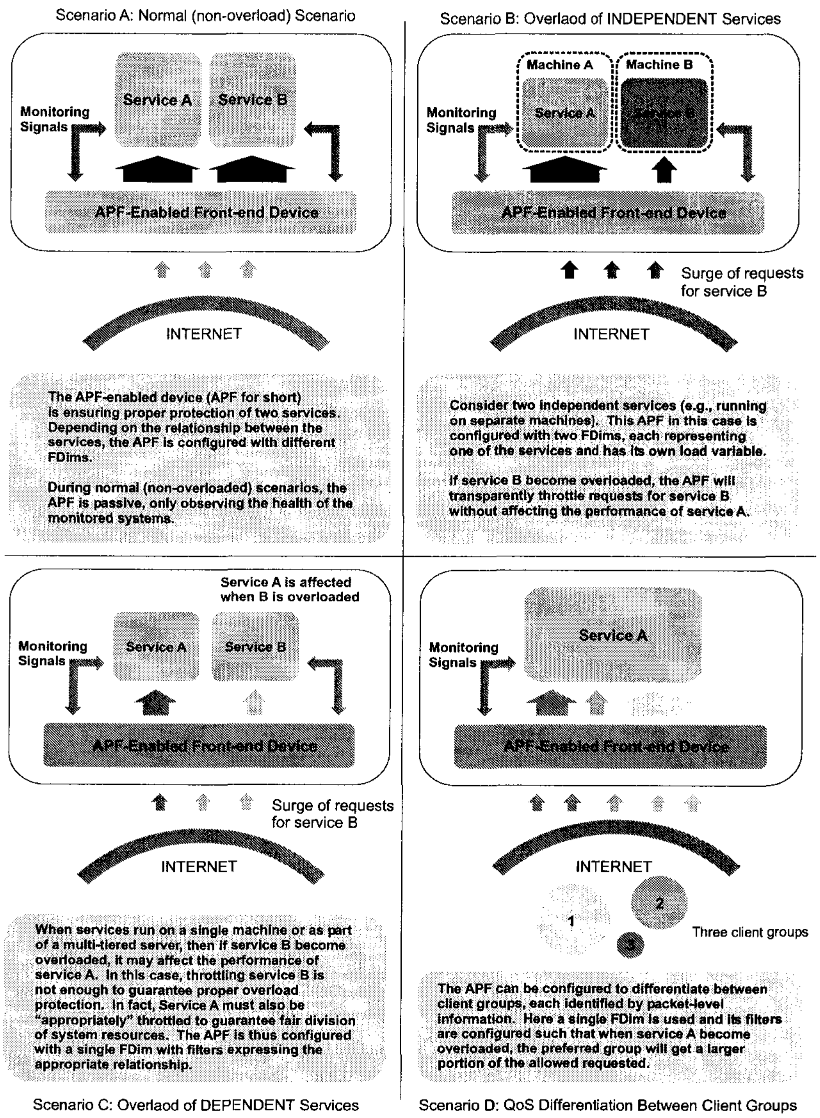
Finally, when defining monitoring variables or indicator, one must pay attention to the interaction between the monitor and the controller. In particular, the recognition lag (i.e., the time between the onset of the overload condition and the time at which the overload manifests itself in the load-indicator) is crucial. If the recognition lag is too long, the controller will react too late to an overload and may cause instability. A short recognition lag, as achieved by basing the monitoring input on short observation histories, triggers fast response to overload but causes larger oscillations between filters.

7.3.5 Providing QoS Differentiation

As shown in Chapter 6, one can use a network-oriented mechanism like APF to provide QoS differentiation to "homogeneous" services or client groups. In such a case, one can use a request allocation mechanism that is akin to Weighted Fair Queueing. In particular, the FDims created by the FM are fair in terms of max-min fair-share resource allocation [71]. The FDims (which consists of a series of increasingly restrictive filters) is configured by specifying the traffic classes, e.g., SYN packets to port 80, and rate ranges, like 10–100 pkts/s. In addition to these ranges, an integer weight is assigned to each traffic class. This weight determines how excess capacity—left-over bandwidth after each traffic class' minimum capacity requirement has been satisfied—is distributed among competing traffic classes. Note that each instance of the FM configures only one FDim. Our results, presented in Section 7.5, show benefits for the APF's fair allocation of system resources.

As part of providing QoS support, the APF can protect from certain Denial-of-Service (DoS) attacks with the help of the FM, presupposing they are measured (see Table 7.1). In this case, the APF signals overloads that cannot be tackled with the installed FDim. Then, the FM identifies ill-behaved traffic classes, and reconfigures the FDims to deny server access to those ill-behaved traffic classes. The ill-behaved traffic class(es) is (are) identified by first denying all incoming requests until the server overload subsides. Subsequently, traffic classes are admitted one-by-one on a probational basis in weight order. All traffic classes that do not trigger another overload are admitted to the server. All other traffic classes are considered ill-behaved and tuned out for a configurable period of time. This feature improved the robustness of APFs' QoS differentiation mechanism significantly. A different implementation may be appropriate if one has an exact estimate of each traffic class' contribution to server load. We did not make such an assumption.

With the aforementioned architecture and capabilities of APFs, there are several deployment configurations for APF, depending on the interdependency of running services and QoS objective (here we include overload protection as providing a form of QoS support to the server). Figure 7.3 shows three possible configurations in Scenarios B, C, and D. It also illustrates the operational transparency of APFs under normal load conditions in Scenario A. These scenarios are closely studied in Section 7.5.


Scenario A: Normal (non-overload) Scenario
Monitoring Signals
Service A
Service B
APF-Enabled Front-end Device
INTERNET
Scenario B: Overlaod of INDEPENDENT Services
Machine A
Machine B
Monitoring Signals
Service A
Service B
APF-Enabled Front-end Device
Surge of requests for service B
INTERNET
Service A is affected when B is overloaded
Monitoring Signals
Service A
Service B
APF-Enabled Front-end Device
Surge of requests for service B
INTERNET
Scenario C: Overlaod of DEPENDENT Services
Monitoring Signals
Service A
APF-Enabled Front-end Device
INTERNET
1
2
3
Three client groups
Scenario D: QoS Differentiation Between Client Groups

The APF-enabled device (APF for short) is ensuring proper protection of two services. Depending on the relationship between the services, the APF is configured with different FDims.

During normal (non-overloaded) scenarios, the APF is passive, only observing the health of the monitored systems.

Consider two independent services (e.g., running on separate machines). This APF in this case is configured with two FDims, each representing one of the services and has its own load variable.

If service B become overloaded, the APF will transparently throttle requests for service B without affecting the performance of service A.

When services run on a single machine or as part of a multi-tiered server, then if service B become overloaded, it may affect the performance of service A. In this case, throttling service B is not enough to guarantee proper overload protection. In fact, Service A must also be "appropriately" throttled to guarantee fair division of system resources. The APF is thus configured with a single FDim with filters expressing the appropriate relationship.

The APF can be configured to differentiate between client groups, each identified by packet-level information. Here a single FDim is used and its filters are configured such that when service A become overloaded, the preferred group will get a larger portion of the allowed requested.

