Addisu Tadesse Eshete

Stateless and Statelet Flow Protection for the Internet

Thesis for the degree of Philosophiae Doctor

Trondheim, November 2012

Norwegian University of Science and Technology
Faculty of Information Technology,
Mathematics and Electrical Engineering
Department of Telematics

NTNU – Trondheim
Norwegian University of
Science and Technology

NTNU
Norwegian University of Science and Technology

Thesis for the degree of Philosophiae Doctor

Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Telematics

ISBN 978-82-471-3953-0 (printed ver.)
ISBN 978-82-471-3954-7 (electronic ver.)
ISSN 1503-8181

Doctoral theses at NTNU, 2012:317

Printed by NTNU-trykk

Preface

This thesis is submitted in partial fulfillment of the requirements for the degree of philosophiae doctor (PhD) at the Department of Telematics, Norwegian University of Science and Technology (NTNU). The research was undertaken during the period from January 2009 to August 2012, and I was hosted and funded by the Centre for Quantifiable Quality of Service in Communication Systems (Q2S) which is a Norwegian Center of Excellence at NTNU. The advisor of the thesis has been Professor Yuming Jiang.

Apart from the scientific research, the doctoral education consists of mandatory courses totalling 30 ECTS credits.

Acknowledgements

First and foremost, I would like to express my gratitude to my thesis supervisor Prof. Yuming Jiang who first invited me for a research visit in September 2008. Throughout the PhD study, he continuously challenged me to aim high and helped me to maintain my research focus. Despite his hectic schedule, he was always available for fruitful discussions and supervision on short notices. I am thankful for all his efforts.

My sincere gratitude also goes to my thesis adjudication committee: Dr. James Roberts from INRIA, Prof. Michael Welzl from University of Oslo and Prof. Peder J. Emstad from NTNU.

I am fortunate to have been at Q2S where the work environment has been imbued with strong collegiate friendship. I sincerely thank all of my colleagues at Q2S for that. Special thanks go to Anniken Skotvoll and Mette Veronica Olsen for organizing many social outings during my stay and saving me from the administrative troubles, and Hans Almåsbakk for maintaining a very good computer facility at Q2S. I would also like to record my gratitude to my officemates Andrés Gonzalez for great friendship inside and outside workplace, and Anne Nevin who has always been lending a supportive ear at all times. I also thank Kashif Mahmood and Lars Landmark for co-authoring, and Laurent Paquereau for initial discussions on `ns-2`.

I am deeply indebted to Nuria Tavera for her love, encouragement and understanding during my long working hours. Many thanks to Andrés, Pern, Achenef, Mark, Laurent, Dirk, Atef and Yanling for our friendship, long chats during coffee breaks and off-work hours.

Finally, but most importantly, my deepest gratitude goes to my parents and siblings who, despite the distance, comfort me with their unconditional love and care throughout my life. I gladly dedicate this thesis to them.

Abstract

Despite its rapid and tremendous growth, the basic packet delivery service in the Internet has largely remained *best-effort and egalitarian*. Consequently, the Internet from the outset lacks powerful mechanisms for fair arbitration of its own shared resources among the users it serves. For example, user flows injecting more bytes per time unit can proportionally receive more service, at the expense of others with lower rates. In the extreme case, certain flows or users can completely be denied of service (i.e., access to link and buffer resources) in the network.

A lot of research has been conducted to correct the fairness limitation and two general, yet complementary, approaches have been followed: end-to-end based (e.g., TCP) and network-based. The first solutions are congestion control algorithms doubling as fairness control algorithms as their secondary goal. However, they require on the part of end hosts universal adoption which limits the scope of their applicability. The latter approaches are to be enabled at the routers and can in turn be classified into two categories: perflow fair queueing (or scheduling) schemes and queue management mechanisms. While the former are powerful in providing high-quality fairness among flows, they are plagued by lack of scalability features as they require complex per packet operations and maintenance of perflow states. The queue management schemes, to the contrary, are generally simpler and hence more scalable, but lack generality and quality flow protection and fairness.

In this thesis, we propose a host of *simple, stateless or statelet network-based single aggregate queue mechanisms for enforcing flow fairness, and/or flow protection* in the Internet.

For flow fairness, which is the stronger requirement, we approximate the max-min fairness of perflow fair queueing using a single queue serving all traversing flows. By assigning sorting tags to arriving packets, we can not only differentiate service priorities of flows but also eliminate the complex buffer partitioning normally required in perflow queueing schemes. Our proposed scheme has several desirable features. It is statelet as the flow state requirement is limited mainly to those flows taking relatively more bandwidth than the fair share. We leverage the limited flow state to develop a packet drop policy which helps avoid unwanted lockout (unfairness) behavior. We demonstrate by extensive simulations that the scheme is highly fair, efficient in resource utilization and suitable for a wide range of traffic adopting heterogenous TCP congestion control algorithms.

For flow protection, we require that the resource share of the high bandwidth flows should be bounded. To that end, we propose, model and analyze a suite of active queue management schemes, which generalize the well-known CHOKe scheme. Network operators can control the desired flow protection level by tuning a configurable parameter which also indexes the specific scheme chosen. Our flow

protection framework not only remains simple and completely stateless, but is also highly effective in controlling the resource share of aggressive flows, as proved and verified by the tight bounds on flow buffer space shares and link utilizations. This allows well-behaved sources to obtain better service, i.e., low queueing delays and higher throughput, in the network.

Another key contribution of the thesis is the first analysis on the transient behavior of CHOKe queue following a change in the source rate of the unresponsive flow. The observed dynamic behaviors seem counter-intuitive as the flow's throughput is moving in a direction opposite to the changes in the flow's source rate. We present two models that characterize and explain these "perplexing" transient behaviors.

Contents

1. Introduction

Fairness among best-effort connections is not just an intuitively desirable property of queueing systems [57], but also one with immense practical benefits. First, it ensures equitable sharing of scarce network resources by restricting the resource allocations of each flow[1]—aggressive or otherwise—to the *fair share*. Second, it can keep networks robust and less prone to potential system manipulation and attacks [103]. Third, by allowing the coexistence of heterogenous end-to-end congestion control algorithms (and lack thereof) in the network [90], it promotes the proliferation of new services over the Internet. Fourth, fair scheduling disciplines can be used together with proper traffic shaping at network ingress to provide service guarantees (e.g., delay guarantees) to user application-flows [43, 89].

Initially designed for scalability and survivability [18], the Internet from the outset lacks network mechanisms that ensure fairness and protection among application-flows.[2] This is mainly because all packets—regardless of the nature of the source generating them— receive *egalitarian best-effort service*. Consequently, user connections injecting more bytes per time unit receive proportionally more service, at the expense of others with lower rates.

There are two general router mechanisms proposed to address the above problem: (1) perflow fair queueing or scheduling, and (2) buffer or queue management mechanisms. The vast majority of the former proposals are both *stateful and complex*. Being stateful, fine-grained per-flow state information is maintained at the routers. The amount of state information may exceed the capacity of the high-speed memory available in the routers. In addition, the application-flows are isolated into a large number of queues [72]. This requires complex and dynamic management of the queue architecture. In contrast to the perflow fair queueing schemes, most of the queue management mechanisms are relatively simpler, but lack the required flow protection level to be practically useful.

This thesis proposes and analyzes several *stateless, partially stateful (a.k.a statelet), and scalable router algorithms* for ensuring flow fairness and protection in the Internet. This chapter serves as the introduction to the thesis. We begin with Sec. 1.1 which identifies the general challenges associated with building efficient and scalable router mechanisms, followed by the motivation for the research work done in Sec. 1.2. Sec.1.3 describes the research methodologies adopted in this work. In

[1] A flow can be defined as a sequence of packets having the same signatures in the packet header. A common signature is the 5-tuple in IPv4 (which consists of source and destination IP addresses and port numbers and protocol type), or the flow label in IPv6.

[2] Fairness is simply an approximate equality between flows' throughput, see Chapter 2 for more detailed notions. It generally implies flow protection, but not vice versa [57]. By protection in this thesis, we mean that the resource shares of misbehaving flow(s) are restricted or bounded.

Sec. 1.4, our key research contributions are summarized. The research papers that make up the thesis are listed in Sec. 1.5. Finally, Sec. 1.6 presents the guidelines for reading the rest of the thesis.

1.1. The Grand Challenges

The Internet is an open-access integrated services network. End users, malicious or otherwise, can freely grab scarce network resources at the rate of their demand, potentially causing congestion at various points in the network. Under congestion, the Internet is equipped neither to provide QoS guarantees nor to protect best-effort connections from high-bandwidth (potentially malicious) flows. Current designs of network architectures for providing flow protection or service differentiation among flows face several closely inter-twined challenges:

- *Internet line speeds are increasing.*

Thanks to optical technology, Internet line speeds are increasing beyond few giga-bits per second. Unfortunately, this decreases the time-budget required for per-packet processing at the routers' ports, see Fig. 2.1(b). When a packet arrives to a router, the input port performs a lookup to determine the output port, and then forwards the packet to it through the switching fabric. Common speed constraints here include both the lookup operation which invokes frequent memory access and the forwarding at the switching fabric. Once forwarded, the packet may still need to be queued at the router's port awaiting transmission. With dwindling time budgets for per-packet handling, the overall forwarding operations should be simple to scale. A notable example of a scalable design is the current Internet which adopts very simple router schemes—FIFO packet scheduling and Drop-Tail buffer management.[3]

- *Internet traffic volume is increasing.*

The second challenge stems from the enormous amount of traffic generated by users. The amount may even overwhelm the available link capacities [65, 1]. Even though, many data networks are over-provisioned and remain under moderate utilizations in the core [73], this is not true everywhere, e.g., at access links and core networks in developing countries [39]. User flows are therefore likely to experience congestion at both ends of communication, and occasionally in the core during times of peak transient demand. Router architectures should scale with the amount of traffic they handle and be able to allocate resources efficiently [53]. It is also preferable that routers do not keep recent histories or states for a large number of flows they serve.

- *Application or service requirements are complex.*

[3] The recent ISC Domain Survey shows an increase by a factor of over 150 in the number of Internet end hosts between the years 1995-2012 [3].

Today, Internet applications vary in their service requirements. Some applications require end-to-end QoS (e.g., delay guarantees), and best-effort connections require flow protection. Normally, provision of service differentiation and flow protection entails more complex per-packet processing, in a direct collision course to the scalability requirements.

Designing network architectures with balanced trade-offs in efficiency, service quality and scalability has been a difficult hurdle.

1.2. Thesis Motivation

The Internet provides a basic egalitarian point-to-point best-effort service [11], with no flow fairness mechanisms in place. An arbitrary flow can increase its resource share merely by increasing its demand. This may lead to congestion in the network and service deterioration to other competing flows. Generally, there are two general approaches proposed to address the flow fairness problem:

1. End-to-end (e2e) based algorithms implemented at the end hosts (e.g., TCP[4])
2. Router based mechanisms; there are two broad sub-categories
   (i) Queueing or scheduling mechanisms
   (ii) Queue or buffer management mechanisms

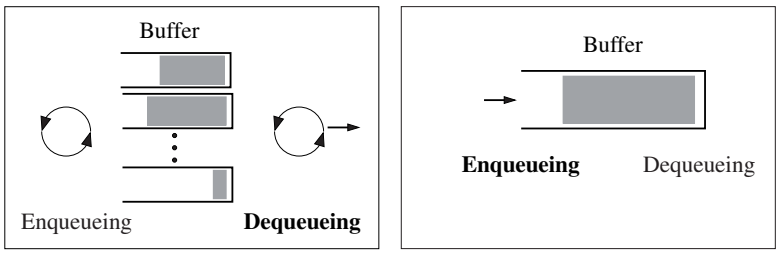
Before highlighting the limitations, we provide the definitions as used in this thesis. See Sec. 2.1 for detailed account of the differences. In this thesis, we refer to those mechanisms at the router port that transmit packets as *queueing or scheduling mechanisms*. On the other hand, those algorithms that decide which packet to enque and / or which packet to drop are termed as *buffer management or queue management mechanisms*. That is, queueing or scheduling mechanisms are executed at dequeueing whereas the buffer (queue) management mechanisms are executed at enqueueing. The differences are pictorially displayed in Fig. 1.1.

Both e2e-based and router-based mechanisms proposed in the literature have limited success in addressing the generic problems stated in Sec. 1.1. Specifically, one or more of the following limitations can be identified in the proposed mechanisms: (1) inefficiency, (2) lack of generality to all flows, and (3) lack of scalability. The limitations are discussed next.

1.2.1. Heterogeneity of Congestion Control

Most of the classic Internet flows are elastic; that is, the flows can adapt their sending rates in response to perceived network congestion and, apart from reliable delivery of their packets, have no strict QoS requirements. These flows could therefore universally adopt TCP as their end-to-end congestion control algorithm. Traditional TCP is coupled with Additive-Increase-Multiplicative-Decrease

[4]Transmission Control Protocol



(a) Perflow queueing or scheduling. (b) Buffer or queue management.

Figure 1.1.: Conceptual models of flow protection architectures (note the bold label indicate the major operation): (a) perflow queueing is a policy of packet dequeueing (or transmitting), and this action is often preceded by sorting of the packets in the queues; (b) queue management is a packet enqueueing and/or dropping policy acting when a packet arrives.

(AIMD) [48, 16] as its congestion avoidance algorithm. When a loss is detected, the flows cut their sending windows by half. Otherwise, the flows increase their sending windows at most by 1 TCP segment. In an environment with synchronized losses, as in Drop-Tail routers, the AIMD algorithm allows TCP sources to converge to fairness [16, 32].

However, the Internet traffic is no longer controlled by a single AIMD algorithm. Apart from the various flavors of TCP congestion avoidance algorithm (e.g., Vegas [12], Highspeed TCP [32], CUBIC [45], Binary Increase Congestion Control [100], Scalable TCP [56]), Internet is also characterized by non-TCP congestion control algorithms, e.g., User Datagram Protocol (UDP) [82], Datagram Congestion Control Protocol (DCCP) [58]. The algorithms react differently to network congestion in their steady state. For example, upon detecting a packet loss, the classic AIMD reduces its sending window by $\frac{1}{2}$, CUBIC by $\frac{3}{10}$, STCP by $\frac{1}{8}$, whereas UDP continues to send packets undeterred.

The heterogeneity in deployed e2e congestion control algorithms in the Internet is expected to rise. In the face of diverse reactions to a packet loss, Internet flows are not expected to converge to fairness in their throughput shares. Therefore, achieving flow protection through e2e mechanisms may require universal adoption of a particular TCP algorithm, e.g., the AIMD algorithm. This limits the generality of the solution to other types of flows, which is the second limitation mentioned above. A related problem with the end-to-end schemes is the apparent lack of fairness enforcement since end users may adopt no congestion control, or end users are free to adopt any greedy e2e congestion control algorithm. Chapter 7 discusses related issues.

Having discussed e2e mechanisms and their limitations, we proceed to router algorithms proposed for flow protection and their limitations.

1.2.2. Complexity of Perflow Queueing

Queueing algorithms are literally packet **dequeueing** policies. We call queueing algorithms designed for flow fairness *perflow fair queueing* or *perflow queueing* for short. They determine, at each transmission epoch, which packet of a flow from those backlogged in the buffer to send when the scheduler becomes idle, see Fig. 1.1(a). Many of the proposed fair queueing mechanisms are technically superior and can provide intricate flow-level fairness in fine time granularity. However, they come with huge costs for practical implementation in high-speed routers because they are both stateful and complex. Well-known examples of such schemes are SFQ [44], SCFQ [41], WFQ [20, 81], WF^2Q [7]. Characteristically, these schemes isolate each flow into separate queues, and packet transmission is typically preceded by sorting among packets found at the head of these queues. While flow isolation acts as a fire-wall and allows for better flow protection, it also requires complex buffer partitioning and scheduling states (e.g., the pointers to each packet queues) required to maintain the complex queue structures [57]. With increasing line speeds in routers, the time budget required for per-packet handling continuously decreases.

Additionally, the sheer amount of per-flow information (or history) to be maintained means that the routers must be shipped with large expensive high-speed memories. A notorious example is the well-known WFQ, which needs to keep perflow state information of all backlogged flows both in the *packet system* and *fluid system* that WFQ is designed to emulate [81].

1.2.3. Inefficacy of Buffer Management Mechanisms

The buffer management (a.k.a queue management) mechanism at the router output port drops packets when a packet arrives to a buffer which is full, or whose thresholds are exceeded. Since packet losses are interpreted by most TCP sources as network congestion, buffer management mechanisms play a critical role in the throughput performance of TCP flows in the Internet.