Figure 7.3: APF Deployment Example

7.4 Prototype Implementation

We implemented an APF prototype as an extension module, called *QGuard*, for the Linux 2.2.14 kernel's firewalling layer `ipchains`. `Ipchains` provides efficient packet header matching functionality and simple packet rejection policies. We implemented additional support for traffic shaping. The Linux Kernel 2.4.0 firewalling support, *netfilters*, already implements this policy. We kept `ipchains` internal data structures of active firewalling rules intact. Filters and FDims are completely embedded in our module. Via a special system call, a user can install a specific filter (a `NULL`-terminated list of firewalling rules) for a specified filtering dimension. If the filtering

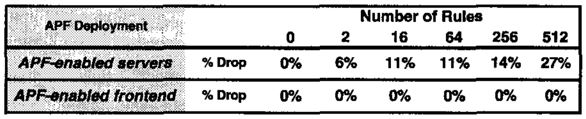
Indicator	Meaning
High paging rate	Incoming requests cause high memory consumption, thus severely limiting system performance through paging.
High disk access rate	Incoming requests operate on a dataset that is too large to fit into the file cache.
Little idle time	Incoming requests exhaust the CPU.
High outbound traffic	Incoming requests demand too much outgoing bandwidth, thus leading to buffer overflows and stalled server applications.
Large inbound packet backlog	Requests arrive faster than they can be handled, e.g., flood-type attacks.
Rate of timeouts for TCP connection requests	SYN-attack or network failure.

Table 7.1: Load indicators used by our prototype

dimension does not exist, it is created implicitly. Each newly-submitted filter is assumed to be more restrictive than the ones that were submitted before. Furthermore, a system call to "flush" the configuration of an FDim allows the administrator to start over with its configuration.

As soon as an FDim is created inside the kernel, user-space scripts may start reporting load for it. This, too, is done via a special system call. Less invasive configuration mechanisms like a configuration file, the `/proc` file system, or SNMP MIBs could have been used instead. However, this detail does not diminish the value of our prototype as a proof-of-concept. To avoid any conflicts between adaptive filters and firewalling rules that warrant network security, dynamic QGuard filters are always the last to be enforced. Thus, a QGuard-enabled system will never admit any packets that are to be rejected for security reasons. Our particular implementation achieves this goal by linking the QGuard firewalling chain as the last rule chain into `ipchain`'s input chain list.

The QGuard implementation relies on a distributed user-space monitor to obtain its load information. It consists of monitors that measures the server's health and a collector component, which resides on the AFP-enabled device. Whenever the load on the monitored servers changes, the collector receives load updates, which it feeds through to the QGuard kernel module. A configuration file specifies the association between monitoring variables and FDims. This flexible monitoring architecture allows for simple reconfiguration of the basic adaptation mechanism. Hence, we were able to explore a variety of different monitoring variables (listed in Table 7.1), their relationship to overload, and their usability in QGuard-based overload defense.



APF Deployment		Number of Rules					
		0	2	16	64	256	512
APF-enabled servers	% Drop	0%	6%	11%	11%	14%	27%
APF-enabled frontend	% Drop	0%	0%	0%	0%	0%	0%

Table 7.2: Basic overhead: aggregated performance drop vs. the number of rules per filter (FDim size 20 filters).

7.5 Evaluation

We studied the performance of APF using the QGuard prototype in two key deployment scenarios: APF-enabled servers and APF-enabled frontends. An Intel Pentium-based PC acts as a server hosting a synthetic load generator (which is based on the technology of Eve) with the following configurable load components:

CPU: The server executes a busy cycle of uniformly-distributed length (with a configurable mean).

File system: The server accesses random blocks in a file of more than two times RAM size to ensure that file accesses will not be redirected to the file cache. If file accesses were redirected to the server's file cache, the true bottleneck would be the server's memory subsystem, not its file I/O.

Network: The server returns response messages of configurable size.

Memory: The server allocates a configurable amount of memory for each received request and inverts one byte on every page in two passes over all pages.

The load generator waits to receive network requests either via TCP or UDP. The load generated in response to a received request can be configured on a per-port basis to include the CPU, file system, network, and memory components.

The client machines, two other Pentium-class machines, connect to the server through a Fast-Ethernet switch. They execute our client emulator generator, Eve, that simulates an arbitrary number of concurrent persistent clients to the load generating server. Unless otherwise stated, QGuard was configured with one FDim of 20 filters. This FDim's load variable was tied to the server's CPU load. Once the FDim was installed, it was not altered during the experiments, unless the benefits of the FM were explicitly analyzed. In this case, the FM adapts FDim's configuration online. A baseline comparing the QGuard-controlled server's performance against the server without QGuard

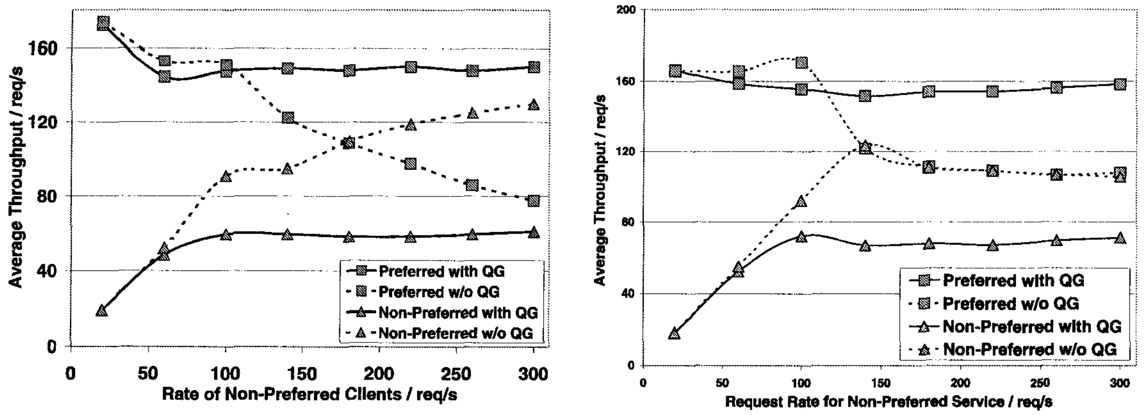


Figure 7.4: QoS differentiation: (left) by source IP, (right) by destination port.

was established in all our tests. Measurements were repeated until the estimated measurement error was under 5% with 95% confidence.