Based on a single queue, buffer management mechanisms are comparatively simpler in design, see Fig. 1.1(b). With no flow isolation, however, they *lack the quality of flow protection afforded by per flow queueing mechanisms*. A typical example is Random Early Detection [10, 38, 36] which defines a global packet drop probability applicable to all traversing flows. However, the global drop rate does not differentiate losses among flows and hence does not yield quality flow protection. To address this problem, a host of RED extensions with differential perflow drop rates have been proposed, e.g., FRED [64] and RED-PD [66]. These schemes are partially stateful or statelet since they keep perflow accounting on a subset of the flows (usually the high-bandwidth flows). By leveraging the flow information, different perflow drop rates can be defined based on the nature of the flows. A potential concern is that the perflow drop rates of these classical schemes are usually optimized to a particular TCP congestion control algorithm, risking generality to other congestion control algorithms.

1.3. Research Methodology

Considering the complexity of perflow queueing and inefficacy of buffer management schemes, two logical research directions can be followed in the design of simple and efficient router fairness schemes.

1. Avoid the complexity of the perflow queueing schemes while simultaneously approximating their quality flow protection.

2. Improve the flow protection quality of queue (buffer) management schemes while simultaneously retaining their simple and stateless nature.

We explore both directions in this thesis, resulting in both statelet and stateless mechanisms. Our proposed mechanisms share the simple scalable architecture of queue management mechanisms in at least three ways (see Fig. 1.2): (1) a design based on single-queues shared by all flows, (2) the major packet operation is undertaken at enqueueing when a packet arrives, and (3) the dequeueing operation involves the transmission of the packet found at the queue head, making this operation as simple as that of FIFO queueing. Our first major undertaking is the provisioning of approximate flow fairness. This requires sorting of arriving packets into the single-queue based on computed tags. Our second major task is to retain all the scalable features of a simple and stateless queue management mechanism called CHOKe, but at the same time generalize it as an improved flow protection framework.

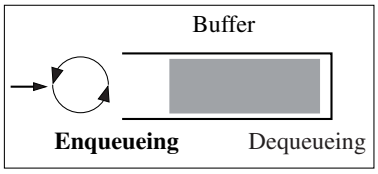


Figure 1.2.: Conceptual model of the proposed architecture. It sorts packets upon enqueueing, and is coupled with a very simple packet dequeueing operation.

For all works in the thesis, testing and performance evaluations of the proposed schemes are carried out through extensive packet-level ns2 [29] simulations. Simulation offers a great deal flexibility in the choice of traffic characteristics, link and buffer capacities and complexity of network topologies. Depending on the nature of experiment and the level of confidence required, simulation experiments are typically replicated between 10-500 times. For some works that can be simplified, particularly those studying the steady state and transient queue behaviors of the stateless and simple flow protection architectures reported in Chapters 4 and 5, analytical models are developed and the results are validated through simulations. Analytical modeling gives more credibility to results and further insights (e.g., limit and asymptotic behaviors). Yet the application of analytical models is limited since the scenarios should generally be simple enough to be mathematically tractable. For

the more complex general scenarios of Chapter 6, we therefore use simulations. For result validation of the simple scenarios in Chapter 3, and partial results in Chapter 6, we trace individual packets through the network. Alternatively, validation can also be carried out through real experiments which offer much more credence, yet are inflexible and more costly in development time. For experiments featuring various TCP algorithms in Chapter 7, we port the real Linux implementations [99] of the TCP algorithms to the ns-2 simulations.

1.4. Contributions

This objective of this thesis is to develop router mechanisms that are simple, practical, preferably stateless, highly efficient in resource utilization and can provide high quality of flow fairness among flows. The proposed mechanisms are all based on single queues. Due to the nature of the problem, some sacrifices may be necessary. For providing tight flow control, our proposed scheme is not only simple and effective, but also completely stateless. For providing a higher grade of flow fairness, on the other hand, we require a statelet single queue framework which sorts packets on arrival. The amount of flow state is, however, limited and is bounded by the buffer size. The proposed scheme is highly fair even when flows exhibit diverse traffic characteristics and / or adopt different congestion control algorithms.

We summarize our key research contributions as follows.

- We proposed and evaluated a partially stateful (a.k.a *statelet*[5]) flow protection architecture, see Fig. 1.2, namely *Approximate Fairness through Partial Finish Time* AFpFT, which possesses a host of desirable properties. Firstly, unlike perflow fair queueing schemes, there is no complex partitioning of the packet buffer. Secondly, packet dequeueing is a simple task of transmitting the head-of-the-line packet. The scheme is therefore lightweight and scalable. The more complex operation is the sorting when packets arrive to the queue. This may not be a big problem since the buffer (backlog) sizes in real networks are generally limited and the packet sorting operation can be optimized. Under AFpFT, an arriving packet is assigned a tag which determines the packet's position in the queue. This enables flows to be treated differently based on their recent shares of bandwidth. Particularly, low and medium rate flows get both time (scheduling) and space (scheduling) priority. Overall, AFpFT is a statelet and scalable scheme.

- We challenge the purported inter-operability between the traditional TCP AIMD algorithm and the various deployed high-speed TCP congestion avoidance algorithms. This study is conducted in the presence of diverse queue management mechanisms at bottleneck links. With increasing heterogeneity of deployed TCP algorithms in operating systems, such a study is crucial for understanding the stability of the Internet. However, literature results in this

[5]Only perflow state for those having packets in the queue are kept.

regard are limited. For our specific study, we consider six different TCP algorithms (namely AIMD, Vegas, HSTCP, CTCP, BIC and CUBIC) and how fairly they share a high-speed bottleneck link adopting one of the following five router mechanisms, namely Drop-Tail, RED, FRED, CHOKe and AFpFT. In the presence of any of the router schemes at the bottleneck, the various TCP algorithms fail to be fair to each other or to the traditional TCP. Particularly, we find that short RTT BIC and HSTCP flows are highly aggressive in the use of link bandwidth. The aggression is severe under Drop-Tail and CHOKe queues. When all constituent flows have large comparable RTTs, however, CUBIC flows turn out to be the most aggressive. We find an explanation for this interesting phenomenon. An exception to the widespread unfairness among TCP flows is our proposed scheme AFpFT which battles both the TCP heterogeneity at traffic sources and differences in the round-trip-times of flows to ensure a high fairness score among the flows. In addition, AFpFT is the most efficient in resource utilization as its link utilization is also the highest.

- We introduce and extensively analyze a suite of active queue management schemes called Geometric CHOKe (gCHOKe), each indexed with a parameter `maxcomp`. gCHOKe rewards successful matching of flow packets with a bonus matching trial. For each packet arrival to queue, the matching process stops either when a total of `maxcomp` matching trials are executed or when a no-match is encountered. The arriving packet and all matched packets are dropped. The bonus trials improve the traffic controlling power of the scheme. When `maxcomp`$=\infty$, the number of matching trials per packet arrival is geometrically distributed. The upper bounds both in buffer share and link utilization of the unresponsive flow improves by about 20% in comparison to those in CHOKe. In addition, up to 14% of the bottleneck link capacity can be saved from the unresponsive flow, allowing responsive or rate-adaptive flows to obtain a better share of resources in the router. CHOKe turns out to be the simplest case of gCHOKe with `maxcomp` set to 1.

- Most studies on CHOKe are limited to the steady state equilibrium behavior of the queue serving many long-lived TCP flows and a UDP flow of fixed source rate. We relaxed the assumptions by allowing the UDP source rate to change over time. Before the queue settles into a new steady state following the UDP rate change, we find that the CHOKe queue first enters into a transient regime characterized by "strange" queue behaviors. We provide an analysis (the first to the best of our knowledge) that characterizes this very intriguing or "perplexing" behavior of CHOKe during the transient regime. For example, when the UDP rate change is upward, the utilization suddenly drops during the transient time, before eventually stabilizing to a new equilibrium value. Our analysis explains this behavior, which is validated through simulations. Similar analysis can be conducted for all gCHOKe variants.

1.5. Publication

Papers Included in the Thesis

The thesis is based on the papers [24, 23, 27, 26, 25, 28] written by the candidate under the supervision of Prof. Yuming Jiang. The papers are listed below:

[A] Addisu Tadesse Eshete and Yuming Jiang. "On the Flow Fairness of Aggregate Queues." *In Proceedings of the First Baltic Congress on Future Internet Communications (BCFIC)*, Riga, Latvia, February 2011.

[B] Addisu Tadesse Eshete and Yuming Jiang. "Approximate Fairness through Limited Flow List." *In Proceedings of the 23rd International Teletraffic Congress (ITC)*, San Francisco, USA, September 2011.

[C] Addisu Tadesse Eshete and Yuming Jiang. "Protection from Unresponsive Flows with Geometric CHOKe." *In Proceedings of the 17th IEEE Symposium on Computers and Communications (ISCC)*, Cappadocia, Turkey, July 2012.

[D] Addisu Tadesse Eshete and Yuming Jiang. "Generalizing the CHOKe Flow Protection." *Computer Networks journal, Elsevier*, September 2012.

[E] Addisu Tadesse Eshete and Yuming Jiang. "On the Transient Behavior of CHOKe." *Submitted to IEEE/ACM Transaction on Networking*, November 2011.

[F] Addisu Tadesse Eshete, Yuming Jiang and Lars Landmark.[6] "Fairness among High Speed and Traditional TCP under different Queue Management Mechanisms." *To appear in Proceedings of the 8th Asian Internet Engineering Conference (AINTEC)*, November 2012.

Other Papers by the Author

[G] Addisu Tadesse Eshete and Yuming Jiang. "Statelet Fair Queueing." *In Workshop on Traffic Engineering and Dependability in the Network of the Future*, Trondheim, Norway, June 2011.

- This is an early version of Chapter 6.

[H] Kashif Mahmood, Yuming Jiang and Addisu Tadesse Eshete. "On The Modeling of Delay and Burstiness for Calculating Throughput." *In Proceedings of the 33rd IEEE Sarnoff Symposium*, New Jersey, April 2010.

[I] Addisu Tadesse Eshete and Yuming Jiang. "Flow Aggregation Using Dynamic Packet State (workshop paper)." *In 16th EUNICE/IFIP WG 6.6 Workshop*, Trondheim, Norway, June 2010.

[6] [F]: Lars Landmark participated in the discussion of the ideas and results.

[J] Addisu Tadesse Eshete, Andrés Arcia, David Ros and Yuming Jiang. "Impact of WiMAX Network Asymmetry on TCP." *In Proceedings of the IEEE Wireless Communications and Networking Conference (WCNC),* Budapest, Hungary, April 2009.

1.6. Thesis Outline

The rest of this thesis is structured as follows. The next chapter is devoted to general background information on the router model, flow fairness and de facto and proposed router fairness mechanisms. The chapter elaborates on the notions of fairness as used in this thesis and the metrics used to express it, and presents the paradigms followed in the technical design of router fairness algorithms, backed by some example router algorithms proposed in the literature. The design trade-offs between complexity and the quality of flow protection is emphasized.

Chapters 3-7 constitute the major body of research work done. Chapter 3 describes the intuitive simplification of perflow queueing mechanisms to a stateful single-queue mechanism serving all flows over a single link. This work identifies an important design challenges in single queue mechanisms, which is the synchronization of losses to certain flows that in turn can significantly undermine the flow fairness of such mechanisms.

In Chapter 4, we develop lightweight generalizations of the CHOKe flow protection algorithm without tampering with its simple and stateless design principle. The generalization principle is: "repeat packet flow matching trials for an arriving packet until a no-match is encountered or as limited by an integral control parameter." As it turns out, this simple rule not only generalizes CHOKe, but also radically improves its flow protection.

A first study on the transient regime of CHOKe, initiated when the rate of unresponsive flow changes is undertaken in Chapter 5. In order to explain and characterize the transient behavior, this chapter presents two models—one based on "the rate reservation argument" and the other on ordinary differential equations (ODE), which is validated by ns-2 simulations in the latter part of the chapter.

Note that the mechanism proposed in Chapter 3 is both stateful, and prone to severe unfairness due to the loss synchronization problem. In Chapter 6, we extend and enhance it as a general router fairness framework called AFpFT. By leveraging the limited flow state maintained at the router, a drop policy is devised to counteract the synchronization. The scheme is shown to be highly fair, statelet and simple—all hallmarks of a scalable architecture.

Chapter 7 uses extensive simulations to compare the throughput performance of six different TCP congestion control algorithms in the presence of several well-known router mechanisms at the bottleneck links. Among other things, the study shows that our AFpFT mechanism proposed in Chapter 6 is indeed fair in a traffic mix of heterogenous TCP congestion control algorithms.

Finally, Chapter 8 concludes the thesis by presenting a summary of our main results and limitations, and discussing potential directions for future work.

2. Background and State of the Art

Since the theme of this thesis is on router-based flow fairness or protection mechanisms, this chapter describes some general background relevant in that regard.

The chapter begins with an architectural view of routers and IP networks as used in the thesis. Sec. 2.2 presents some of the various notions of fairness and the performance metrics used in this thesis to express flow fairness. Traditionally, Internet fairness control is achieved through congestion control algorithms at end users. Sec. 2.3 is devoted to some discussion on TCP congestion control. We present the state of the art on router based fairness mechanisms and their categories in Sec. 2.4. Finally, Sec. 2.5 engages in general discussion on contentious and open issues related to fairness.

2.1. IP Router Architecture

Routers are the building blocks of packet-switched networks such as the Internet, see Fig. 2.1. They consist of ports to which packets arrive ("input ports") and ports from which packets depart ("output ports"). See [62, 22] and references therein for more thorough presentations. Typically, a router port defines a separate network. Router ports are interconnected by high-speed switching fabrics. A primary goal of routers is to switch packets arriving at an input port to an output port using the switch fabric, or equivalently from one network to another. This is usually determined by destination address carried by the packet and a lookup of the routing table maintained by the router.

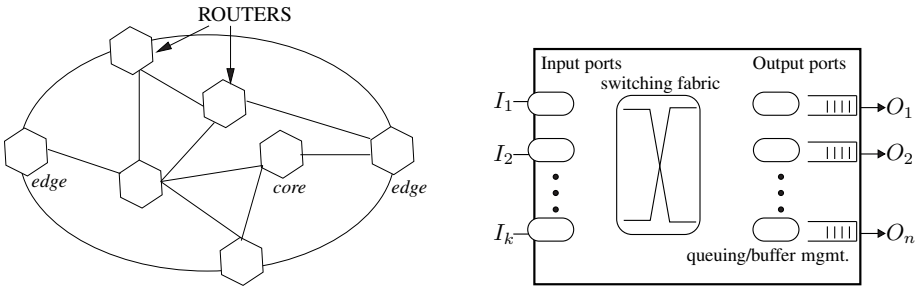


Figure 2.1.: High-level view of (a) a network and (b) a router model.

More often than not, several packets destined to the same output port may arrive to several of the input ports at the same time. Since the output ports are probably just as fast as the input ports which are bufferless, some of the incoming packets may need to be dropped at the output port, while some may need to be queued at the output port and scheduled for transmission. This is where the various router based queueing and buffer management mechanisms discussed in Chapter 1 are required, (see Figs. 1.1(a), 1.1(b), 1.2).

The above architecture describes what is called *output queued* (OQ) switches or OQ routers. OQ switches are generally impractical since the speeds of the switch fabric should increase both with line speeds and number of input ports. They are, however, easier to model and understand.

Practical routers include, among others, input queued (IQ) switches with buffers only at the input ports, and combined input output queued (CIOQ) switches with buffers both at input and output ports. When packets arrive to an input port of an IQ router equipped with a switch fabric not fast enough (relative to input line speeds) or busy with the transfer of other packets from other ports, the new arrivals may be queued at the input port for later transmission. Even though IQ switches are common in practice, they are usually modeled as output queued switches for ease of understanding [22]. Hence, our subsequent discussion is on OQ switches.

Since the input ports of output queued switches are bufferless, they are required to operate at line speed, which could be very high in core routers. That means, a lookup and packet forwarding (to the appropriate output port) have to be completed before the next packet arrives at the input port. There are several optimizations to expedite the lookup process, e.g., using content addressable memories (CAMs) or other type of hardware-based lookups, lookups based on efficient data structures or caches, see [97] and references therein. It is possible that several input ports—operating at line speeds—may receive packets to be forwarded to the same output port, instantly building up the queues in the process. In order for the output port to cope with this challenge, the packet scheduling (queueing) and dropping mechanisms adopted at the router should be computationally simple.

2.2. Notions of Fairness

A simple meaning of fairness is that all backlogged flows have approximately equal shares of the network resources, typically bandwidth. There are several notions of fairness, e.g., idealized fluid fairness [81], max-min fairness [8] and TCP based fairness. We explain some of these concepts next ([33] lists additional pointers).

2.2.1. Idealized Fairness

In the idealized fluid fairness (e.g., found in the Fluid Fair Queueing (FFQ) scheme), flows can not only transmit in infinitely divisible entities or bits, but also simultaneously. A flow is characterized by a flow weight / share r_f which quantifies the normalized service w_f it can receive in a unit of time. Suppose flow f is backlogged

during (t_1, t_2), the normalized service received by the flow is:

$$w_f(t_1, t_2) = \frac{W_f(t_1, t_2)}{r_f} \tag{2.1}$$

where $W_f(t_1, t_2)$ is the amount of service received by the flow (i.e., number of flow bits transmitted) during (t_1, t_2).

The following relation holds for two arbitrary flows f and g continuously backlogged during (t_1, t_2) or symbolically $f, g \in B(t_1, t_2)$.

$$w_f(t_1, t_2) = w_g(t_1, t_2) \qquad \Rightarrow \qquad \left| \frac{W_f(t_1, t_2)}{r_f} - \frac{W_g(t_1, t_2)}{r_g} \right| = 0 \tag{2.2}$$

(2.2) says that FFQ provides perfect weighted fairness to competing flows. However, FFQ is hypothetical because (i) flows can practically send only in units of packets and not in bits, (ii) a real server or router cannot serve several flows at the same time. Nevertheless, the objective of the *real* packetized approximations of FFQ such as WFQ, SFQ and SCFQ is to keep the fairness measure $|\cdot|$ in Eq. 2.2 as close to zero as possible [44, 41].

2.2.2. TCP Fairness

TCP fairness is basically fulfilling the TCP-friendliness criterion: A flow is said to be *TCP fair* or *TCP friendly* if it has approximately the same flow rate as a standard TCP [48, 5] or TCP Friendly Rate Control [84] flow under the same network conditions (e.g., similar packet loss rates and round-trip times). The average TCP flow throughput T (pkts/sec) is governed by the square-root formula [35] (see [75] for a more general TCP throughput formula),

$$T = \frac{1}{RTT}\sqrt{\frac{3}{2p}} \tag{2.3}$$

where p and RTT are the loss event rate and the round-trip time, respectively. Equivalently, the average TCP sending window W in packets is given by the *TCP response function*,

$$W = \sqrt{\frac{3}{2p}} \tag{2.4}$$

Note that TCP-fairness does not imply *RTT-fairness*; specifically, the throughput ratio is a power law of the inverse RTT ratio. When packet losses are synchronized, as is common in Drop-Tail routers, the ratio between two TCP flow throughput of different RTTs is $T_1/T_2 \sim \left(\frac{RTT_2}{RTT_1}\right)^{\alpha}$ [100], where $1 < \alpha < 2$. That means, short RTT TCP flows gain more bandwidth than long RTT flows.

2.2.3. Max-min Fairness

Another well-known fairness is the max-min fairness, which naturally favors smaller rate flows. It is an allocation where a flow resource share cannot be increased without decreasing the share of a flow with a smaller resource share.

For a congested link with output capacity C and serving n flows, the max-min fair share rate Φ_{share} uniquely satisfies the condition:

$$C = \sum_{i=1}^{n} \min(\Phi_i, \Phi_{share}) \tag{2.5}$$

where Φ_i is the demand (e.g., incoming rate) of flow i.

Example: Consider 4 flows with incoming rates 2, 4, 6 and 8 competing for a resource of capacity C =12. Applying (2.5), the fair share becomes $\Phi_{share} = \frac{10}{3}$. So the first flow obtain its full demand of 2, and the remaining three are constrained with the resource share rate of $\frac{10}{3}$.

2.2.4. Fairness Score

How do we quantify the fairness among resource allocations? One well-known fairness score is Jain's fairness index [16]. For a resource shared by n connections, where the resource share of connection i is denoted by $x_i, i \in \{1, \cdots, n\}$, the fairness index f becomes,

$$f = \frac{(\sum x_i)^2}{n \sum x_i^2} \tag{2.6}$$

Note that $1/n \leq f \leq 1$. The fairness score becomes m/n when $m \leq n$ connections share the full resource equally. The ideal fairness score is 1 which is achieved when $x_i = 1/n$ for all i. A completely unfair system where a single connection grabs all the shared resource has a fairness score of $1/n$.

2.3. End-to-end Congestion Control Algorithms as Fairness Mechanisms

Congestion control is a (typically distributed) algorithm to share network resources among competing traffic flows [79]. Apart from the efficiency in the use of network resources (see Sec. 2.5), one objective of congestion control is to provide fairness among competing flows [9, 33, 63]. This is logical since fairness as a desirable performance goal is more meaningful under congestion situations. As a result, mechanisms to enforce fairness in the Internet are often proposed as congestion control mechanisms, hence our subsequent discussion on congestion control algorithms.

Internet congestion is generally a *network phenomenon* caused by *traffic sources at end hosts*. Therefore, there are generally two parts to Internet congestion control: (1) those implemented at the end-hosts to reactively or proactively respond to

network connection, called in [55] primal congestion control, and (2) those implemented in the routers, called in [55] dual congestion control. Default approach in the Internet is based on the former, as Internet mainly depends on TCP-based rate control with the network involvement reduced to announcing limited congestion information implicitly in the form of packet drops and delays, or explicitly using Explicit Congestion Notification (ECN) marks [84].

The end-to-end congestion control is typically exemplified by the Standard TCP [48, 5, 4] which provides reliable in-order delivery of bytes. Traditional TCP adopts the Additive Increase Multiplicative Decrease AIMD($1,\frac{1}{2}$) congestion avoidance algorithm, whereby a TCP source increases its window by at most 1 TCP segment per loss-free round-trip time, but decreases it by $\frac{1}{2}$ upon a packet loss event. The AIMD algorithm also allowed Internet connections to converge to fairness [16]. Recall from Sec. 2.2.2 earlier that traditional TCP flows under similar network conditions obtain compatible flow throughput. By ensuring equitable sharing among Internet flows and averting the congestion collapse of the 1980s [71, 48], the traditional TCP algorithm has contributed greatly to the robustness of the Internet.

The success of the traditional algorithms is not permanent, however, as their limitations in adapting to new networks (e.g., high-speed and large delay links) and in fulfilling applications' QoS requirements become increasingly evident. We specifically point out two issues next (see also the closely related discussion in Sec. 1.2.1).

- The AIMD window increases are deemed too conservative to efficiently utilize the vast link capacities available now. As a result, a host of high-speed TCP congestion avoidance implementations begin to take foothold. Refer to Chapter 7 for details.

- TCP provides reliable delivery of bytes, and this is done by retransmitting lost or delayed packets. Many other Internet applications may prefer timely delivery to reliable delivery. These applications may not even adopt congestion control mechanisms at all, and hence send packets undeterred in the face of network congestion. Some applications implement congestion control algorithms at the application layer, rendering TCP based congestion control mechanisms redundant and unnecessary.

Because of the above limitations, a large number of Internet end users may adopt other forms of congestion control algorithms, or no congestion control at all. In the face of increasing heterogeneity, the traditional TCP algorithms may no longer work optimally (fairly). For example, TCP flows usually relinquish the use of network resources in the presence of aggressive connections (e.g., UDP flows). Hence, the traditional TCP algorithms fail to provide performance incentives for widespread user adoption. In order to impose fairness despite the heterogeneity in deployed TCP congestion control algorithms, the intervention from routers may be necessary. The next section deals with router-based fairness mechanisms.

2.4. Router-Enforced Flow Fairness Mechanisms: A Taxonomy

The architectural blueprint of the original Internet is the end-to-end design principle [85] which argues:

> *The function in question can completely and correctly be implemented only with the knowledge and help of the application standing at the end points of the communication system. Therefore, providing that questioned function as a feature of the communication system itself is not possible.*

Following the above principle, the bulk of network communication functions were moved to end user applications. The routers keep no fine-grained state information about the flows / connections they are serving—a design approach called "fate-sharing" [18]. As a direct consequence of the approach, TCP at Internet end hosts implement sophisticated congestion control and flow control functions. And the routers, on the other hand, simply forward packets based on a simple FIFO principle and, when the packet buffers become full, drops incoming packets based on a simple Drop-Tail buffer management strategy.

The end-to-end design principle and the "fate-sharing" approach greatly contribute to the scalability and robustness of the Internet. But as it turns out, without cooperation between the end hosts, a simple network cannot enforce any sort of flow fairness or protection. Internet started to become vulnerable to users, who intentionally or unintentionally, do not adopt any kind of end-to-end based congestion control. Users lacking congestion control can control the scarce network resources at the rates of their traffic injection. So starting from the work of Nagle [72], sophisticated router-enforced fairness mechanisms started to pop up, deviating from the "traditional" scalable design principles in the process. In general, two broad categories of router-enforced fairness mechanisms have been proposed (see also Sec. 1.2):

1. Perflow (fair) queueing or scheduling mechanisms which are typically exemplified by SFQ [44], WFQ [20], WF^2Q [7] and SCFQ [41]

2. Queue or buffer management mechanisms which are typically exemplified by RED [10, 38, 36]

The two mechanisms can be thought of as resource allocation policies; specifically, they allocate the link bandwidth and the buffer space, respectively. Queueing mechanisms dequeue packets over the outgoing link, while queue management mechanisms drop incoming packets or admit them into the queue.

First, we mention the basic de facto router schemes in the Internet. Default queueing scheme is FIFO which serves or transmits packets based on first come first served basis. And the default queue management scheme is Drop-Tail that drops incoming packets when the buffer is full, and admits them otherwise.

2.4.1. Perflow Queueing Mechanisms

Recall from Sec. 2.2.1 that perfect fairness can be provided by the fluid fair queueing FFQ. FFQ is, however, unrealistic. Researchers have therefore proposed various packet-level approximations to FFQ. Prominent real packetized approximations include Weighted Fair Queueing (WFQ) [20, 81], Worst-case Weighted Fair Queueing WF^2Q [7], Start-time Fair Queueing (SFQ) [44] and Self-Clocked Fair Queueing (SCFQ) [41]. These packetized algorithms attempt to closely emulate FFQ's perfect fairness. For example, the service (in bits transmitted) received by a backlogged flow under WFQ can only lag behind that of the corresponding flow service under FFQ by no more than the maximum packet length.

All the above algorithms use the concept of *virtual time* $v(t)$ and finish timestamp $F(\cdot)$ and start timestamp $S(\cdot)$ defined for each arriving packet. Let us denote,

$$\begin{array}{rcl} v(t) & : & \text{the virtual time of the server or queue at real time } t \\ p_f^j & : & \text{the } j^{th} \text{ packet flow } f \\ S(p_f^j) & : & \text{the start timestamp of } p_f^j \\ F(p_f^j) & : & \text{the finish timestamp of } p_f^j \end{array}$$

The computations of the packet timestamps follow. *Upon arrival of packet* p_f^j *to queue, the timestamps for the arriving packet* p_f^j *become:*

$$S(p_f^j) = \max(v[A(p_f^j)], F(p_f^{j-1})) \qquad j \geq 1 \tag{2.7}$$

$$F(p_f^j) = S(p_f^j) + \frac{l_f^j}{r_f} \qquad j \geq 1 \tag{2.8}$$

where $v[A(p_f^k)]$ is the server virtual time at the time of p_f^k's arrival to queue, l_f^k is packet length, and r_f is the flow's weight.

A major difference between the schemes is how the packet's service tag is defined, which in turn impacts how the packets are scheduled for transmission in the system. For SFQ, a packet's service tag is the start timestamp of the packet, hence the name of the scheme. However, for WFQ and SCFQ, the service tag is the finish timestamp of the packet. Another key difference between the schemes is the definition of the server's virtual time $v(t)$ at time t,

- [FFQ, WFQ]
$$\frac{dv}{dt} = \frac{C}{\sum_{k \in B(t)} r_k} \tag{2.9}$$
where $B(t)$ denotes the set of flows that are backlogged at time t.
- [SCFQ, SFQ] $v(t)$ is the service tag of the packet in service at t. That is, when an arbitrary packet arrives to queue at t, the virtual time is the start tag (in SFQ) or finish tag (in SCFQ) of the packet being transmitted.

Note that under all of the above schemes,

- packets are served in increasing order of the service tags.
- when the server becomes idle (i.e. no busy period), $v(t)$ is reset to 0.

As can be seen, as a direct packet approximation of FFQ, WFQ shares identical server virtual time definition to that of the fluid FFQ. SFQ's and SCFQ's $v(t)$ equals the service tag of the packet under transmission at that time and hence is self-clocked to the schemes. That means, $v(t)$ under SFQ and SCFQ takes up, respectively, the start timestamp and the finish timestamp of the packet undergoing transmission at t.

Perflow queueing mechanisms have a lot of desirable features. By building firewalls around flows, they provide robust protection from potential aggressive or malicious flows. Together with resource reservation and admission policies, they are known to provide strong QoS guarantees at flow granularity levels [80, 57, 43, 89]. For example, the service delivered by WFQ to a flow does not fall behind the corresponding FFQ system by more than one maximum packet size. Because of their strong service semantic, perflow queueing mechanisms are one of the elements in the provision of real-time services in the Integrated Services framework [11].

However, their strong semantic also counts against their scalability for high-speed implementations. A first scalability constraint arises from the isolation of flows into separate physical or logical queues. This requires complex buffer partitioning or a large number of physical queues, see Fig. 1.1(a). Second, in order to determine the most eligible packet to transmit, the head-of-the-line packets in the queues may need to be sorted with respect to their service tags. A third constraint may arise from sorting among queues preceding packet dropping decisions.[1] For example, a common drop policy associated with perflow queueing mechanisms is the Longest Queue Drop (LQD) [94]. When the buffer becomes full, LQD drops packets from the flow queue holding the largest amount of traffic and this calls for sorting among the queues. A fourth scalability limitation is their stateful nature. For example, to compute the start tag of every arriving packet in (2.7), the server needs to hold the finish tag of the previous packet. An extreme example is WFQ. WFQ keeps the information of the flows both for the *packet* and the *fluid FFQ* system which it attempts to emulate. A majority of Internet flows are short-lived [105, 87, 14, 83]. Continuously installing, updating and removing the flow states may be a cumbersome computation at current line speeds.

There are some less complex implementations of perflow fair queueing such as DRR [86]. Packet handling in DRR is a simple $O(1)$ operation. Nevertheless, both the partitioning of the buffer into thousands of logical queues and the maintenance of flow states may still be necessary if DRR is to replicate the fairness of other perflow queueing algorithms.

Recent works [60, 59] show that the actual number of flows that require scheduling at a router is limited and this number does not increase with link speeds. The finding is significant since it directly challenges the earlier claims of complexity associated with high-speed implementation of perflow fair queueing mechanisms.

[1] Sorting is unnecessary if the drop policy is simple, e.g., Drop-Tail.

While the number of flows in progress increases with link speed, the number that requires scheduling is merely in hundreds even on gigabit-per-second links.

2.4.2. Queue Management Mechanisms

The default fairness and congestion control mechanism in the Internet is FIFO queueing together with Drop-Tail buffer management strategy at routers, and TCP congestion control algorithms at end hosts. However, this arrangement has several drawbacks, as documented in the RED manifesto [10] and [38]. First, it may result in a lock-out where only few flows monopolize the buffer space denying service to other flows. This may result in extreme unfairness in the system. Second, Drop-Tail is characterized by *global TCP synchronization* where most of the TCP connections receive notifications to reduce their windows at the same time, resulting in loss of throughput in networks and sustained periods of under-utilization. Third, due to oscillations in queue size and the tendency to maintain full queue under overloaded situations, both the maximum and average queueing delays can be unpredictable and large. Furthermore, Drop-Tail biases against bursty flows.

2.4.2.1. Random Early Detection (RED)

To correct the above shortcomings, researchers proposed the RED scheme [10, 38, 36] and recommended it for widespread deployment. The main idea behind RED is to start dropping or marking packets long before the buffer becomes full, signaling early congestion notifications to sources that can then gracefully slow down. RED is based on a host of tunable parameters: the maximum and minimum queue thresholds $\max_{th}$ and $\min_{th}$, respectively, maximum drop probability $\max_p$ and queue averaging constant w_q. The basic RED operation is briefly described here.