7.5.1 Basic Overhead

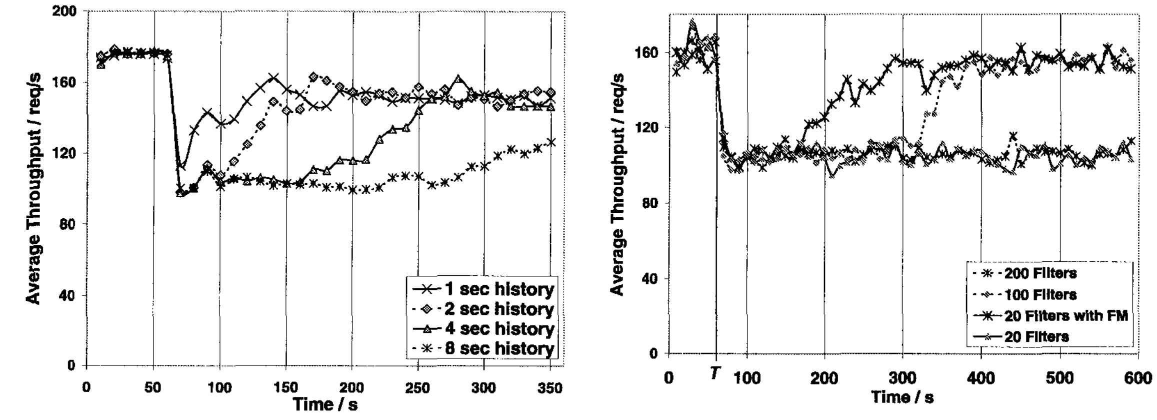
APF overhead depends on how often filters are switched and on the number of rules per filter. We determined that the delay between filter switches has only little impact on the aggregated throughput of a QGuard-protected server—it can be done in constant time. However, the number of rules per filter can affect performance, since the APF-enabled device must check each incoming packet against its rule-base. Table 7.2 shows that QGuard, when configured on the server machine, suffers a performance drop when the number of rules becomes large. However, this penalty disappears when QGuard is configured on a frontend device.

7.5.2 Overload Protection and QoS Differentiation

QGuard is designed to protect servers from overload and degrade QoS gracefully during overload. Overload protection was tested by subjecting our server to various overload conditions while monitoring the overall health of the server. In extreme cases, such as heavy memory swappping, the uprotected server crashed, whereas QGuard avoided the crash.

We validated QoS differentiation in two different scenarios: (left) the differential treatment of co-hosted services by performing destination port-based differentiation and (right) the differentiation based on clients' IP addresses. Based on our analysis in Chapter 6, we initially assumed homogeneous load. Later, we relax this assumption.

Each filter was configured to deliver up to three times more throughput to the preferred service



(left) History length (right) Filter grain

Figure 7.5: Rate of convergence

than to the non-preferred service. As Figure 7.4(left) shows, QGuard manages to provide QoS differentiation between the preferred and the non-preferred services. The throughput of the preferred service remains stable, regardless of the increase in the number of clients for the non-preferred service. This compares favorably with a 50% performance drop for clients of the preferred service without QGuard.

On first analysis of the ratios of preferred vs. non-preferred in Figure 7.4(left), the measured 2:1 ratio in throughput contradicts the configured 3:1 ratio. The reason for this apparent anomaly is that the preferred service sees all of its offered load being served. Hence, the remaining server capacity is allocated to the non-preferred service. Server capacity is not wasted.

QoS differentiation is enforced as well when traffic classes represent client groups. Figure 7.4(right) shows the results of our differentiation by client IP. In the absence of QGuard, Figures 7.4(left) and (right) differ slightly due to the server's request management and the CPU scheduling mechanism. When differentiation is done on the basis of the requests' IP source addresses, the requests access a shared listen queue for the service. Hence, if the non-preferred clients' IPs send requests at a higher rate, they will consume a larger percentage of the server's listen queue. This, in turn, implies that they can snatch all processing resources from preferred clients, unless QGuard throttles request rates. In contrast, when two different services are both flooded with client requests, the ratio of client requests does not matter under continuous overload. The UNIX scheduler will simply split the CPU fairly between both overloaded services, unless QGuard fends off the overload.

7.5.3 Responsiveness to Overload

Not only is it important to show that APF is capable of providing QoS differentiation, but one also wants to show how quickly it responds to overload. The following experiments subjected the QGuard-protected server to a sudden request surge and studied how quickly it restores preferred clients' throughput to 160 $reqs/s$. The experiments highlight the importance of the two configuration parameters: (1) the size of the averaging history window that is used to compute server load averages, and (2) the length of FDims.

Initially, only one client class, the preferred clients, request service from the server. Because the server can handle all requests easily, QGuard does not throttle incoming traffic at all until non-preferred clients cause a heavy load surge—more than twice the server's capacity—after time 60s. Figure 7.5(left) shows that QGuard's reaction to this overload depends on the length of the sliding history window over which the load index is computed—longer histories imply a larger switching delay. As expected, short histories produce fast response to overload. However, if histories are reduced to the duration of scheduling time-slices, load averages will become unstable and cause an unpredictable response (not shown in the graph).

Another experiment (Figure 7.5(right)) shows how the grain of installed filters impacts convergence to the best filter. In the underlying experiment we pretended not to know the server's request-handling capacity and generated filters covering evenly the range from 0 to 5000 SYN-packets per second. Figure 7.5 shows that neither very fine-grained (200 filters) nor too coarse-grained FDims (20 filters) warrant precise QoS control. An FDim of 200 filters converges so slowly that it cannot protect the server from overload. In contrast, the FDim consisting of 20 filters does not provide any filter that is close to the server's true capacity and therefore, oscillates between denying access to all incoming requests and accepting by far too many. These oscillations cause QGuard's overload protection mechanism to fail. For the workload of this experiment, 100 filters covering the range of 5000 SYN-packets per second achieved good overload protection and QoS differentiation. The FM, however, achieves faster response to overload and similar QoS control with an FDim of only 20 filters. This observation indicates the importance and effectiveness of our online FM when used to protect servers of unknown capacity.

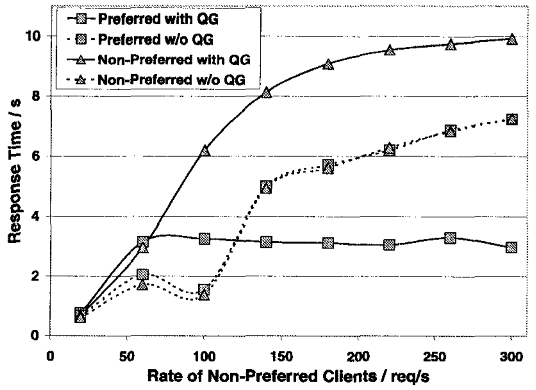


Figure 7.6: Preferred-client response time under increasing non-preferred client load

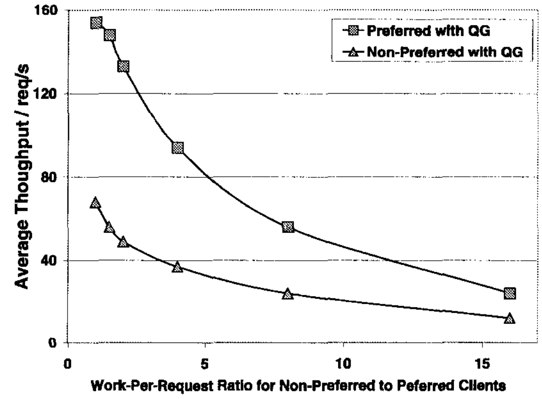


Figure 7.7: Achieved QoS differentiation under increasing per-request work

7.5.4 Limitations of Adaptive Packet Filters

Figure 7.6 shows response time differentiation between preferred and non-preferred clients. In fact, the preferred clients' response time is much shorter than that of the non-preferred ones. However, the response time for the preferred clients, who, by and large, comply with the rate limitation—except for some random spikes—is larger with competing requests than that without. This is due to the interaction that were studied in earlier chapters between rate controls and TCP, which, if enforced in the traffic shaper, cause some of the preferred clients' SYN packets to get lost. These lost SYN packets are retransmitted after a long timeout (3s in Linux), which, for some requests, increases average response time.

Another limitation of APFs relates to disparate per-request work requirements of the controlled traffic classes (Figure 7.7). Consistent with our analysis of workload sensitivity analysis in Chapter 6, we observed that QGuard's ability to distinguish the preferred service from the non-preferred service is degraded when the amount of work required to serve one request of the non-preferred service is orders of magnitude larger than that required to serve one request of the preferred service. Although QGuard does the best it can to provide QoS differentiation, controlling workload on a per-packet basis is too coarse-grained when each packet imposes large amounts of work. We conclude that APF-filtering solutions should not be used if few individual requests cause long-lasting server activity. In this context, fine-grained OS resource reservation mechanisms like Resource Containers [14], Scout [117], or Virtual Services are needed. Fortunately, most network-based services—FTP, HTTP, etc.—handle requests at high rates, thus permitting APF-based QoS management in most cases.

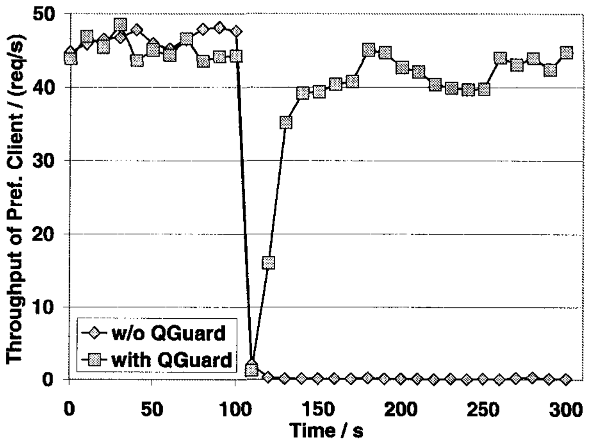


Figure 7.8: Ping-flood recovery with QGuard and the FM

In its current implementation, QGuard polices packets after they are copied from the NIC into kernel/router buffers. This raises concerns with respect to the so-called receive-livelock problem [91]. When packets arrive frequently enough to lock up the server/router CPU, QGuard will fail to provide overload protection. We do not believe, however, that this concern is justified when considering the implementation of APFs for routers, firewalls, and load-balancers because they are capable of processing all packets at wire-speed.

7.5.5 Further Benefits of the FDim Manager

As mentioned earlier, the FM is a useful addition to APFs if deployed in unknown server environments and whenever workload characteristics change dynamically. The FM aims to speed up convergence to the optimal shaping rate by keeping the number of filters small without sacrificing the ability to provide QoS differentiation (see Section 7.5.3).

In addition to the improvement in convergence behavior, the FM can respond to uncontrollable overload signals raised by the load-controller whenever the installed FDims fail to contain an overload. Figure 7.8 (at 100s) shows this mechanism in action as the server comes under a "ping-flood" attack. As Figure 7.8 shows, the FM determines incoming ICMP requests as the source of the overload and blocks this traffic class. The server continues to serve the preferred clients at almost maximal rate, requiring less than one minute from the onset of the ping-flood until recovery. The server recovered even though the ping-flood never stopped. To demonstrate QGuard's recovery from such lock-up caused by an ICMP flood, it was necessary to add a second 100Mbps Ethernet card to the server. With only one interface the server's CPU was powerful enough to process both incoming requests (at a slightly lower rate) and ECHO REQUEST packets.

It is important to note that APF can only partially mitigate the effects of ping flood on available bandwidth. Although incoming ECHO REQUESTs are filtered out, they may still saturate incoming link bandwidth. To protect the incoming link, APFs would have to be deployed not only on the Internet's edge, as proposed in this section, but also in its core. However, this raises further issues beyond the scope of this work.

7.6 Conclusions

The generic nature of the APF abstraction is the result of three major design choices. First, the dynamic adaptation mechanism allows APF deployment in clusters of unknown request processing capacity. Second, the configurable monitoring-feedback mechanism makes its use against a variety of adverse server conditions possible. Third, network-orientation make application and OS modifications unnecessary, thus facilitating APF use in arbitrary service environments. These benefits clearly justify further research on APFs and should encourage its use in OSs and network infrastructure devices.

Since most current OS kernels implement IP-based packet-filtering, APFs can be added to them easily. Furthermore, firewalling devices, edge-routers, and IP-layer switches, and anything else connecting servers to the Internet are primary candidates for APF integration. As we have shown in our experiments, an APF-enabled frontend does not affect server performance, yet it provides QoS differentiation. To become a key component of typical server cluster frontends [5, 36, 48, 58], the collector-to-monitor network interface needs standardization. Without standards, monitors cannot be reused for different APF-enabled frontends.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

This dissertation has focused on providing QoS support to Internet services using network-oriented mechanisms. In particular, we have examined several important issues in the design of effective traffic controllers. Our approach focused on the impact of traffic controllers on the underlying traffic and also on the controlled services. We used a combination of analytical modeling—to provide deeper understanding of the expected behavior of network traffic and services under control—and practical implementation—to test actual deployment issues in real systems. This dissertation makes several key contributions toward deploying network-oriented mechanisms to control Internet services. These contributions, particularly pertaining to the discovery and control of clients' persistence, will have potential for making a significant impact on the design of future network controllers.

We summarize the primary contributions made by this dissertation, which is followed by a discussion of future research directions.

8.1 Primary Contributions

This dissertation makes the following contributions toward advancing the state of the art in using NOSCs:

- A central issue in the design of any network control system is understanding the reaction of input traffic to a given control policy. This is crucial to the correct design and fine-tuning of NOSCs. We have introduced the *persistent client* model to capture the true reaction of client requests to the applied controls. This model is a departure from traditional approaches where client models are based on pre-collected traces that do not incorporate the effects of applied controls. We have used the persistent client model to carefully study the behavior of aggregate traffic in end-servers for different load scenarios. Based on our analysis, we have discovered the re-synchronization of dropped requests. This discovery shows that the burstiness of request arrivals can be exacerbated by traditional traffic policing mechanisms like RED and Token Buckets. Furthermore, such mechanisms will unnecessarily increase the client-perceived latency.

To improve the accuracy of our evaluation, we have integrated the proposed persistent client model into our measurement infrastructure. We have built Eve, a scalable, extensible, and programmer-friendly tool that allows fast development of clients to test high capacity Internet servers. Eve was used to simulate thousands of persistent clients with very high accuracy.

- We have used our model of persistent clients to develop an analytical model relating the drop probability and the connection-establishment delay—one of the primary components of the client-perceived latency when accessing an Internet service. Our analytical construction was used as a basis to create *persistent dropping*, a simple and novel mechanism that minimizes the client-perceived delay and the amount of retransmitted requests while achieving the same control targets as traditional control mechanisms. We have shown that persistent dropping is important in providing effective NOSCs for three reasons: (1) it improves the rate of convergence of incoming traffic to the applied control without sacrificing fairness, (2) it eliminates the delay component that is attributed to the applied controls, and (3) it is both easily implementable and computationally tractable.