When a packet arrives to a RED queue, there are two basic operations to determine the packet drop rate or drop probability r. First, the average queue length avg is computed as $avg = (1 - w_q) \times avg + w_q \times q$ where q is the current queue size. Second, the packet dropping probability is computed as $r = \max_p(avg - \min_{th})/(\max_{th} - \min_{th})$. Based on the value of avg, we can define three regions of dropping probability. If $avg < \min_{th}$, $r = 0$ and no packet is dropped or marked. Let us call this regime the *no drop* region. Note that packets can still be dropped if the queue overflows. As avg increases from $\min_{th}$ to $\max_{th}$, r increases linearly from 0 to $\max_p$. This corresponds to *early drop* where the incoming packet is dropped probabilistically. When $avg \geq \max_{th}$, $r = 1$, the incoming packet is discarded outright and this deterministic drop is called *forced drop*. Forced drop can also happen when queue q becomes full.

It is fitting to mention some proposed enhancements to the basic RED described above. When avg slowly moves up and down $\max_{th}$, r abruptly alternates between $\max_p$ and 1.0. To avoid this problem, the `gentle` parameter is introduced [30]. With `gentle` set, as avg increases from $\max_{th}$ and $2 \times max_{th}$, r increases linearly from $\max_p$ to 1.0. That means, the region of early drop is expanded when `gentle` is set. Another parameter is `adaptive` [36] which periodically (by default every

0.5s) updates the value of $\max_p$ so that the average queue length avg is maintained in the interval $[\min_{th} + 0.4(\max_{th} - \min_{th}), \min_{th} + 0.6(\max_{th} - \min_{th})]$. This can improve the throughput performance of basic RED.

Just like FIFO and Drop-Tail, RED is oblivious to flow identities. RED applies equal or global drop rate to all flows, regardless of their traffic nature. Depending on the nature of flows—duration and size, presence or absence of underlying congestion control algorithm and the mode of operation (e.g., slow start or congestion avoidance)—equal drop rate does not generally result in fairness in a network with a mixture of various types of traversing flows. For example, some flows are unresponsive to packet drops and do not back off; some flows back off, but immediately grab any spare bandwidth when it becomes available (adaptive-large/responsive-large); and yet some flows back off but do not raise sending rates much even when a spare bandwidth becomes available (responsive-small). The responsive-large flows may be TCP flows operating in congestion avoidance phase whereas the responsive-small flows usually reside in a TCP slow start phase. The reactions to packet loss are different among these flow types: The unresponsive flows send packets undeterred; responsive-large flows halve their congestion windows; responsive-small reset their windows and hence their sending rates.

The above fairness limitation of RED leads to research on queue management (QM) schemes that improve on RED by differentiating losses among flows.

2.4.2.2. Statelet Approximately Fair RED enhancements

An important component missing in RED is a mechanism that *identifies* the flows that contribute greatly to the network congestion and then applies correspondingly higher per-flow drop rates to those flows (in addition to the global RED drop rate). Several such algorithms have been proposed, e.g., Flow RED (FRED) [64], RED with Preferential Dropping (RED-PD) [66], Approximate Fairness through Differential Dropping (AFD) [76], CHOose and Keep for responsive Flows and CHOose and Kill for unresponsive Flows (CHOKe) [78]. *The main differences between the above schemes lie in the flow identification mechanisms adopted.* Some key insights obtained from the underlying RED can be handy for identification:

Observation 1. *Since a RED queue is unleaky with FIFO queueing discipline, the flow packet distributions must be uniform throughout the queue. Therefore, a higher rate flow must also have a correspondingly higher buffer share in RED.*

Observation 2. *Similarly, since the RED drop rate is equal to all flows, a higher rate flow must have a correspondingly larger number of drops.*

The improved flow fairness over RED *does not come for free*, however. The flow identification requires that the queue management schemes keep per-flow accounting on the flows, making them *partially stateful or statelet*. The amount of flow-level state, however, is generally limited, since state is kept only for those flows that have packet(s) in the queue. We now describe the algorithms succinctly, and emphasize the identification engines in each.

FRED attempts to share the buffer space equitably among the active flows. In addition to the RED parameters, FRED defines global parameters $\min_q$, $\max_q$ and $avgcq$, and perflow parameters $strike_f$ and $qlen_f$. $\min_q$ and $\max_q$ are respectively the minimum and maximum number of packets a flow is allowed to buffer. $qlen_f$ holds the number of flow f's packets found in the buffer, and $avgcq$ is the average number of packets buffered per flow. A flow can freely buffer up to $\min_q$ packets without loss, but not more than $\max_q$ packets. If the queue is congested and $qlen_f > \min_q$, a flow f's arriving packet may be dropped with some probability. A flow cannot buffer more than $\max_q$ packets, and every time the flow tries to exceed the $\max_q$ limit in the buffer, the flow parameter $strike_f$ is incremented. Flows with higher $strike_f$'s are not allowed to buffer more than $avgcq$ packets. High bandwidth flows have tendencies to strike too often, raising their probabilities to be punished with higher dropping. Small fragile flows, on the other hand, are guaranteed to maintain $\min_q$ packets in queue and receive better service than otherwise possible.

RED-PD [66] follows from *Observation 2* above. A history of packet drops in a RED queue can reveal the identities of those flows that consume larger shares of the link capacity. High bandwidth flows are more likely to have multiple drops in a time window. These flows are monitored and dropped at a pre-filter to bring their rates to a target bandwidth. The high rate flows, therefore, may suffer two level droppings—at the pre-filter and at the RED queue. The drop at the RED queue, called *the ambient drop* in the paper, is common to all flows. The flows that are monitored and filtered are only a subset with drop counts more than that of a TCP friendly flow under similar network conditions. These flows can be identified from the TCP throughput formula (2.3), where RTT is replaced by a (configurable) target round-trip time RTT_t and p is the ambient drop rate measured at the imbedded RED queue. Note that there is a tradeoff between the quality of fairness and the amount of state (i.e., number of controlled flows) to be maintained. The quality of fairness increases with RTT_t, which unfortunately increases the amount of flow state.

A high rate flow is one with a high incoming rate. In AFD, a history of recent packet arrivals is kept in a shadow buffer which is then used to identify the high bandwidth flows in the queue. From this history, it is possible to estimate both the incoming rate of each flow and the max-min fair share. Note that both the methods of estimating flow arrival rates and of computing the perflow drop rates are borrowed from CSFQ [92], see below. For example, for a flow i with arrival rate Φ_i, the perflow drop rate is given by (2.10). Only flows with higher rates than the fair share are kept in the shadow buffer, since the flow would have zero perflow drop rate otherwise. To reduce the complexity and amount of perflow state, the authors sample the arriving packets with a certain frequency.

All the above active queue management schemes are *statelet* since they keep flow history on packet arrivals (AFD), RED drops (RED-PD), or count of buffered packets (FRED). A completely stateless and simple scheme is CHOKe [78] that can be implemented by a simple tweaking of the RED algorithm. The key idea reads:

> *When a packet arrives at a congested router, CHOKe draws a packet at random from the FIFO buffer and compares it with the arriving packet. If they both belong to the same flow, then they are both dropped; else the randomly chosen packet is left intact and the arriving packet is admitted into the buffer with a probability (based on RED) that depends on the level of congestion.*

Note that the idea follows directly from *Observation 1*. An arriving packet of a high rate flow has a correspondingly high probability of matching a packet randomly picked from the queue. As can be seen, no flow state is required. Despite its lightweight nature, CHOKe is very effective in limiting / penalizing unresponsive flows. Using simple network settings, for example, researchers have proved that as the rate of the unresponsive flow increases without bound, its actual throughput indeed falls to 0 [98, 96, 77]. Such punitive actions by routers to misbehaving flows may be necessary if networks are to avoid potential congestion collapse [31, 71, 35].

2.4.3. Other Schemes

Core-Statleess Fair Queueing (CSFQ) [92] is a scheme proposed for DiffServ-like architectures. CSFQ edge routers are stateful but core routers are stateless, hence the name of the scheme. It introduced a novel method of estimating flow rates based on their packet arrival information (perflow state) at edge (more precisely, ingress) routers. From the knowledge of flow arrival rates, ingress routers can also compute the max-min fair share Φ_{share}, see (2.5). Labeling the estimated incoming rate of flow i as Φ_i and the computed max-min fair share Φ_{share}, the perflow drop rate becomes,

$$d_i = \max\left(0, \frac{\Phi_i - \Phi_{share}}{\Phi_i}\right) = \left(1 - \frac{\Phi_{share}}{\Phi_i}\right)^+ \tag{2.10}$$

If the packet is not dropped upon arrival, the flow's new or outgoing rate becomes $(1 - d_i) \times \Phi_i = \min(\Phi_i, \Phi_{share})$ which is then encoded into the packet header. The packet is then passed on to the next router downstream. Being completely stateless, core routers cannot leverage perflow information. However, they can estimate the max-min fair share Φ_{share} from the knowledge of the aggregate traffic rate and the outgoing link capacity. By computing the Φ_{share} and reading the flow incoming rate Φ_i from the packet headers, core routers can use (2.10) to compute both the perflow drop rate and the flow's outgoing rate. The flow outgoing rate is inserted into the packets which are then forwarded to routers downstream. This process continues until the packet arrives to CSFQ egress routers which eventually remove the CSFQ information from packet headers for compatibility to existing architectures.

CSFQ is probably the first attempt to eliminate or get rid of perflow scheduling states in core routers by inserting the states into packet headers. A similar idea is

adopted in [54] for developing the core-stateless version of the Virtual Clock [104] scheme.

2.5. General Discussion

This thesis so far focuses on the various mechanisms proposed to provide flow fairness in the Internet. However, flow based fairness is not without its limitations and criticisms. This section is devoted to some discussion on those contentious topics and other miscellaneous issues.

The importance of flow rate fairness, e.g., max-min fairness, has been questioned by Briscoe [13] who advocates for what is called the cost-fairness [55]. Accordingly, senders should be held accountable for the congestion they cause. The metric to arbitrate cost-fairness is the congestion volume, which is the congestion times the bit rate of the user causing the congestion, summed over time. An example is provided using two users sending at rates 200kbps and 300kbps into a 450kbps line for 0.5s. Congestion in this case is (200+300-450)/(200+300)=10%, so the congestion volume each causes is $200k \times 10\% \times 0.5$=10kb and 15kb respectively. As it turns out, the cost—hence the 'blame for congestion attributed to the sender'— depends on not only the *flow rate* but also on the *congestion*. For realizing cost-fairness, protocols may need to be developed at higher levels to integrate the congestion costs across different flows over time.

Other views on the topic of fairness is the following. Floyd and Allman [34] argue that flow rate fairness is practically useful, stating their case with the current Internet where most TCP congestion control algorithms enable best-effort connections to achieve rough flow rate fairness. Other researchers argue that the difference between the max-min and cost-fairness is hardly significant [60] realistically, or is a trivial issue of how to share out the excess capacity [79].

Regardless of the differing views, there are several limitations and/or open issues regarding flow fairness; see [79, 34] for details. These include the absence of fairness enforcement mechanisms, the definition of flow granularity, the relationship between flow bandwidth and RTT, fairness measured in packets per second or bits per second, and the precision of fairness (what benchmark is the fairness goal?). Probably the most important limitations are the first two. Since the bulk of the thesis discusses the enforcement mechanisms, we only and briefly consider the second open issue here: the flow granularity.

A flow can be defined by any combination of the elements among the 5-tuple. Recall from Chapter 1 that a flow is mainly defined in this thesis based on all elements of the five-tuple. If fairness is defined per connection, a flow can cheat by splitting its identity into multiple little flows. Single-user web clients and peer-to-peer traffic are allowed to create multiple connections between two end users [34].

For example, a single HTTP 1.1 user client can maintain two persistent connections with any server, while Windows XP Pro allows up to ten simultaneous peer-to-peer connections. A flow can also be defined based on source-destination IP addresses. Still, a single IP address may represent multiple end entities residing behind a NAT. On the other hand, a single host can cheat by creating (spoofing) several source addresses. Without the precise definition of flows, it may be difficult to enforce fairness among flows.

This thesis concerns mainly with flow fairness enforcement mechanisms, using algorithms implemented at routers. This, however, should not rule out the important roles that e2e congestion control algorithms play for fair, and particularly efficient, utilization of resources. To illustrate this point, we discuss two situations: (1) the avoidance of congestion collapse from undelivered packets [35], and (2) improving the service quality of voice and video connections on best-effort Internet [39]. For (1), imagine flows, some of which lack congestion control, traversing multiple congested links. Even if we adopt perflow fair queueing in the network, the flows may still squander significant portions of scarce bandwidth because most packets are carried through the network only to be dropped eventually on downstream congested links. This problem could be avoided through the use of e2e congestion control which enables the flows to adapt their rates to the bandwidth available end-to-end. For (2), imagine a highly congested bottleneck link adopting fair queueing mechanisms and traversed by voice flows running over best effort (say, UDP) connections. With excessive congestion, the voice flows suffer drops. Lacking e2e congestion control, most of the flows react by adding more FEC level which unfortunately exacerbates the underlying congestion. The voice flows could avoid excessive packet drops and make more efficient use of resources if they employ end-to-end rate control and codecs that adapt to the available bandwidth.

3. Single-queue Approximation of Perflow Fair Queueing

This chapter presents a simplified design and implementation of perflow fair queueing mechanisms. The objective is to avoid the complex buffer partitioning associated with perflow fair queueing while retaining approximate, if not total, flow fairness. This is accomplished by a single aggregate queue serving all flows. For prototyping, we use the well-known Start-time Fair Queueing (SFQ), hence we coin the term 'Single-queue SFQ' or S-SFQ to the specific implementation. The aggregate queue orders packets based on their timestamps rather than order of arrivals. Our simulations show that S-SFQ offers significantly better flow fairness than other default single-queue schemes such as RED and FIFO while retaining higher link utilization. We also discuss the adverse effect of packet loss synchronization problem common in such aggregate queues. The fairness qualities of aggregate queue based router schemes may easily and single-handedly be taken away by this problem. Loss synchronization is caused by timing effects and may surface during overloaded (rather than early) drops when the buffer becomes full (e.g., FIFO) or certain (upper) buffer thresholds are exceeded (e.g., RED).

The remainder of the chapter is structured as follows. Sec. 3.1 presents the motivation for this work and some background, followed by the design of S-SFQ in Sec. 3.2. The loss synchronization problem and the associated buffer and link usage discrimination against certain flows is initially discussed at the beginning Sec. 3.3. A further discussion of the problem with emphasis on RED follows in Sec. 3.4. The rest of Sec. 3.3 deals with simulation experiments and description of the results. We present some general comments in Sec. 3.5 before presenting our conclusions and directions for future work in Sec. 3.6.

3.1. Introduction

Apart from the e2e congestion control schemes at the end hosts, two major types of router mechanisms have been proposed to achieve congestion control, and flow protection in the Internet, see Sec. 2.4. Probably the most common ones are *perflow fair queueing algorithms* (e.g., [44, 20, 86]), defined in the framework of Integrated Services, which maintain a separate FIFO queue for each flow and, at each transmission epoch, decide the next packet to send from among the backlogged queues

(flows). Since flows are isolated into their own queues, a flow can be protected from misbehaving ones and fairness can be achieved. *Perflow dropping* or *buffer management* mechanisms have a simpler design with a single FIFO queue and determine the network conditions and connections for packet discard in order to meet certain capacity or quality of service conditions. Such schemes may need to maintain some perflow information to ensure flow protection. Examples of such schemes are variants of Random Early Detection (RED), e.g., Fair RED (FRED [64]).

3.1.1. Motivation

We intend to replicate the fairness of perflow queueing mechanisms using the simple queue design of buffer management mechanisms. In particular, we study the performance of Start-time Fair Queueing (SFQ) [44] using a single global queue for all flows, rather than the norm of assigning a FIFO queue for every flow serviced by the router. Assigning a queue for each incoming flow may practically become too cumbersome to be deployable in the Internet when the number of flows can be very high. We call such a modification to SFQ as *single-queue SFQ* or S-SFQ for short. S-SFQ keeps some perflow information, but does not hold detailed statistics of how many perflow packets have been dropped or are currently found in the queue. We have chosen SFQ because of its proven qualities routinely required in integrated services network (data, video, audio): low implementation complexity compared to the well-known Weighted Fair Queueing (WFQ), better flow isolation compared with a competing fair queueing algorithm called Self-Clocked Fair Queueing [41], ability to provide fairness under variable server capacity (e.g., in wireless networks) and ability to provide low average and maximum delay. An SFQ variant has also found use in real networks [61].