- We have proposed the use of explicit reject message when the server wants to immediately inform the client of his rejected request. This will effectively (1) reduce the number of client transmissions, (2) reduce the wait time for the client to realize the server's decision to deny him access, and (3) possibly break the re-synchronization behavior of re-transmitted attempts by the client. Because there is no universal mechanism for providing admission control at the IP level, we proposed a new IETF Internet standard for extending the Internet Control Message Protocol (ICMP). This new message can be incrementally deployed and is backward compatible with today's network stack implementations. We have shown that this new capability is an effective way for servers to deal with clients' persistence.

- To handle the burstiness of request arrivals, we have introduced the *Abacus Filter*, a new multi-stage filter that can perform accurate traffic prediction of future arrivals based on the applied control policy. Abacus filters complemented our persistent dropping technique in two ways: (1) it better adapts to changing arrival patterns, (2) it balances between maximizing the number of admitted requests and minimizing the resulting client-perceived latency. The Abucus Filter models the time of the next TCP retransmissions to estimate the amount of available future capacity and only admits the portion of a burst that, with high probability, will eventually be admitted. We have shown that with Abacus Filters, bursty traffic or a sudden jump in network load does not affect the overall delay of successful connections.

- We have formally analyzed multi-threaded Internet services to study the effects of the applied NOSCs on the running services. We have introduced *multi-threaded round robin* (MTRR) server, a new model that incorporates the effects of increased concurrency on the performance of the underlying services. Our MTRR server model is a generalization of existing time-sharing process models and is more representative of current service implementations. In particular, our model captures three important elements: (1) the effects of concurrency on the performance of a service, (2) the interactions among threads/processes of different services, and (3) the relationship between request arrival and service performance. We have used our analytical study of MTRR server model to answer two important issues in the design of effective QoS mechanisms: (1) predict the true impact of the applied rate-based control on client-perceived latency, (2) find the maximum arrival rate that guarantees certain response times to different client groups.

- We have proposed *Adaptive Packet Filters*, a new adaptation mechanism that integrates signaling information with dynamic control. Adaptive Packet Filters require minimal OS modifications to provide easy integration of our new control filters (i.e., persistent dropping and abacus filters) into exiting OS architectures. We have shown that the current implementation provides full overload protection as well as service differentiation.

8.2 Future Directions

The work in this dissertation can be extended in several directions. These are briefly described as follows:

- **Persistence model of real users:** In this dissertation, we have only modeled the persistence that is caused by TCP congestion control and browser parallelism. It is not uncommon for a typical user to press the reload button several times or open new browser windows when the server is not responding. Therefore, the actual users' behavior is also an important factor that should be studied. Performing such study is, however, not easy for two reasons. First, it is practically impossible to extract the user's expected reaction to different controls from pre-collected traces. Such traces normally do not include enough information and only reflect the behavior of users under specific (unknown) network and server load conditions. Second, while using real users in a controlled laboratory setup is probably the most appropriate method of performing the study, much care has to be taken in choosing the demography of real users. This is further complicated by the fact that users often react differently depending on the relative importance of the information that they are trying to access.

- **Identifying interactions between different controls:** The persistent client model shows a very important result of protocol design: different control mechanisms, under certain conditions, can interact with each other in a very unpredictable manner. We have focused in this dissertation on one particular instance, namely the interaction between active queue management controls and TCP congestion control. However, other and more general interactions are just as (if not, even more) important. Identifying such interactions is not a straightforward task as different controls are being applied at various stages of the communication hierarchy.

- **Building non-intrusive controls:** Persistent dropping is an example of a control technique that does not interact with other control mechanisms. That is, those requests that manage to pass through will not incur any delay. Thus, once they are accepted by the corresponding service, they will seem as if they were not subjected to any control mechanism prior to their acceptance. It would be interesting to extend the same design philosophy to other control mechanisms. Unfortunately, it is still not clear whether other network and application control mechanisms can be modified to be non-intrusive.

- **Network-device integration:** In this dissertation, we have primarily focused on applying NOSCs in end-servers. They are intended as the first line of defense against server overload. However, NOSCs should not be limited to only end-servers. They can be integrated into various network devices such as edge routers and load balancers. This ability would make them most suited for controlling server-farms since they can be placed in front-end devices. Network devices, in general, have either a light-weight design (e.g., switches) or are performance-sensitive (e.g., routers and load balancers). Therefore, much care has to be taken into identifying what can feasibly be placed in such devices and integrating various signaling information with these controls. It would be interesting to investigate the feasibility of placing NOSCs on network devices in terms of resource requirements as well as efficiency. It is also interesting to investigate various techniques to incorporate signaling information.

APPENDIX

APPENDIX A

Reject Message Extension for ICMP

Internet Draft	H. Jamjoom K. G. Shin
Document: draft-jamjoom-icmpreject-00.txt	University of Michigan
Expires: October 2002	August 2002

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026 [1]. Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts. Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet- Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at:

http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at:

http://www.ietf.org/shadow.html

Abstract

This document specifies the incorporation of a Reject message to ICMP to improve a servers ability for controlling TCP connection requests during periods of overload beyond dropping them. The Reject message serves two purposes: (1) it can inform a connecting host (the client) to completely abort the connection attempt as if a connection timeout has expired, and (2) modify the retransmission timeout interval to reduce the clients wait times and also to break the re-synchronization

of connection requests when multiple SYN packets are dropped. Introducing an additional ICMP message, as opposed to modifying the behavior of TCP, was motivated by the need for backward compatibility (i.e., a host can ignore the ICMP Reject message without affecting the behavior of TCP) and incremental deployment.

Conventions used in this document

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119 [2].

This document is mainly concerned with two types of host machines: "SERVERS" and "CLIENTS". A "CLIENT" is a host machines that send a request via TCP (only) to a "SERVER" machine, which, in turn, process the request and sends a response back. A "GATEWAY" is interpreted as any packet-forwarding device between the client and the server. Each "REQUEST" is assumed in this document to require a separate connection, which means that it will involve a three-way connection establishment before any data is sent between the server and client.

Table of Contents

1. Introduction

Short-lived Transport Control Protocol (TCP) connections are typical of interactive Web sessions. They are delay-sensitive and have transfer times dominated by TCP back-offs, if any, during connection establishment [3]. Unfortunately, arrivals of such connections (i.e., SYN packets) at a server tend to be bursty [4], and when the server is busy, they may get dropped, triggering retransmission timeouts. This results in long average client-perceived delays. To date, TCP does not provide any explicit mechanism during the connection establishment phase that improves a servers controls beyond dropping connection requests. In fact, TCP relies solely on dropping packets to trigger clients to back off (exponentially) and retransmit their request at a later time. This process is repeated until either the connection is established or a connection timeout expires and the connection attempt is aborted [5].

Three particular issues arise from TCPs limitation. First, we observed that when a large number of packets are dropped due to a burst, these will re-synchronize themselves and will be retransmitted by the clients and arrive at the server at roughly the same time resulting in an additional burst at the server. Second, because TCP exponentially backs off, the client experiences excessively long delays even when the server is temporarily busy and drops the clients connection requests. We assume here that the servers listen buffer is full. The server, in this case, has no control over how long the client should wait before trying to connect again. Third, in some cases the server may have an advantage for rejecting a clients connection request to (a) immediately inform the client that it will not be able to process its request and, thus, minimize the clients wait time (traditionally, the client must wait for a long connection timeout period), and (b) minimize the number of retransmissions by the client since a single Reject message is returned to the client.

Existing TCP control mechanisms, which include Explicit Congestion Notification [6] and ICMP Source Quench Message [7], only take effect after the connection is established. However, providing better controls for the connection establishment phase is of importance, especially for short-lived TCP connections. Two basic functionalities should be provided. First, a server should be able to inform the client of when to resubmit the connection request. The client, in return, should reduce its retransmission timeout to that value. Second, the server should be able to reject the connection request altogether. The client would then abort the connection as if the connection timeout has been reached.

The addition of one ICMP message type will achieve both goals. The use of an ICMP message, as opposed to modifying the actual transport protocol, is motivated by the need for incremental deployment and backward compatibility. More specifically, if the ICMP message is dropped by intermediate gateways or if the client does not honor such message, then the client will perceive a typical TCP behavior, namely an unacknowledged SYN packet. This will, in turn, trigger a retransmission timeout as defined in [5]. On the other hand, if the client does honor such message, then only TCP timeout values are transparently adjusted, and thus, higher-level applications should continue to work properly.

There is one issue of importance, namely, the security effects of modifying TCPs timeout values. This arises because the clients retransmission timeout value is set under the discretion of the server. A minimum timeout value must then be enforced to avoid the situations where the server drops SYN packets and informs the client to retransmit immediately, which would create another form of Denial of Service attack. We recommend that this value set to the initial RTO value, which is typically 3 sec. In that respect, the server is only able to control the exponential backoff behavior of clients in a highly secure fashion.

Source Reject Message

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|     Type      |     Code      |          Checksum             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                Minimum Retransmission Time                    |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      Internet Header + 64 bits of Original Data Datagram      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

IP Fields:

Destination Address

The source network and address of the original datagram's data.

ICMP Fields:

Type

19

Code

0 = abort connection;

1 = modify retransmission timeout.

Checksum

The computation of the checksum follows the same requirements in other ICMP messages as described in [7]: "The checksum is the 16-bit ones's complement of the one's complement sum of the ICMP message starting with the ICMP Type. For computing the checksum, the checksum field should be zero. This checksum may be replaced in the future."

Minimum Retransmission Time:

This field contains the number of milliseconds that the client must wait before retransmitting the SYN packet. It must contain a minimum value of 3 seconds.

Internet Header + 64 bits of Data Datagram

Similarly, the same specification as [7] for the Internet header plus the first 64 bits of the original datagram's data is used. The host uses this data to match the message to the appropriate process. Of course since this is the message in response to a TCP SYN packet, the port numbers are assumed to be in the first 64 data bits of the original datagram's data.

Description

A host receiving a Reject message will either abort the connection as if a TCP connection timeout has expired (Code 0) or change the retransmission timeout of the corresponding TCP connection, which is determined from the Internet Header and the 64 bits of Data, to the value specified in the Minimum Retransmission Timeout field (Code 1). If the Minimum Retransmission Time is less than the recommended 3 seconds, the host should discard the message and wait for the regular TCP timeout to expire.

Security Considerations

The use of ICMP Reject does not increase a clients or a servers vulnerability beyond existing TCP implementations or other ICMP messages. The inclusion of the original SYN packet header in the Reject message allows the receiving host to validate the authenticity of the message.

In the case of a Denial of Service (DoS) attack at the server, the server must implement similar defense techniques for minimizing the number of Reject messages sent to its clients. This is in par with traditional techniques used for reducing the number of SYN ACK replies.

References

1. Bradner, S., "The Internet Standards Process – Revision 3", BCP 9, RFC 2026, October 1996.

2. Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997

3. Olshefski, D., J. Nieh, and D. Agrawal, Inferring Client Response Time at the Web Server, In Proceedings of ACM SIGMETRICS 2003, June 2003

4. Feldmann, A., Characteristics of TCP Connection Arrivals. Self-Similar Network Traffic and Performance Evaluation. John Wiley and Sons, Inc., 2000, ch. 15, pp. 367-399

5. Postel, J. (ed.), "Transmission Control Protocol DARPA Internet Program Protocol Specification," RFC 793, USC/Information Sciences Institute, September 1981

6. Ramakrishnan, K., S. Floyd, and D. Black, The Addition of Explicit Congestion Notification (ECN) to IP, RFC 3168, TeraOptic Networks, September 2001

7. Postel, J. (ed.), "Internet Control Message Protocol DARPA Internet Program Protocol Specification," RFC 792, USC/Information Sciences Institute, September 1981

Author's Addresses

Hani Jamjoom
University of Michigan
1301 Beal Ave.
Ann Arbor, MI 48109-2122
Email: jamjoom@eecs.umich.edu

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] ABDELZAHER, T., AND BHATTI, N. Web Content Adaptation to Improve Server Overload Behavior. In *International World Wide Web Conference* (May 1999).

[2] ABRY, P., BARANIUK, R., FLANDRIN, P., RIEDI, R., AND VEITCH, D. The Multiscale Nature of Network Traffic: Discovery, Analysis, and Modelling. *IEEE Signal Processing Magazine 19*, 3 (May 2002), 28–46.

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

[4] ALMEIDA, J., DABU, M., MANIKUTTY, A., AND CAO, P. Providing Differentiated Levels of Service in Web Content Hosting. In *ACM SIGMETRICS Workshop on Internet Server Performance* (June 1999).

[5] ALTEON WEBSYSTEMS. Web OS Traffic Control Software. http://www.alteonwebsystems.com/products/webos/, 2000.

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

[7] ARLITT, M. F., AND WILLIAMSON, C. L. Web Server Workload Characterization: The Search for Invariants. In *Proceedings of the ACM SIGMETRICS '96 Conference* (Philadelphia, PA, April 1996).

[8] ARON, M., DRUSCHEL, P., AND ZWAENEPOEL, W. Cluster Reserves: A Mechanism for Resource Management in Cluster-Based Network Servers. In *Proceedings of ACM SIGMETRICS* (June 2000).

[9] AURRECOECHEA, C., CAMPBELL, A., AND HAUW, L. A Survey of QoS Architectures. *Multimedia Systems 6*, 3 (1998), 138–151.

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

[11] BALAKRISHNAN, H., PADMANABHAN, V. N., SESHAN, S., STEMM, M., AND KATZ, R. TCP Behavior of a Busy Internet Server: Analysis and Improvements. pp. 252–262.

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

[13] BANGA, G., AND DRUSCHEL, P. Measuring the capacity of a Web server. In *Proceedings of The USENIX Symposium on Internet Technologies and Systems* (December 1997).

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

[15] BARFORD, P., BESTAVROS, A., BRADLEY, A., AND CROVELLA, M. E. Changes in Web Client Access Patters: Characteristics and Caching Implications. In *World Wide Web, Special Issue on Characterization and Performance Evaluation* (1999), pp. 15–28.

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

[17] BENNETT, J. K., CARTER, J. K., AND ZWAENEPOEL, W. Munin: Distributed Shared memory Based on Type-Specific memory Coherence. In *Proceedings of the Second ACM Symposium on Principles and Practice of Parallel Programming* (1990), pp. 168–176.

[18] BERAN, J. Statistical Methods for Data with Long-Range Dependence. *Statistical Science 7*, 4 (1992), 404–427.

[19] BHATTI, N., AND FRIEDRICH, R. Web Server Support for Tiered Services. *IEEE Network 13*, 5 (Sep.–Oct. 1999), 6764–71.