3.1.2. Contribution

The contribution of this chapter is twofold. First, we provide a design and implementation of perflow fair queueing mechanisms in general, and SFQ in particular, based on a single shared queue, rather than a rack of queues. This in turn calls for a non-FIFO queue that can sort packets based on their encoded timestamps. This is because all flows share the queue and the policy for flow scheduling should not merely base on the order of packet arrivals. Upon arrival, packets are assigned start tags computed by the underlying SFQ algorithm. Additionally, S-SFQ's potential in terms of fairness and link utilization is investigated through simulation, and compared against the other well-known single-queue-based router schemes such as FIFO (Drop-Tail) and RED. The second important contribution is that we identify and discuss a recurring problem in the context of this work called *loss synchronization* that has a potential to obliterate the desirable qualities of the underlying scheme. We find that this problem commonly arises in shared queues with no specialized (randomized) buffer management.

3.2. Single-queue SFQ

Recall from Sec. 2.4.1 that a packet's service tag under SFQ equals its start timestamp $S(\cdot)$. The timestamp computation is given by (2.7) [44]. Note that the start tag computation depends on the finish timestamp—computed by (2.8)— of the previous packet of the same flow which may already have left the server.[1] Arriving packets are first assigned service tag before being transmitted in ascending order of their tags.

The objective of S-SFQ is to approximate the per-flow SFQ, keep its desirable properties while getting rid of the per-flow queue requirements, or the complexity of buffer partitioning, by employing a single global queue. We implemented S-SFQ in ns-2 [29] and the pseudocode of the implementation is shown in Fig. 3.1. The flow information we intend to keep or save under S-SFQ can be seen at line 7 of the Enqueueing Algorithm. The $v(t)$ (line 4, Dequeueing) is the server virtual time at time t, and this value is used in start tag computation (line 5, Enqueueing). $v(t)$ is reset when the queue becomes idle or empty (line 7, Dequeueing). As discussed before, the queue **Q** is not FIFO since packets are served (i.e., queued and transmitted) in increasing order of their start tags rather than order of their arrivals. A packet service tag is computed on arrival using Eq. 2.7. The packet is then placed in the right slot in **Q** based on this value. When the buffer becomes full, we pick the packet at the tail to drop since it has the largest start tag. Unlike FIFO/Drop-Tail, this packet may not be the packet that arrives last. The dequeueing procedure is very simple since S-SFQ draws a packet from **Q** head for transmission.

[1] A flow entry is removed after a certain period of inactivity (timeout).

Enqueueing Algorithm	Dequeueing Algorithm

1: Upon receiving p_f^j at t
2: if (p_f^j first of f) then
3: add f to flow list $\mathcal{F}$;
4: end if
5: compute $S(p_f^j)$; // (2.7)
6: encode $S(p_f^j)$ into p_f^j;
7: compute $F(p_f^j)$ and save; // (2.8)
8: enque p_f^j into **Q** // *non-FIFO*
9: if **Q**-size $\geq$ **Q**-limit then
10: draw packet p from **Q** tail;
11: drop p;
12: end if
	1: draw p from **Q** head
	2: if (p exists) then
	3: extract S(p) from p header
	4: $v(t) \Longleftarrow S(p)$;
	5: else
	6: // **Q** empty; resetting params
	7: $v(t) \Longleftarrow 0.0$
	8: $\forall f \in \mathcal{F}$ reset f finish tag to 0.0
	9: end if
	10: return p

Figure 3.1.: Enqueueing and Dequeueing in S-SFQ

3.3. Performance Evaluation

This section is devoted to evaluation of S-SFQ using extensive simulation. To give some context we compare and contrast its performance to the status quo having global queues (FIFO, and RED). Unless otherwise stated, the default RED parameter values are used, e.g., the initial $max_p = 0.1$, $min_{th} = 5$ and $max_{th} = 15$. Since the setting of optimal RED parameters is more empirical than exact, rather than hardcode the parameters, we enable both `gentle` and `adaptive` parameters, see Sec. 2.4.2 for detail. These parameters allow RED to 'auto-tune' to improve network performance. The main performance metrics of interest in this chapter are fairness (simple and weighted), flow throughput and link utilization.

Every simulation is replicated 30 times and the 90%-level confidence intervals are computed. The confidence intervals are very small in the majority of cases and therefore, unless stated otherwise, they are not reported here. The flow start times are uniformly distributed, mostly on [0,1].

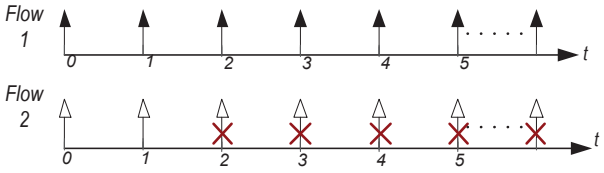
3.3.1. Buffer Usage Discrimination and Loss Synchronization

Early at this stage, we would like to emphasize probably the most important observation in this work—the problem of loss synchronization caused by buffer usage discrimination against some flows. The impact of the problem is so severe that it can totally impair the fairness of the underlying mechanism. However, we discover that this problem is not unique to S-SFQ. In Sec. 3.4, we point out scenarios where the problem manifests in RED during *forced drop*.

For clarity of exposition, we use an oversimplified topology shown in Fig. 3.2(a). Two CBR sources generate packets at the same constant rates, say *1 pkt/s*. It may be possible to get such packet trains in sub-RTT time-scales when a window of TCP packets are transmitted [50]. The guaranteed rates of both flows (r_f in Eq. 2.8) are also *1 pkt/s*. The router has a buffer size of 2 packets and link capacity C of *1 pkt/s* and link delay 1ms. Both flows start transmission at time 0. We expect the router to be congested and the two flows to share the link approximately equally. Surprisingly, we found that *all* but 2 packets from flow 2 are dropped (Fig. 3.2(b)). All flow 1 packets are transmitted.

Let us find out the reason why. At time 0, packets p_1^1 and p_2^1 are at the link in that order. Both have start tags of 0. p_1^1 is transmitted immediately and p_2^1 queued. Their finish tags $F(p_1^1) = S(p_f^j) + l_f^j/r_f$ are $0 + 1/1 = 1$. At $t = 1$, packets p_1^2 and p_2^2 arrive, both of which observe a server virtual time of 0. Note that the virtual time in SFQ is the start tag of the packet being dequeued. When p_1^2 and p_2^2 arrive, p_1^1 is being transmitted. Since the packet start tag is computed as the maximum between the virtual time on arrival and the finish time of previous packet, then both $S(p_1^2)$ and $S(p_2^2)$ are 1. And the finish times $F(p_1^2)$ and $F(p_2^2)$ become 2. In

(a) S-SFQ bottleneck, C=2 pkt/sec, B=2 pkt, 1ms link delay.



(b) Packets transmitted and dropped (marked **X**).

Figure 3.2.: Start tag of packet p_f^k is $S(p_f^k) = k - 1$, and $F(p_f^k) = k$ where $k \geq 1$. All packets of flow *1* but only the first two packets of flow 2 are transmitted as shown.

short, for any packet p_f^k of both flows, we find that $S(p_f^k) = k - 1$ and $F(p_f^k) = k$. By $t = 2$, packets p_1^1 and p_2^1 have already left, p_1^2 is being transmitted, p_2^2 holds one slot in the queue, and packets p_1^3 and p_2^3 just arrive. One of the last two packets should be dropped as the buffer space is exhausted. Since both have service tags of 2, the packet that arrives slightly later—which is p_2^3—is dropped. This process continues. All but the first two packets of the second flow are dropped and we verify by simulation this is indeed the case. All packets belonging to the first flow are transmitted. A repeat experiment with buffer size of N does not fundamentally solve the problem. In that case, all packets from flow 1 and only N packets from flow 2 are transmitted.

The problem lies on synchronization and timing of drops. Regardless of its size, the buffer always becomes full exactly when a certain flow 2 packet arrives. The result is that flow 2 gets discriminated from getting its fair share of buffer space and link bandwidth, resulting in a total loss of flow fairness in S-SFQ. This synchronized drop problem is similar to the *lockout* exhibited by the Drop-Tail routers where few flows monopolize the link. The above experiment is repeated using FIFO/Drop-Tail queue.[2] All but the first packet of flow 2 are dropped and all packets of flow 1 are transmitted.

[2]FIFO is the default queueing algorithm in routers. Therefore, in the rest of this thesis, when we say a Drop-Tail / RED / CHOKe / gCHOKe queue, the FIFO queueing discipline is implicit. Similarly, when we say a 'FIFO' queue without stating the packet drop policy, Drop-Tail is implicit. That means, unless stated otherwise, FIFO, Drop-Tail and FIFO/Drop-Tail are used inter-changeably in this thesis.

Sec. 3.4 continues the discussion on this problem in some detail with special emphasis on RED.

3.3.2. Traffic with Contrasting RTTs and Packet Sizes

The topology is shown in Fig. 3.3. All links are 10Mbps, and link delays are as shown. Both sources S1 and S2 adopt TCP New Reno and transmit packet sizes of 1000 and 512 bytes respectively. The maximum window sizes are 250 segments for both. The bottleneck is the link connecting the router to the sink. All buffer sizes are 50kB. The router applies different scheduling or buffer management schemes.

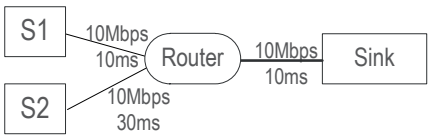


Figure 3.3.: TCP sources of different packet lengths and round-trip times.

Under the status quo (that is, FIFO and RED), we expect flow 2 to have less throughput and link utilization since it has smaller packet sizes and longer RTTs. Note the maximum advertised window for both is equal in number of segments, but the segment sizes in bytes vary almost by a factor of two. Recall that average TCP sending rate is proportional to W/RTT (see (2.3) and (2.4)), where W is the average congestion window. It is therefore trivial to understand that flows with longer RTTs and smaller segment sizes obtain poorer throughput. The results are tabulated in Table 3.1.

Table 3.1.: Flow link utilization under three router schemes shown in Fig. 3.3.

Metric	Flows	FIFO	RED	S-SFQ
Perflow link utilization [%]	Flow 1	75	64	59
	Flow 2	22	30	38
	Total	97	94	97

Even though we are far from achieving total fairness, Table 3.1 clearly shows the performance improvement in total link utilization and particularly in flow fairness of the S-SFQ over FIFO and RED. We do not see any total dominance of any particular flow as in FIFO where flow 1 link utilization is 3.5 times that of flow 2. And RED does not render fairness at all: flow 1 still grabs more than 65% of the used capacity. We discuss this *RTT unfairness* in Sec. 2.2.2. In addition, link utilization in RED is poorer than both FIFO and our mechanism. In a bid for lower queueing delays, RED commits less than the full buffer capacity.

Is there a way to improve the flow fairness of S-SFQ shown in Table 3.1? We vary the bottleneck buffer capacity from *25kB* to *200kB*. We adopt a strategy of twice the delay bandwidth product (*of flow 2*) as a buffer limit for our simulation, i.e., $2 \times RTT \times BW = 2 \times 80ms \times 10Mbps = 200kB$. For each combination of buffer capacities and router schemes, we run a batch of 30 replications and the results are reported below.

Table 3.2 shows the bottleneck link utilization as buffer capacity is varied. While larger buffer sizes often translate into better link utilizations under both FIFO and S-SFQ, this is not generally the case in Adaptive RED as can be seen from Table 3.2. As explained in Sec. 2.4.2, RED is designed to lower average queueing delay by controlling the average queue size avg. Rather than committing the full buffer capacity, avg is maintained somewhere between two queue thresholds. Recall from Sec. 2.4.2 that avg is allowed to go as high as $2 \times \max_{th}$ when `gentle` is set and as high as $\max_{th}$ otherwise. Since an incoming packet may be dropped before the buffer becomes full, the bottleneck may not be fully utilized. As a consequence, RED has poorer link utilization than both FIFO and S-SFQ. The performance of RED may be improved by fine-tuning[3] of parameters when we know the traffic characteristics of the network. This may not be possible in real networks. Apart from lack of exact principles on how to set the RED parameters, some earlier works argue that modifying these parameters may have no significant improvements in its performance [17, 69] or that the "best" parameters chosen for one metric may result in poorer performance in another metric [17].

Fig. 3.4 shows the ratios of average link utilizations between the two TCP flows of Fig. 3.3 and their error bars. The per-flow link utilizations can easily be computed by consulting the ratios from the figure and the total link utilizations from Table 3.2. With an increasing buffer size, the sorting becomes excessive, but that allows S-SFQ to significantly improve its fairness performance. The fairness can approach the ideal value of *1* with large buffer sizes. For Adaptive RED, there is no real gain in fairness with increasing buffer sizes. Just like in Table 3.1, the link share of flow

[3]For example, in this case, we can set $\min_{th}$ and $\max_{th}$ to vary with the available buffer sizes.

Table 3.2.: Effect of buffer capacity on link utilization under different router schemes.

Buffer Size	Link Utilization [%]		
	FIFO	RED	S-SFQ
25kB	91	94	90
50kB	97	94	97
75kB	98	94	98
100kB	98	94	99
125kB	98	94	98
175kB	98	94	98
200kB	98	94	99

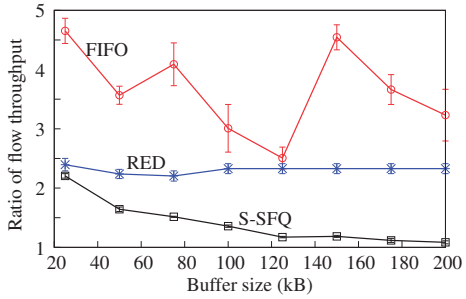


Figure 3.4.: TCP throughput ratio.

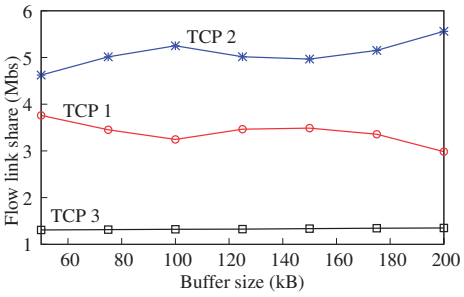


Figure 3.5.: S-SFQ's proportional fairness.

1 (short RTT flow) is more than twice than that of flow 2. Even worse than RED is FIFO which unfairly allocates to flow 2 an arbitrary, yet small, fraction of the link share used by flow 1.

3.3.3. Proportional Fairness

In Sec. 3.3.2, the SFQ weights or rates (r_f in Eq. 2.8) given to all flows are equal; hence flow fairness is basically an approximate equality between the flows' throughput. S-SFQ can provide approximate *weighted fair share allocation* [57]. This is possible by manipulating the different weights allocated to the flows. We use the same topology and flows of Figure 3.3. An additional TCP flow (flow 3) is introduced with the same RTT and packet size of flow 2. However, flow 3's receiver (or advertised) window *rwnd* is limited to 25 segments. Flow 2 which is the longer RTT and 512 byte-sized flow is allocated twice the weight of the other flows. Fig. 3.5 shows how the link is shared among the three flows.

The equivalent integral *weights (or rates)* are 1, 2, 1 respectively for flows 1, 2 and 3. In an ideal fair scheme, the flows attain link shares in Mbps of 2.5, 5, and 2.5, respectively. Since flow 3's window is capped at 25, its maximum throughput of $W/RTT = 25 \times 552^{\dagger} \times 8/(2 \times 0.04) \approx 1.4$Mbps is less than its allocated share. The left over bandwidth $10 - 1.4 = 8.6Mbps$ must be split between flows 1 and 2 in proportion to their weights. The end result is ideally: 2.9Mbps for flow 1, 5.7Mbps for flow 2, and 1.4Mbps for flow 3. The ideal fair ratio of link utilizations between flow 1 and flow 2 is $\frac{1}{2}$. Fig. 3.5 demonstrates that S-SFQ's approximate fairness improves with increasing buffer sizes. While flow 2 may claim a larger share via its weight, it is inherently at disadvantage due to its longer RTT and smaller packets. Applying the average TCP throughput formula $W \times PktSize/RTT$ crudely to the flows, flow 2 would receive throughput approximately 25% of that of flow 1 under normal situations (FIFO queues). This observation can also be verified from the

[†]TCP segment size is actually 512B. But since we measure utilization on the link, we add TCP/IP header of 40B.

FIFO graph of Fig. 3.4. Under such throughput prohibitive characteristics of flow 2, any gross approximation to the ideal $\frac{1}{2}$ would be acceptable. By contrast, both Drop-Tail and RED lack mechanisms for providing such differentiated service.