[20] BHOJ, P., RAMANATHAN, S., AND SINGHAL, S. Web2K: Bringing QoS to Web Servers. Tech. Rep. HPL-2000-61, HP Labs, 1999.

[21] BIGGS, N. L. *Discrete Mathematics.* Oxford University Press, New York, 1989.

[22] BLAKE, S., BLACK, D., CARLSON, M., DAVIES, E., WANG, Z., AND WEISS, W. An Architecture for Differentiated Services. *RFC 2475* (1998).

[23] BORLAND, J. Net Video Not Yet Ready for Prime Time. Tech. rep., CNET news, February 1999.

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

[25] BRADEN, R., CLARK, D., AND SHENKER, S. Integrated Services in the Internet Architecture: an Overview. *RFC 1633* (1994).

[26] BRUNO, J., BRUSTOLONI, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. Retrofitting Quality of Service into a Time-Sharing Operating System. In *USENIX Annual Technical Conference* (June 1999), pp. 15–26.

[27] BRUSTOLONI, J., GABBER, E., SILBERSCHATZ, A., AND SINGH, A. Signaled Receiver Processing. In *Procedings of the 2000 USENIX Annual Technical Conference* (San Diego, CA, June 2000), USENIX.

[28] CAO, J., CLEVELAND, W. S., LIN, D., AND SUN, D. X. Internet Traffic Tends to Poisson and Independence as the Load Increases. Tech. rep., Bell Labs, 2001. URL: http://cm.bell-labs.com/cm/ms/departments/sia/wsc/webpapers.html.

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

[30] CARRIERO, N., GELERNTER, D., AND LEICHTER, J. Distributed Data Structures in Linda. In *Proc. ACM Symposium on Principles of Programming languages* (1986), pp. 236–242.

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

[32] CHANG FENG, W., KANDLUR, D., SAHA, D., AND SHIN, K. G. The BLUE Active Queue Management Algorithms. *IEEE/ACM Trans. on Networking 10*, 4 (September 2002), 67–85.

[33] CHEN, X., AND HEIDEMANN, J. Flash Crowd Mitigation via Adaptive Admission Control based on Application-level Observations. Tech. Rep. ISI-TR-557, USC/Information Science Institute, May 2002.

[34] CHEN, X., MOHAPATRA, P., AND CHEN, H. An Admission Control Scheme for Predictable Server Response Time for Web Accesses. In *Proceedings of the 10th International World Wide Web Conference* (2001), pp. 545–554.

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

[36] CISCO INC. Local Director (White Paper). http://www.cisco.com/, 2000.

[37] COMMITTEE, S. D. SPECweb. Tech. rep., April 1999. http://www.specbench.org/osg/web/.

[38] CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. *Introduction to Algorithms*. The MIT Press, 1990.

[39] CROVELLA, M., AND LIPSKY, L. Long-Lasting Transient Conditions in Simulations with Heavy-Tailed Workloads. In *Winter Simulation Conference* (1997), pp. 1005–1012.

[40] CROVELLA, M. E., AND AN MOR HARCHOL-BALTER, R. F. Connection Scheduling in Web Servers.

[41] CROVELLA, M. E., AND BESTRAVROS, A. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. *ACM/IEEE Transactions on Networking 5*, 6 (Dec 1997), 835–846.

[42] DEMERS, A., KESHAV, S., AND SHENKER, S. Analysis and Simulation of a Fair Queueing Algorithm. In *Proceedings of ACM SIGCOMM '89* (August 1989).

[43] DUFFIELD, N. G., GOYAL, P., GREENBERG, A., MISHRA, P., RAMAKRISHNAN, K. K., AND VAN DER MERIVE, J. E. A Flexible Model for Resource Management in Virtual Private Networks. In *Proceedings of ACM SIGCOMM '99* (1999).

[44] EGGERT, L., AND HEIDEMANN, J. S. Application-Level Differentiated Services for Web Servers. *World Wide Web 2*, 3 (1999), 133–142.

[45] ELLIOTT, J. Distributed Denial of Service Attacks and the Zombie Ant Effect. *IT Pro* (March 2000).

[46] EMBRESHTS, P., AND MAEJIMA, M. An Introduction to the Theory of Self-Similar Stochastic Processes.

[47] ENGELSCHALL, R. S. Portable Multithreading:The Signal Stack Trick For User Space Thread Creation. In *Procedings of the 2000 USENIX Annual Technical Conference* (San Diego, CA, June 2000), USENIX.

[48] EXTREME NETWORKS, INC. Policy-Based QoS. http://www.extremenetworks.com/products, 2000.

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

[50] FELDMANN, A., GILBERT, A., WILLINGER, W., AND KURTZ, T. The Changing Nature of Network Traffic: Scaling Phenomena. *Computer Communication Review 28*, 2 (April 1998).

[51] FELDMANN, A., GILBERT, A. C., HAUNG, P., AND WILLINGER, W. Dynamics of IP Traffic: A Study of the Role of Variability and Impact of Control. In *Proceedings of the ACM SIGCOMM '99* (1999), pp. 301–313.

[52] FELDMANN, A., GILBERT, A. C., AND WILLINGER, W. Data Networks as Cascades: Investigating the Multifractal nature of Internet WAN traffic. *Computer Communication Review 28*, 4 (1998), 42–55.

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

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

[55] FLOYD, S., AND JACOBSEN, V. Link-Sharing and Resource Management Models for Packet Networks. *Transactions on Networking 3*, 4 (1995), 365–386.

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

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

[58] FOUNDRY NETWORKS. Server Load Balancing. http://www.foundrynetworks.com/, 2000.

[59] GARBER, L. Denial-of-Service Attacks Rip the Internet. *Computer* (2000).

[60] GARRETT, M., AND WILLINGER, W. Analysis, Modeling, and Generation of Self-Similar VBR Video Traffic. In *Proceedings of ACM SIGCOMM '94* (September 1994).

[61] GROBB, D., AND HARRIS, C. M. *Fundamentals of Queueing Theory.* Wiley Interscience, 1985.

[62] GUO, L., CROVELLA, M., AND MATTA, I. How Does TCP Generate Pseudo-self-similarity? In *Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems* (Cincinnati, OH, August 2001).

[63] HAHN, G. J., AND SHAPIRO, S. S. *Statistical Models in Engineering.* John Wiley and Sons, Inc., 1976.

[64] HAND, S. M. Self-Paging in the Nemesis Operating System. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation* (New Oreleans, Lousiana, February 1999), USENIX, pp. 73–86.

[65] HEIDEMANN, J., OBRACZKA, K., AND TOUCH, J. Modeling the Performance of HTTP Over Several Transport Protocols. *IEEE/ACM Transactions on Networking 5*, 5 (November 1996), 616–630.

[66] HEWLETT PACKARD CORP. WebQoS Technical White Paper. http://www.internetsolutions.enterprise.hp.com/ webqos/products/overview/wp.html, 2000.

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

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

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

[70] JUNG, J., KRISHNAMURTHY, B., AND RABINOVICH, M. Flash Crowds and Denial of Service Attacks: Characterization and Implications for CDNs and Web Sites. In *Proceedings of the 11th International World Wide Web Conference* (Honolulu, Hawaii, May 2002).