3.3.4. Impact of Unresponsive Flows

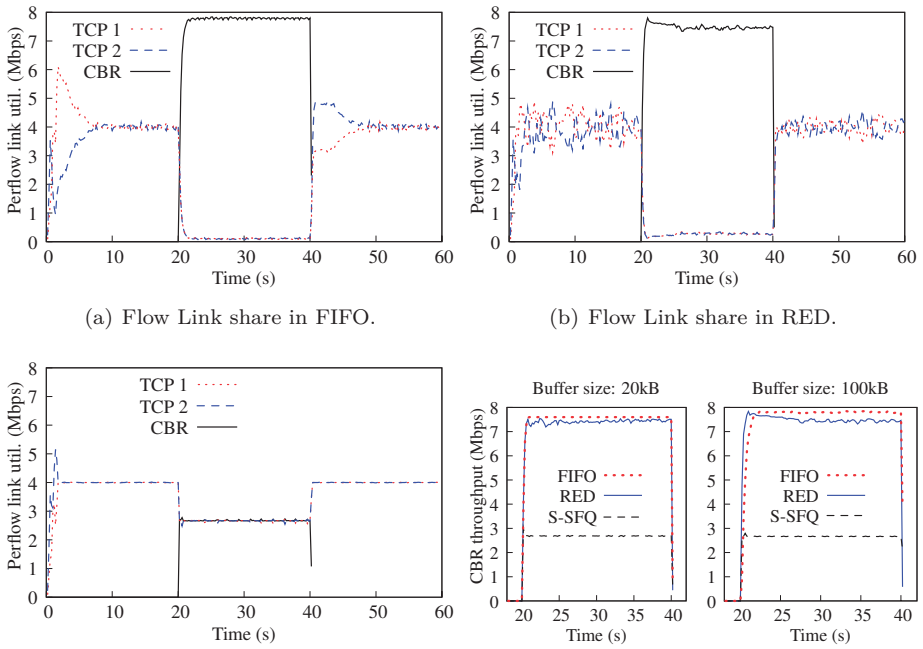
It is interesting to see whether S-SFQ fares well in the presence of unresponsive flows. We use a dumbbell topology with three sources, 1 CBR and 2 TCP flows, competing over an 8Mbps bottleneck in a manner similar to Fig. 3.3. The CBR source generates at the rate of the full bottleneck capacity, i.e 8Mpbs, and is started and stopped at 20s and 40s of simulation, respectively. The two TCP sources have equal RTTs, equal advertised windows and start uniformly on [0,1]. We steadily vary the buffer capacity at the bottleneck from 20kB to 100kB. The comparison between the three router schemes are discussed below.

Figures 3.6(a), 3.6(b) and 3.6(c) report link utilizations of the flows in each scheme with a 60kB buffer size. Results using other buffer sizes are similar. Only S-SFQ provides nearly perfect fairness between the TCP flows with no difference of note in their link shares at any point of our experiments. For instance, both TCP flows receive 4Mbps in the absence of the CBR. Remarkably, when the unresponsive flow is injected at $t = 20$s at full link rate, S-SFQ reacts quickly to ensure a new fair rate to both TCP flows. Both RED and FIFO in particular relinquish the link almost exclusively to the CBR flow, close to shutting down the TCP flows. Fig 3.6(d) show that this link monopoly by CBR in both FIFO and RED does not abate with increasing buffer size.

3.4. Loss Synchronization

This section continues the discussion on the most important issue that affects all the router mechanisms we considered in this chapter—loss synchronization. It is important to distinguish this problem from *global synchronization* typically exhibited by Drop-Tail queues. Global synchronization is a throughput-impairing condition that occurs when all flows reduce their windows at the same time upon receiving simultaneous congestion notifications or packet drops, see also Sec. 2.4.2. One of the side goals of RED is to avoid this problem of Drop-Tail. We do not discuss global synchronization further in this chapter.

As discussed in Sec. 3.3, synchronization of loss introduces peculiar fairness problems in S-SFQ and Drop-Tail. The discussion in this section is a follow-up with our focus shifted to RED.



(a) Flow Link share in FIFO.

(b) Flow Link share in RED.

(c) Almost perfect fairness.

(d) S-SFQ restricts unresponsive flows better.

Figure 3.6.: Effect of unresponsive flow sending at full rate of bottleneck. S-SFQ performs better than the FIFO and RED.

The main argument of this section is as follows. The loss synchronization problem of S-SFQ discussed in Sec. 3.3.1 is not unique to S-SFQ or Drop-Tail. We also noted this problem even in RED during forced drops. A total impairment of fairness in RED is observed during persistent congestion caused by unresponsive flows. The topology of Fig. 3.2(a) is used where the bottleneck is a 1Mbps, 1ms Adaptive RED link with a buffer capacity of 100 packets. Both CBR flows generate traffic at the rate of the bottleneck capacity link (1Mbps). The two CBR sources start at time 0 and the simulation runs for 200 seconds. From the resulting simulation trace we found the following statistics: both flows generate 25000 packets; 24925 transmitted, 46 dropped from flow 1; but only 75 packets are transmitted from flow 2 and the remaining 24925 packets are all dropped. We set up another experiment this time with $\min_{th}$ and $\max_{th}$ configured at 25% and 75% of full buffer capacity, respectively. The problem escalates: all except 2 packets of flow 1, but only 102 packets of flow 2 are successfully transmitted. The main reason is that RED reduces to Drop-Tail and its fairness suffers from loss synchronization problem, see Fig. 3.7.

Since `gentle` parameter is set, see Fig. 3.7, the drop probability linearly and rapidly (due to persistent congestion) increases from 0.1 (default max_p) to the

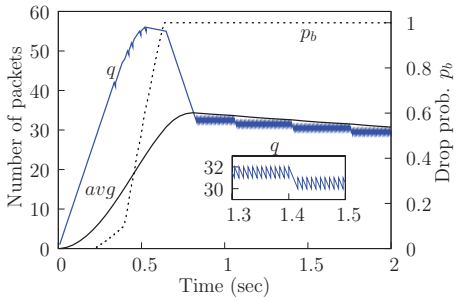
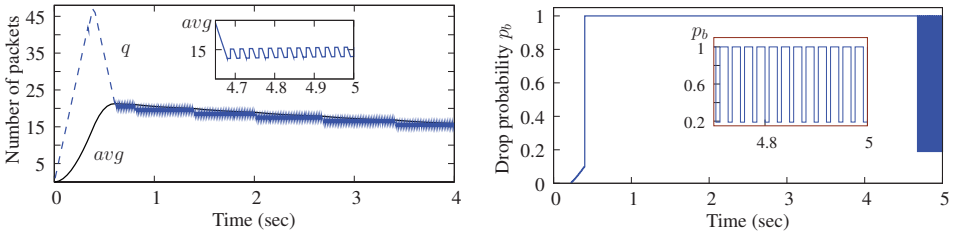


Figure 3.7.: Persistent congestion in RED reduces to loss synchronization, as in Drop-Tail routers. p_b increases rapidly, but does not abruptly jump because `gentle` is set.

maximum of 1.0 as avg increases from $max_{th} = 15$ (default) to $2 \times max_{th} = 30$. In the face of heavy congestion, $2 \times max_{th}$ is the upper limit for avg in steady state. In the steady state and under heavy congestion, q stabilizes around avg. This is done in two steps. Firstly, since the queue receives two packets but can transmit only one at each epoch, it enqueues the first packet that arrives but drops the second one. The first packet comes from flow 1 and the second from flow 2. Hence all flow 2 packets are dropped in the steady state and the dropping is synchronized to that particular flow. This behavior is identical to lockout of Drop-Tail on full buffer. Secondly, in the long run, receiving 2 packets but transmitting one and dropping another only keeps q at a constant level. Since p_b was small initially, the queue has already been filled with many packets. To reduce the queue size q, say from 55 to the required value of $\approx 2 \times max_{th} = 30$, we must drop additional packets. This explains why some flow 1 packets are occasionally dropped, causing regular decreases of q by 2 packets—one from each flow (see the snapshot in Fig. 3.7). When $q \approx 28$ at $t = 2.12s$, only (in fact all) flow 2 packets are dropped thereafter.



(a) avg and instantaneous queue q.

(b) Flapping of drop prob. p_b.

Figure 3.8.: Synchronization of flow 2 loss when `gentle` is not set. avg repeatedly crosses up and down max_{th} (15 by default) (a), which causes p to frequently flap (b). $p \approx 1$ results in forced drop which may lock RED into loss synchronization.

If RED's `gentle` were not enabled, however, then we would expect the p to abruptly jump to 1.0 when $avg = max_{th} = 15$ and kick start deterministic dropping and loss synchronization at that point (see Fig. 3.8). Due to synchronization caused by forced drop (i.e., when $p = 1.0$), the number of dropped packets from flow 2 is one order of magnitude larger than flow 1; that is, 33 packets from flow 1 and 535 packets from flow 2 are dropped during the interval $0.4 \leq t \leq 4.68$ where $p = 1$.

In general, during loss synchronization, arbitrary flows suffer in resource utilization. The identities of flows that suffer seem to depend on the timing of packet arrivals.

3.5. Discussion and Related Work

In the loss synchronization problem of Sec. 3.3.1, we could have introduced per flow information to aid the discard policy of S-SFQ when packets of equal timestamps (start tags) are received on full buffer. For instance, we can define new per flow variables to hold the number of packets of a flow so far buffered or dropped and such information can be used to restore fairness. Implicitly or explicitly, this indeed is the case in several of the fair per-flow architectures such as WFQ [81], FRED [64], Cross-Protect [61]. For example, FRED uses per flow accounting to improve the fairness of RED. For comparison, we repeat the UDP experiment of Sec 3.4 using FRED instead of RED. FRED results in very high fairness between the two UDP flows. Both UDP flows share the link equally and experience equal number of drops.

We believe that if a network operator insists on a per-flow scheduling, it may be useful to leverage the flow-level information in the discard policies as well. Similar argument is presented in [94]. In addition, it is possible to deploy such flow level architectures at the edge of networks where the number of flows and capacity of residential routers are both limited. However, deployment of stateful per-flow schemes at core routers may not be scalable and indeed counters the spirit of many (potentially stateless) single global-queue proposals. In this chapter, we only keep limited flow states in order to approximate the per flow SFQ. The well-known *global* queue management algorithms—Drop-Tail and Random Early Detection (RED)—serve all flows using single queues without keeping per flow state information. While both are based on simple-queue design with FIFO scheduling, they cannot replicate the fairness of perflow scheduling schemes. Both are vulnerable to the loss synchronization problem as stated earlier.

In the original RED paper [38], the following conclusion is drawn:

> *The probability of marking a packet from a particular connection is roughly proportional to that connection's current share of bandwidth through the RED gateway.*

This observation generally holds in the region of early drops. Since this region is insignificant under heavy congestion (Figs. 3.7 and 3.8), the assumption may not always reflect a typical RED operation. During forced drop, loss synchronization may prevail and invalidate the assumption. As a result, a flow with a lion share of buffer space and link usage may still experience the fewest drops, and increase its buffer share and dominate the link.

3.6. Conclusion

In this chapter, we first present an intuitive SFQ design using a single queue. The queue orders packets in order of their time-stamps rather than arrivals. Though we have used unrealistic network topologies, the simulation experiments raise severe fairness issues under de facto single-queue schemes currently available, i.e., RED and FIFO/Drop-Tail. S-SFQ can provide high fairness among flows, regardless of the flow traffic type, packet sizes or round-trip times. Especially under larger buffer sizes, S-SFQ's fairness can closely approximate the intricate fairness afforded by perflow SFQ. S-SFQ's advantage over the perflow equivalents is its relatively simple single-queue design. S-SFQ still keeps some perflow information in order to approximate the fairness of SFQ. The flow information is removed when there are no more packets of the flow in the queue, and when the flow timer expires.

The second important contribution of this chapter is the study on flow buffer discrimination caused by synchronized packet loss. As it turns out, S-SFQ fairness performance can be heavily impaired. We observe that this is a common problem in all schemes we study in this chapter. The problem may manifest itself during full buffers in FIFO and S-SFQ and forced drops in RED. It is less prevalent in closed-loop traffic situations. The problem has a severe debilitating effect on the qualities of the underlying scheme. We believe the deterministic nature (e.g., $p_b \approx 1$) of packet drops is to blame. We propose a solution in Chapter 6.

4. Generalizing the CHOKe Flow Protection

There exists a natural tussle between simplicity of router algorithms, and the quality of service they provide. On the one hand, flow-oblivious buffer management mechanisms such as Drop-Tail and RED are simple, completely stateless and highly scalable. However, they allow traffic flows to increase the shares of bandwidth and buffer space merely by increasing their sending rates. On the other hand, perflow fair queueing algorithms are characterized by strong perflow service semantic and flow-level fairness achieved by complex buffer partitioning, and maintenance of perflow states. Since achieving the best of the two worlds has proved elusive, the two challenges that may naturally follow are (Sec. 1.3): *(1) to "approximate" the service quality of perflow fair queueing using the simple aggregate queue design of buffer management schemes, or (2) to compromise the strong service semantic of perflow fair queueing using the framework of simple stateless designs?* The last chapter attempts to address the first challenge, and a more thorough development of the scheme will follow in Chapter 6. In this chapter, we present one way to address the second challenge. To that end, we propose a suite of very simple and completely stateless active queue management mechanisms, collectively called geometric CHOKe or *gCHOKe*. The price paid for the lightweight and scalable design is powerful flow protection in lieu of full-fledged flow fairness.

4.1. Introduction

4.1.1. Background

The perflow fair queueing schemes and the flow-myopic router algorithms (Drop-Tail and RED) stand at the opposite ends of both the fairness and complexity spectra. The simple stateless RED, for example, allows flows to grab network resources at their traffic injection rate. Perflow queueing schemes, on the other hand, are complex and stateful but provide max-min fairness. In order to produce the synergy of the two desirable qualities (simplicity vs. quality flow fairness), some sacrifices may be necessary. In this chapter, we trade flow protection, which is a softer[1] requirement than flow fairness, for simplicity and statelessness. To this

[1] Fairness automatically provides flow protection, but the converse is not true [57].

end, we propose a suite of simple and stateless active queue management (AQM) schemes, collectively called geometric CHOKe (gCHOKe), to protect responsive flows from unresponsive ones. Rather than requiring max-min flow fairness as in complex perflow queueing mechanisms, the interest here is *how much traffic control the protection mechanism wields over the use of bandwidth and buffer space by the unresponsive or malicious flow(s).*

Our proposed gCHOKe scheme has its root in and is a generalization of the CHOKe scheme, which in turn can be extended from the RED queue. For clarity of presentation, we repeat the observation that leads to the design of CHOKe (see Sec. 2.4.2.2). Remember that packet drops and admissions at a congested RED queue are randomized. Due to this, both the packet admission and drop histories at the RED queue form an unbiased statistics about the state of affairs regarding the rate of active flows in the queue. Specifically, a flow is more likely to have packet drops or admission tallies in proportion to its arrival rate. CHOKe uses the recent packet admissions—packets queued in the RED buffer—to penalize the high bandwidth flows. It reads:

"*When a packet arrives at a congested router, CHOKe draws a packet at random from the FIFO buffer and compares it with the arriving packet. If they both belong to the same flow, then they are both dropped; else the randomly chosen packet is left intact and the arriving packet is admitted into the buffer with a probability (based on RED) that depends on the level of congestion.*"

It turns out that CHOKe keeps no explicit flow information and can be designed with simple tweakings of the RED algorithm.

Apart from its simple and stateless implementation, another desirable property of CHOKe as a flow protection mechanism is that it ensures bounded bandwidth and buffer shares [98, 96, 77]. Specifically, an unresponsive flow in CHOKe cannot exceed a certain bandwidth share or buffer limit in the presence of many responsive flows. The following theorems [98] provide an overview of this property, where UDP represents the extreme of unresponsive traffic:

Theorem 4.1.1 *(i) The maximum UDP bandwidth share in CHOKe is bounded by* $(e+1)^{-1} = 26.9\%$.

(ii) This is attained when UDP input rate, after congestion based dropping, is $C(2e-1)/(e+1) = 1.193C$*, where* C *is the link capacity, and*

(iii) In that case, CHOKe-based UDP dropping rate is $(e-1)/(2e-1) = 38.7\%$.

Theorem 4.1.2 *When UDP input rate increases without bound, its buffer share approaches 50% but its link utilization approaches 0.*

Router mechanisms may be required to apply such punishment strategies to unresponsive flows of extreme rates [31, 71, 35].

4.1.2. Motivation and Contribution

Given the nice properties of CHOKe described above, it is natural to ask if CHOKe can be generalized and/or if its performance can be improved without tampering with its simple and stateless design. Trying to give answers to these questions forms the motivation of this work. Particularly, we aim to generalize CHOKe, while retaining its desirable features (i.e., simplicity and statelessness), and simultaneously empower it with tighter or more powerful controls on the use of link bandwidth and buffer space.

We propose geometric CHOKe (gCHOKe), which turns out to be highly intuitive and inherits the design principle of CHOKe. The basic principle is to reward each successful flow comparison of packets with an *extra or bonus flow matching* trial. The scheme is characterized and indexed by a single configurable control parameter called `maxcomp` $\in [1, \ldots, \infty)$ which limits the maximum number of successful flow comparison attempts that can be tried per arrival. As we shall see in Sec. 4.3.3, when `maxcomp` is unlimited, the number of matching trials executed per arriving packet follows a geometric distribution, hence the name of the scheme. If `maxcomp` is limited, we obtain a truncated geometric distribution. It turns out that CHOKe is just a special case with `maxcomp` 1. The power of flow protection generally improves with `maxcomp`. When `maxcomp`>1, the per packet processing in gCHOKe can be slightly more complex than in CHOKe.[2] Still, gCHOKe is lightweight as the mean number of matching trials per arrival in the worst case is very few, see Sec. 4.7.

In this chapter, we provide an accurate analysis on the throughput and buffer occupancy that an unresponsive flow can maximally receive under gCHOKe of any `maxcomp`. The analysis, which is validated through simulation, shows that gCHOKe—compared to the plain CHOKe (which is just the simplest case of gCHOKe)—can achieve over 20% improvement in the bounds of both bandwidth and buffer space used by the aggressive flow. In addition, up to 14% of the total link capacity can be saved from the unresponsive flow, allowing responsive or rate-adaptive flows to obtain a better share of resources in the router.

4.1.3. Chapter Organization

The rest of this chapter is structured as follows. Sec. 4.2 outlines the main idea behind gCHOKe. The gCHOKe model and assumptions for the analysis are presented in Sec. 4.3. Sec. 4.4 presents theoretical analysis of UDP throughput. Model validation and simulation results are presented in Secs. 4.5 and 4.6, respectively. Sec. 4.7 engages in further discussion related to gCHOKe. Sec. 4.8 presents the related works. Finally, our conclusions are presented in Sec. 4.9.

[2] Though CHOKe is a special case, when we draw comparisons between CHOKe and gCHOKe, we mean gCHOKe with `maxcomp`>1.

4.2. Geometric CHOKe (gCHOKe)

4.2.1. The Scheme

A schematic diagram of gCHOKe is shown in Fig. 4.1. When a packet arrives to a congested queue, gCHOKe randomly samples (picks) a packet from the queue. If the incoming and the sampled packets belong to the same flow, gCHOKe continues to sample another random packet from the queue. Either of the two conditions can stop the matching process: (i) when a matching trial fails, or (ii) when a total of `maxcomp` trials are executed. All matched packets and the arriving packet are dropped. In the event of no matching at first attempt, the sampled packet is restored to the queue, but the arriving packet may still be dropped with a probability that depends on the level of congestion. If the buffer is managed by RED, which is the case in this work, this probability is determined by the RED parameter setting. Throughout this chapter, we call this congestion-based dropping probability the *ambient drop rate*, see also RED-PD in Sec. 2.4.2.2.

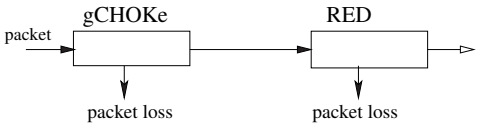


Figure 4.1.: Schematic of a gCHOKe queue.

The alert reader may have noticed that setting `maxcomp` to 1 reduces gCHOKe to CHOKe.

Recall that the main idea and design principle behind CHOKe is that a high-bandwidth unresponsive flow will likely have more packets in the buffer, hence a higher probability for flow matching and consequently dropping. Due to this, CHOKe punishes the unresponsive flow from completely dominating the use of the buffer and the link. This principle of CHOKe is inherited by gCHOKe, and so are the statelessness and simplicity. Additionally, gCHOKe rewards each successful matching with a *bonus trial*. A sequence of matching trials per arrival provide an extra level of protection to rate-adaptive flows from unresponsive ones. By choosing / tuning `maxcomp`, a desired protection level may be achieved. This introduces flexibility for traffic control, which is however lacking in the original plain CHOKe.

Note that `maxcomp` can assume any integer in the range $[1, \cdots, \infty)$. When the maximum number of trials per arrival is unlimited, i.e., `maxcomp`$\rightarrow \infty$, the matching process can stop only through the fail of a matching trial. This is an extreme case specifically studied in [27]. `maxcomp` value between 1 and infinity, exclusive, gives performance between that of plain CHOKe and this extreme gCHOKe.

4.2.2. Example Scenario

We compare gCHOKe with unlimited `maxcomp` against CHOKe and RED using simulation of the network setup shown in Fig. 4.3 where there are $N = 32$ TCP flows and the link capacity is $1Mbps$. The result (Fig. 4.2) shows that RED has no fairness mechanism in place. The UDP flow controls over 90% of the link capacity and this starves out the TCP flows. Under CHOKe, UDP is restricted to 26% of the link capacity. Using the enhanced flow protection afforded by gCHOKe, UDP throughput is restricted further down to 19%—a saving of a modest 7% of the link capacity, or an overall improvement of 27% over CHOKe. The savings on link capacity can be much higher with higher rate UDP flows (see Fig. 4.10(a)).

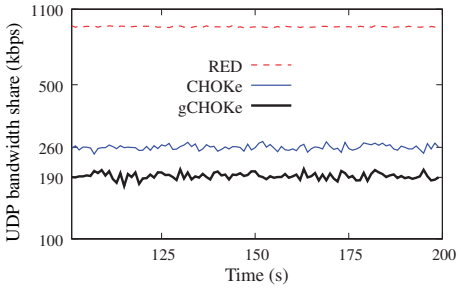


Figure 4.2.: gCHOKe can restrict high bandwidth flows better than CHOKe and RED.

4.3. The Model

This section lays out the model and theoretical foundation for the study of steady state performance of gCHOKe.

4.3.1. The System and Assumptions

The main studied system is shown in Fig. 4.3.[3] The link capacity is C (packets/sec). We study behavior in the steady state which we assume exists. The backlog size in steady state is assumed to stabilize at b.[4] There is a single unresponsive / aggressive UDP flow and N rate-adaptive similar TCP flows. Note that similar model and assumptions have been used for CHOKe analysis [98] [96].

For analytic simplicity, we consider a gCHOKe queue where the ambient and gCHOKe based droppings are reversed, see Fig. 4.4. The terms h_0, r and $\mathrm{p}_{gCHOKe}(m)$

[3]See Secs. 4.4.3 and 4.7.2 for scenarios involving multiple UDP flows and multiple congested links.

[4]Note b depends on the traffic load and the `maxcomp`, see Sec. 4.6.2.

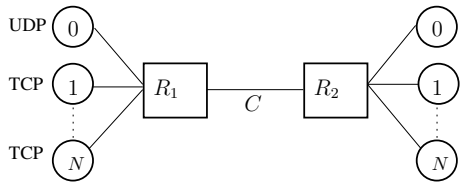


Figure 4.3.: System model.

will become clear in later sections. The idea of reversing the order of RED and CHOKe parts was initially suggested in [98] and has also been adopted in [96]. We will illustrate in Sec. 4.5.1 that the errors due to this reversal are negligible.

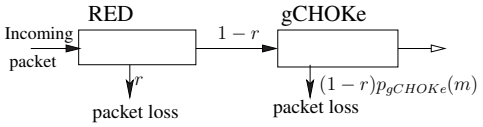


Figure 4.4.: Schematic of the gCHOKe analytic model.

4.3.2. Notations

Notations are shown in Table 4.1. Flows are indexed by i, where $i \in (0, N)$. Index 0 denotes the UDP flow, and indices $1 \ldots N$ denote TCP flows. gCHOKe(m) denotes a gCHOKe scheme with `maxcomp` value set to m. Loss rate, loss probability, dropping rate, and dropping probability are used interchangeably in this chapter.

Table 4.1.: Notation.

Parameters	Semantics
m	`maxcomp` of the gCHOKe under study
x_i	source rate of flow i
μ_i	utilization of flow i
b_i	number of flow i packets in buffer
b	total backlog $b = \sum_{i=0}^{N} b_i$ in packets
p_i	total flow i drop or loss rate (both RED and gCHOKe) caused by the arrival of a flow i packet to queue
h_i	the ratio b_i/b (matching probability)
τ	the steady-state queueing delay for TCP flows
r	congestion or RED-based dropping probability, i.e., the ambient drop probability (common to all flows)

4.3.3. The Analytical Foundation

The analysis is based on deriving the overall loss / admission probabilities in the steady state for an arbitrary flow.

Consider a flow i whose packet reaches the gCHOKe(m) queue. Since the first dropping is due to RED, by assumption, the packet is dropped with probability r or is admitted by RED with probability $1-r$. After admission by RED, a random packet is sampled from the queue for matching. The flow matching probability is $h_i = b_i/b$. If the packets match, the arriving packet earns a bonus to continue further matching by drawing another packet. Matching is aborted either when a no-match is encountered or when a maximum of m comparisons have been tried. Assuming that h_i does not change much in a matching experiment, this results in a sequence of conditional Bernoulli trials. The probability of exactly $k \in \{1, \ldots, m-1\}$ successful matches (or hits) is given by $h_i^k(1-h_i)$ and this results in a loss of $k+1$ packets from flow i.[5] However, if a total of m matchings are performed, the process stops outright with a loss of $m+1$ packets. Therefore, every packet arrival from flow i causes an average packet loss of,

$$r + (1-r)\left[\sum_{k=1}^{m-1}(k+1)h_i^k(1-h_i) + (m+1)h_i^m\right] \tag{4.1}$$

Now, let us assume that a total of P packets of flow i arrive to queue during a long interval of time in the steady state. These arrivals, on the average, incur $P\times(4.1)$ losses. The resulting overall packet loss probability becomes $P\times(4.1)/P$. That means, (4.1) represents the overall packet loss probability p_i caused by a packet arrival. Notationally,

$$p_i = r + (1-r)\left[\sum_{k=1}^{m-1}(k+1)h_i^k(1-h_i) + (m+1)h_i^m\right] \tag{4.2}$$

A second way to derive the overall drop probability is as follows. Consider an arbitrary packet of flow i arriving to queue. This packet survives both dropping and gets queued to the tail with probability $(1-r)(1-h_i)$. This packet may still be dropped when future packets of the same flow trigger potentially multiple matchings. How many comparisons or samplings can be tried per each incoming packet? In other words, how many matching trials can be performed before the matching process becomes aborted either by a flow mismatch or due to exhaustion of the allowed m matchings / hits? The two conditions are explained below,

(1) Matching process stopped by a flow mismatch. If $n \leq m$ comparisons are executed, the first $n-1$ of them must be matching or hits and the last one is the mismatch that aborts the process. This happens with probability $h_i^{n-1}(1-h_i)$.

[5] k matched packets plus the arriving packet are dropped.

(2) Matching process stopped outright after m successful hits. This can happen with probability of h_i^m.

The expected number of matching trials per arriving packet is the sum,

$$\sum_{n=1}^{m} n \cdot (1-h_i)h_i^{n-1} + mh_i^m \tag{4.3}$$

Consequently, when $m \to \infty$, the number of matching trials follows a geometric distribution. Otherwise, it follows the truncated geometric distribution

Since a steady state queueing delay τ is assumed, an average of $x_i(1-r)\tau$ flow i packets arrive during τ. A total of $\tau x_i(1-r)\left[\sum_{n=1}^{m} n(1-h_i)h_i^{n-1} + mh_i^m\right]$ flow matchings can be tried before the enqueued packet gets transmitted. The probability of a trial failing to match the enqueued packet is $(1-1/b)$. Therefore, the overall probability with which the enqueued packet of flow i survives all droppings is given by,

$$1-p_i = (1-r)(1-h_i)\left(1-\frac{1}{b}\right)^{\tau x_i(1-r)\left[\sum_{n=1}^{m} n(1-h_i)h_i^{n-1}+mh_i^m\right]}. \tag{4.4}$$

4.4. UDP Throughput Analysis of a gCHOKe(m) Queue

Using the gCHOKe model developed in the last section, we are interested in how much bandwidth UDP can "steal" in the presence of many TCP sources. For a given gCHOKe(m) queue, the only independent parameter in the analysis is the incoming rate x_0 of UDP.

We assume that the gCHOKe link is fully utilized, i.e.,

$$x_0(1-p_0) + Nx_1(1-p_1) = C. \tag{4.5}$$

For large N,

$$h_1 = \frac{b_1}{b} = \frac{b_1}{b_0 + Nb_1} \leq \frac{1}{N} \approx 0. \tag{4.6}$$

Using (4.6) in (4.2), we get for a TCP flow

$$p_1 \approx r. \tag{4.7}$$

Eqs. (4.6) and (4.7) imply that TCP packets seldom trigger matching and are mostly dropped due to ambient dropping. This is encouraging since TCP flows suffer losses no more than they would under RED (see also Sec. 4.6.2).

The approximations (4.6) and (4.7) together with (4.5) can be used to derive τ. The number of TCP packets in the buffer is $Nb_1 = b - b_0 = b(1 - h_0)$, and the aggregate TCP rate is given by $Nx_1(1 - p_1)$. By virtue of Little's Theorem,

$$\tau = \frac{Nb_1}{Nx_1(1-p_1)} = \frac{b(1-h_0)}{C - x_0(1-p_0)}. \tag{4.8}$$

From (4.8) and (4.4), we find for flow 0,

$$1 - p_0 = (1-r)(1-h_0)\left(1 - \frac{1}{b}\right)^{\frac{bx_0(1-r)(1-h_0)}{C-x_0(1-p_0)}\left[\sum_{n=1}^{m} n(1-h_0)h_0^{n-1} + mh_0^m\right]}$$

For large b, the approximation $(1-1/b)^b \approx e^{-1}$ can be used to simplify the above equation to,

$$1 - p_0 = (1-r)(1-h_0)e^{-\frac{x_0(1-r)(1-h_0)}{C-x_0(1-p_0)}\left[\sum_{n=1}^{m} n(1-h_0)h_0^{n-1} + mh_0^m\right]} \tag{4.9}$$

From (4.2),

$$\begin{aligned} 1 - p_0 &= 1 - \left(r + (1-r)\left[\sum_{k=1}^{m-1}(k+1)h_0^k(1-h_0) + (m+1)h_0^m\right]\right) \\ &= (1-r)\left(1 - \left[\sum_{k=1}^{m-1}(k+1)h_0^k(1-h_0) + (m+1)h_0^m\right]\right) \\ &= (1-r)(1 - \mathrm{p}_{gCHOKe}(m)) \end{aligned} \tag{4.10}$$

where,

$$\begin{aligned} \mathrm{p}_{gCHOKe}(m) &:= \sum_{k=1}^{m-1}(k+1)h_0^k(1-h_0) + (m+1)h_0^m \\ &= 2h_0 + \sum_{k=2}^{m} h_0^k. \end{aligned} \tag{4.11}$$

For a packet to be finally transmitted, it must escape both the ambient dropping and successive matching trials from incoming packets. The factor $(1 - r)$ in (4.10) corresponds to the probability of surviving the ambient (RED) drop, while the second factor $(1 - \mathrm{p}_{gCHOKe}(m))$ is the probability of escaping gCHOKe based packet loss. Therefore, $\mathrm{p}_{gCHOKe}(m)$ is the packet loss probability *solely* due to gCHOKe. $\mathrm{p}_{gCHOKe}(m)$ captures a packet's drop probability caused by the packet matching queued packets on its arrival or by being matched by future packets of the same flow. It is clearly marked in the gCHOKe schematic depicted in Fig. 4.4. Trivially for CHOKe,

$$\mathrm{p}_{gCHOKe}(1) = \mathrm{p}_{choke} = 2h_0. \tag{4.12}$$

Remark It is easy to see that $\mathrm{p}_{gCHOKe}(m),\ m > 1$ is convex increasing with h_0 while p_{choke} v.s. h_0 is linear. That means, at higher buffer shares, gCHOKe(m), $m \neq 1$ increases its drop rate faster than CHOKe.

Equating (4.10) and (4.9) for flow 0 and simplifying, we obtain

$$\begin{aligned}\frac{1-h_0}{1-\mathrm{p}_{gCHOKe}(m)} &= \exp\left(\frac{x_0(1-r)(1-h_0)}{C-x_0(1-p_0)}\left[\sum_{n=1}^{m} n(1-h_0)h_0^{n-1}+mh_0^m\right]\right)\\ &= \exp\left(\frac{x_0(1-r)(1-h_0)/C}{1-x_0(1-p_0)/C}\left[\sum_{n=1}^{m} n(1-h_0)h_0^{n-1}+mh_0^m\right]\right)\end{aligned} \tag{4.13}$$

Define the UDP utilization μ_0,

$$\mu_0 = x_0(1-p_0)/C. \tag{4.14}$$

From (4.10), (4.11), and (4.14), we obtain for $x_0(1-r)/C$,

$$x_0(1-r)/C = \mu_0/(1-\mathrm{p}_{gCHOKe}(m)) \tag{4.15}$$

Remark (4.15) effectively captures the UDP traffic input to the gCHOKe block. See Fig. 4.4.