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

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

[73] KLEINROCK, L. Time-Shared Systems: A Theoretical Treatment. *Journal of the ACM 14* (April 1967).

[74] KLIENROCK, L. *Queueing Systems, Volume I: Theory*. Wiley Interscience, 1975.

[75] KLIENROCK, L. *Queueing Systems, Volume II: Computer Applications*. Wiley Interscience, 1976.

[76] KRISHNA, C. M., AND SHIN, K. G. *Real-Time Systems*. McGraw-Hill, 1997.

[77] LAKSHMAN, K., YAVATKAR, R., AND FINKEL, R. Integrated CPU and Network I/O QoS Management in an Endsystem. In *Proc. 5th International Workshop on Quality of Service (IWQOS'97)* (1997), pp. 167–178.

[78] LAKSHMAN, T. V., AND MADHOW, U. The Performance of TCP/IO for Networks with High Bandwidth-delay products and random loss. *IEEE/ACM Transactions on Networking 5*, 3 (July 1997), 336–350.

[79] LAZOWSKA, E. D., ZAHORJAN, J., GRAHAM, G. S., AND SEVCIK, K. C. *Quantitative System Perforamnce: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., 1984.

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

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

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

[83] LIU, X., SHA, L., DIAO, Y., AND FROEHLICH, S. Online Response Time Optimization of Apache Web Server. In *Proc. 11th International Workshop on Quality of Service (IWQOS'2003)* (Monterey, CA, June 2003).

[84] LU, C., ABDELZAHER, T. F., STANKOVIC, J. A., AND SON, S. H. A Feedback Control Approach for Guaranteeing Relative Delays in Web Servers. In *IEEE Real-Time Systems Symposium* (Taipei, Taiwan, December 2001).

[85] MANAJAN, R., BELLOVIN, S. M., FLOYD, S., IOANNIDIS, J., PAXSON, V., AND SHENKER, S. Controlling High Bandwidth Aggregates in the Network. *SIGCOMM Computer Comm. Review 32*, 3 (July 2002).

[86] MARIE, F. Netfilter Extensions HOWTO. http://www.netfilter.org.

[87] MENASC, D., ALMEIDA, V., RIEDI, R., RIBEIRO, F., FONSECA, R., AND JR., W. M. Characterizing and Modeling Robot Workload on E-Business Sites. In *Proceedings of ACM SIGMETRICS'01* (June 2001).

[88] MISRA, V., GONG, W., AND TOWSLEY, D. Stochastic Differential Equation Modeling and Analysis of TCP Windowsize Behavior. Tech. Rep. ECE-TR-CCS-99-10-01, Department of Electrical and Computer Engineering, University of Massachusetts, 1999. Presented at Performance'99, October Istanbul 1999.

[89] MOGUL, J. Observing TCP Dynamics in Real Networks. In *Proceedings of the ACM SIGCOMM '92* (1992), pp. 305–317.

[90] MOGUL, J. C. The Case for Persistent-Connection HTTP. In *Proceedings of the ACM SIGCOMM '95* (1995), pp. 299–313.

[91] MOGUL, J. C., AND RAMAKRISHNAN, K. K. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *Transactions on Computer Systems 15*, 3 (August 1997), 217–252.

[92] MOGUL, J. C., RASHID, R. F., AND J. ACCETTA, M. The Packet Filter: An Efficient Mechanism for User-Level Network Code. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles and Design* (November 1987), ACM.

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

[94] MOSBERGER, D., AND JIN, T. httperf — A Tool for Measuring Web Server Performance. Tech. rep., HP Research Labs. http://www.hpl.hp.com/personal/David_Mosberger/httperf.html.

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

[96] NORROS, I., MANNERSALO, P., AND WANG, J. Simulation of Fractional Brownian Motion with Conditionalized Random midpoint displacement. *Advances in Performance Analysis 2*, 1 (1998), 77–101.

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

[98] PANDEY, R., BARNES, J. F., AND OLLSSON, R. Supporting Quality of Service in HTTP Servers. In *Symposium on Principles of Distributed Computing* (1998), pp. 247–256.

[99] PARK, K., AND WILLINGER, W., Eds. *Self-Similar Network Traffic and Performance Evaluation.* John Wiley & Sons, Inc., 2000.

[100] PAXON, V. End-to-end Internet Packet Dynamics. In *Proceedings of the ACM SIGCOMM '97* (1997), pp. 139–152.

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

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

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

[104] PRUTHI, P., AND ERRAMILLI, A. Heavy-Tailed ON/OFF Source Behavior and Self-Similar Traffic. In *Proceeding of ICC'95* (June 1995).

[105] REUMANN, J., MEHRA, A., SHIN, K., AND KANDLUR, D. Virtual Services: A New Abstraction for Server Consolidation. In *Proceedings of the 2000 USENIX Annual Technical Conference* (San Diego, CA, June 2000), USENIX.

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

[107] ROSS, S. M. *Applied Probability Models with Optimization Applications.* Dover Publications, Inc., 1992.

[108] RUSSELL, P. IPCHAINS-HOWTO. http://www.rustcorp.com/linux/ipchains /HOWTO.html.

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

[110] SAHU, S., NAIN, P., TOWSLEY, D., DIOT, C., AND FIROIU, V. On Achieveable Service Differentiation with Token Bucket Marking for TCP. In *Proceedings of ACM SIGMETRICS* (June 2000).

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

[112] SAYOOD, K. *Introduction to Data Compression.* Morgan Kaufmann Publishers, Inc., 1996.

[113] SHA, L., LIU, X., LU, Y., AND ABDELZAHER, T. Queueing Model Based Network Server Performance Control. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium* (Austin, TX, December 2002).

[114] SHENKER, S. A Theoretical Analysis of Feedback Flow Control. In *Proceedings of the ACM SIGCOMM '90* (1990), pp. 156–165.

[115] SILBERSCHATZ, A., GALVIN, P., AND GAGNE, G. *Applied Operating System Concepts.* John Wiley and Sons, Inc., 2000.

[116] SOVOIA, A. The Science of Web Site Load Testing. Tech. rep., Keynote System, Inc., 2000. http://www.keynote.com.

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

[118] TAKAGI, H. Queueing Analysis of Polling Models. *ACM Computing Surverys 20,* 1 (1988).

[119] VASILIOU, N., AND HANAN. Providing a Differentiated Quality of Service in a World Wide Web Server.

[120] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well-Conditioned Scalable Internet Service. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles* (October 2001).

[121] WOLFF, R. W. *Stochastic Modeling and the Theory of Queues.* Prentice-Hall, Inc., 1989.

[122] ZHANG, H., AND FERRARI, D. Rate-Controlled Static Priority Queueing. In *Proc. of the IEEE INFOCOM 1993* (San Francisco, 1993), pp. 227–236.

[123] ZHANG, H., AND FERRARI, D. Rate-Controlled Service Disciplines. *Journal of High-Speed Networks 3,* 4 (1994).

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

[125] ZHANG, L., SHENKER, S., AND CLARK, D. Observations on the Dynamics of a Congestion Control Algorithm: The Effects of Two-Way Traffic. In *Proceedings of the ACM SIGCOMM '91* (1991), pp. 133–147.

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

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