Using (4.14) and (4.15) in (4.13), we obtain

$$\frac{1-h_0}{1-\mathrm{p}_{gCHOKe}(m)} = \exp\left(\frac{\mu_0}{1-\mu_0}\cdot\frac{(1-h_0)}{1-\mathrm{p}_{gCHOKe}(m)}\left[\sum_{n=1}^{m} n(1-h_0)h_0^{n-1}+mh_0^m\right]\right) \tag{4.16}$$

Equation (4.16) is the departure point for deriving the UDP throughput and buffer shares in a gCHOKe(m) queue. Taking the ln on both sides and organizing, we obtain generic expression for the UDP utilization,

$$\mu_0 = \frac{\ln\left(\dfrac{1-h_0}{1-\mathrm{p}_{gCHOKe}(m)}\right)}{\gamma_m(h_0)+\ln\left(\dfrac{1-h_0}{1-\mathrm{p}_{gCHOKe}(m)}\right)} \tag{4.17}$$

where

$$\begin{aligned}\gamma_m(h_0) &= \frac{(1-h_0)}{1-\mathrm{p}_{gCHOKe}(m)}\left[\sum_{n=1}^{m} n(1-h_0)h_0^{n-1}+mh_0^m\right]\\ &= \frac{1-h_0^m}{1-\mathrm{p}_{gCHOKe}(m)}.\end{aligned} \tag{4.18}$$

4.4.1. Examples

In this section, we clarify the queue properties using $m = 1, 2, 4$ as examples. Note that CHOKe is just a special case when $m = 1$. We reserve a special mention for $m \to \infty$ in Sec. 4.4.2.

$$\mathrm{p}_{gCHOKe}(1) = \mathrm{p}_{choke} = 2h_0, \qquad \gamma_1(h_0) = \frac{1-h_0}{1-2h_0}$$

$$\mathrm{p}_{gCHOKe}(2) = h_0^2 + 2h_0, \qquad \gamma_2(h_0) = \frac{1-h_0^2}{1-(h_0^2+2h_0)}$$

$$\mathrm{p}_{gCHOKe}(4) = h_0^4 + h_0^3 + h_0^2 + 2h_0, \qquad \gamma_4(h_0) = \frac{1-h_0^4}{1-(h_0^4+h_0^3+h_0^2+2h_0)}$$

Substituting the above equations into (4.17) gives the UDP throughput, graphically illustrated in Fig. 4.5 for CHOKe, and gCHOKe with $m = 2, 4, \infty$.

The figure shows that even with $m = 2$, the protection quality of gCHOKe is superior to CHOKe. The difference between gCHOKe(4) and gCHOKe(∞) results is very small. Practically, this means we require per arriving packet only few matching trials to accurately approximate the best protection afforded by the gCHOKe(∞) scheme. The peak UDP utilizations are found to be $\mu_{0,m=4} = 20.6\%$ and $\mu_{0,m=\infty} = 20.5\%$ both measured around $h_0 = 29.2\%$.

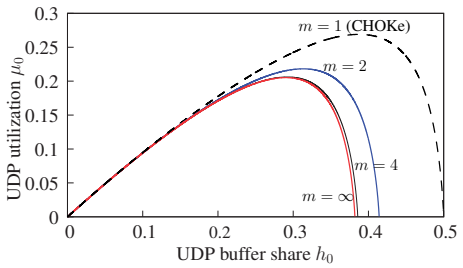


Figure 4.5.: μ_0 v.s. h_0 under CHOKe, and gCHOKe with $m = 2/4/\infty$.

4.4.2. Properties of gCHOKe(∞)

Since $m \to \infty$ is the gCHOKe benchmark in flow protection, we will study the properties of this queue further in this section. The only stopping condition for the matching process is a mismatch. Consequently, the terms $(m+1)h_i^m$ and mh_i^m in (4.2), (4.3) and subsequent equations vanish as $m \to \infty$.

For this queue, we opt to drop all references of m in this section. Equations (4.2), (4.4) and (4.11) for the UDP flow become (see also [27]),

$$p_0 = r + (1-r)\frac{2h_0 - h_0^2}{1-h_0} \tag{4.19}$$

$$1 - p_0 = (1-r)(1-h_0)\left(1 - \frac{1}{b}\right)^{\tau x_0 (1-r)/(1-h_0)} \tag{4.20}$$

$$\mathrm{p}_{gCHOKe} = 2h_0 + \sum_{k=2}^{\infty} h_0^k = \frac{2h_0 - h_0^2}{1-h_0}. \tag{4.21}$$

In addition, setting $m \to \infty$ in (4.18), we obtain

$$\gamma(h_0) = \frac{1-h_0}{h_0^2 - 3h_0 + 1}. \tag{4.22}$$

Upon substitutions into (4.17), μ_0 in gCHOKe(∞) becomes,

$$\mu_0 = \frac{\ln\left[(1-h_0)\gamma(h_0)\right]}{\gamma(h_0) + \ln\left[(1-h_0)\gamma(h_0)\right]}. \tag{4.23}$$

The next two theorems summarize the limit properties of this queue.

Theorem 4.4.1 *The maximum UDP link utilization under gCHOKe(∞) is $\mu_0^{max} \approx 20.5\%$, which is achieved when $h_0 \approx 29\%$.*

Proof Eq. (4.23) has a maximum within the allowed h_0 range (see Fig. 4.5 and the next theorem). Numerically, we obtain the maximum at $h_0^* \approx 0.29$, and

$$\mu_0 \le \frac{\ln\left[(1-h_0^*)\gamma(h_0^*)\right]}{\gamma(h_0^*) + \ln\left[(1-h_0^*)\gamma(h_0^*)\right]} \approx 0.205. \quad \blacksquare$$

Theorem 4.4.2 *The UDP buffer share in gCHOKe(∞) is upper-bounded as $h_0 \le (3-\sqrt{5})/2 \approx 38.2\%$.*

Proof The proof follows from (4.21) and the constraint that $\mathrm{p}_{gCHOKe} = \dfrac{2h_0 - h_0^2}{(1-h_0)} \le 1$. Solving $2h_0 - h_0^2 \le 1 - h_0$ gives $h_0 \ge \frac{3+\sqrt{5}}{2}$ or $h \le \frac{3-\sqrt{5}}{2}$. The first is invalid since h_0 cannot be greater than 1. The proof follows from the second. $\blacksquare$

From Theorems 4.4.1 and 4.4.2, it is easy to see that large UDP buffer occupation does not generally yield high UDP utilization in gCHOKe due to the leaky nature of the queue. This property is a sharp contrast to nonleaky queues with FIFO service discipline, such as RED. In addition, as h_0 increases from 0.29 to $h_0^{max} = (3-\sqrt{5})/2 \approx 0.382$, p_{gCHOKe}, as a convex function of h_0, rapidly increases to 1 and the UDP utilization quickly falls off from the maximum possible $\mu_0^* = 0.205$ to near *zero*.

From (4.15), (4.21) and (4.22), we find the ratio of UDP traffic admitted by gCHOKe:

$$\frac{\mu_0}{x_0(1-r)/C} = \frac{h_0^2 - 3h_0 + 1}{1-h_0} = \frac{1}{\gamma(h_0)}. \tag{4.24}$$

That means, out of $x_0(1-r)$ UDP rate that survives the ambient drop, $\mu_0 C$ are transmitted by gCHOKe and $x_0(1-r) - \mu_0 C$ are dropped by gCHOKe. The rate of gCHOKe based dropping is then,

$$\begin{aligned}\frac{x_0(1-r) - \mu_0 C}{x_0(1-r)} &= 1 - \frac{\mu_0 C}{x_0(1-r)} \\ &= 1 - \frac{1}{\gamma(h_0)} = \frac{2h_0 - h_0^2}{1-h_0} = \mathrm{p}_{gCHOKe}.\end{aligned} \tag{4.25}$$

It follows that $1/\gamma(h_0)$ and p_{gCHOKe} are gCHOKe's admission and dropping rates respectively.

4.4.3. Multiple UDP Flows

As long as the number of UDP flows are not too many, or that the aggregate UDP traffic rate is not too high to turn the ambient drop rate r significant (see Sec. 4.5.1), the gCHOKe model can be extended with multiple ν UDP flows. Without loss of generality, we assume that the UDP flows have equal source rates x_0. This change is reflected in the queueing delay τ. Eq. (4.8) changes to,

$$\tau = \frac{Nb_1}{Nx_1(1-p_1)} = \frac{b(1-\nu h_0)}{C - \nu x_0(1-p_0)}. \tag{4.26}$$

Inserting (4.26) into (4.4) and following similar arguments as before, we obtain for UDP utilization (instead of (4.17))

$$\mu_0 = \frac{\ln\left(\dfrac{1-h_0}{1-\mathrm{p}_{gCHOKe}(m)}\right)}{\gamma_m(h_0) + \nu \ln\left(\dfrac{1-h_0}{1-\mathrm{p}_{gCHOKe}(m)}\right)} \tag{4.27}$$

where $\mathrm{p}_{gCHOKe}(m)$ is still given by (4.11) and

$$\gamma_m(h_0) = \frac{(1-\nu h_0)}{1-\mathrm{p}_{gCHOKe}(m)} \left[\sum_{n=1}^{m} n(1-h_0)h_0^{n-1} + mh_0^m \right] = \frac{(1-\nu h_0)\sum_{k=1}^{m} h_0^{k-1}}{1-\mathrm{p}_{gCHOKe}(m)}. \tag{4.28}$$

For $m = \infty$, we have:

$$\gamma_m(h_0) = \frac{(1-\nu h_0)}{(1-h_0)(1-\mathrm{p}_{gCHOKe}(m))}.$$

Note that (4.11), (4.27) and (4.28) are generic system equations that describe queue properties of gCHOKe of arbitrary `maxcomp` in the presence of one or more UDP flows of similar source rates x_0.

Table 4.2.: Summary of steady state gCHOKe model with multiple UDP inputs.

Parameter	gCHOKe(m) Model Formulation	Equations
gCHOKe drop date	$\mathrm{p}_{gCHOKe}(m) = 2h_0 + \sum_{k=2}^{m} h_0^k$	(4.11)
Intermediate variable	$\gamma_m(h_0) = \frac{(1-\nu h_0)\sum_{k=1}^{m} h_0^{k-1}}{1-\mathrm{p}_{gCHOKe}(m)}$	(4.28)
UDP utilization	$\frac{\ln[(1-h_0)/(1-\mathrm{p}_{gCHOKe}(m))]}{\gamma_m(h_0)+\nu\ln[(1-h_0)/(1-\mathrm{p}_{gCHOKe}(m))]}$	(4.27)
UDP I/O relation	$x_0(1-r)/C = \mu_0/(1-\mathrm{p}_{gCHOKe}(m))$	(4.15), (4.24)

A summary of the theoretical results for the gCHOKe(m) model is given in Table 4.2. The table is useful for understanding the relationships between the various parameters of the model. Many of them are self-explanatory and can be obtained from the formulae above. The "I/O(input/output)" in the last row refers to the relation between the input UDP traffic after ambient dropping, i.e., $x_0(1-r)$, and the UDP utilization or transmission rate $\mu_0 C$.

4.5. Model Validation

We have implemented gCHOKe and all simulations are performed in ns2-34 using the following default parameters. $m = 10$ for gCHOKe(∞). There are $N = 400$ TCP sources, each adopting SACK and connected to R_1 (see Fig. 4.3) by access links whose latencies are uniformly distributed on $[30, 50]$ms. All packets are of 1000 bytes size. The bottleneck link capacity is $C = 45$Mbps, its link latency is 50ms and buffer capacity $B = 1$MB, which is around 100% of the bandwidth-delay product. The RED buffer thresholds are set at $\min_{th} = 100$ and $\max_{th} = 1000$.

Flow start times are random and uniformly distributed on $[0, 10]$s. All simulations are replicated 10 times and run for 200s, but only the results of the last 100s are considered. The 95% confidence intervals are very small and hence are not reported.

As mentioned before, the only independent parameter of the model is the input UDP traffic rate x_0 ($\nu = 1$ is assumed, unless stated otherwise). We study gCHOKe queue properties as we vary the input UDP rate x_0 from $0.1C$ to $10C$.

4.5.1. Impact of Drop Reversal

In the interests of analytic simplicity, the actual RED and gCHOKe based droppings depicted in Fig. 4.1 have been reversed in Fig. 4.4. We remark that the drop reversal does not introduce noticeable p_0 approximation errors in (4.2). If we followed the actual gCHOKe scheme depicted in Fig. 4.1, instead of (4.2), the total loss rate would be,

$$p_0 = \mathrm{p}_{gCHOKe}(m) + r(1 - h_0) \tag{4.29}$$

where $\mathrm{p}_{gCHOKe}(m)$ is given by (4.11).

We assert that the difference between (4.29) and (4.2) is $r(\mathrm{p}_{gCHOKe} - h_0)$ which is insignificant, largely due to small r. Fig. 4.6 plots r as the input UDP traffic rate varies from $0.1C$ to $10C$. Compared to that of plain RED, the ambient loss rate r in gCHOKe is generally lower. This is because gCHOKe drops excessively in response to increasing input x_0, making the average queue size `avg` significantly smaller in gCHOKe. Even though UDP load increases 100-fold, r in gCHOKe rises in the worst case to 9.1%.

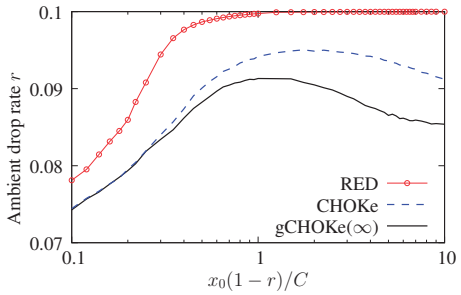
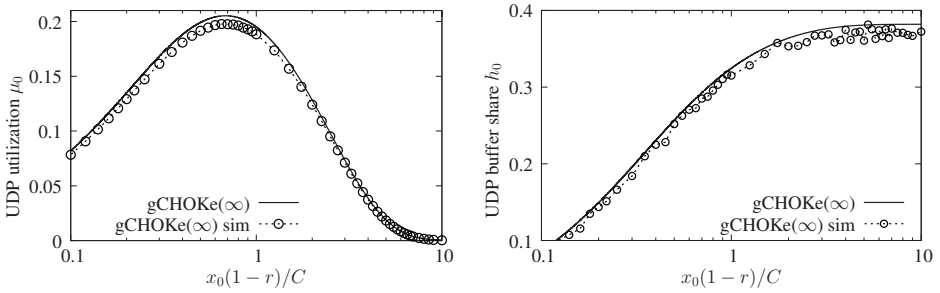


Figure 4.6.: Ambient loss rates r

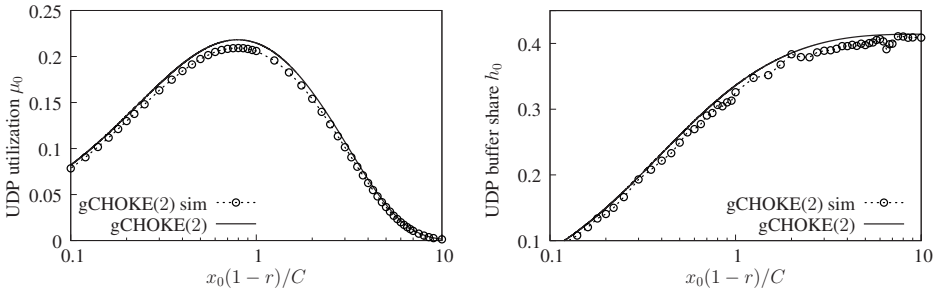
4.5.2. UDP Buffer Shares and Utilizations

Fig. 4.7 demonstrates both simulation and theoretical results on UDP utilization and buffer shares as we vary the input UDP load from $0.1C$ to $10C$. The theoretical plots are obtained based on parameter interdependencies summarized in Table 4.2.

The match between theoretical and simulation results is strikingly remarkable, validating the analytical model and results of gCHOKe. The figure illustrates results only for $m = 2$ and $m = \infty$. We obtain consistent observation for other m values, but we do not show the results to save space. For example, for CHOKe where $m = 1$, our analysis and simulations reproduce those results reported in [98].

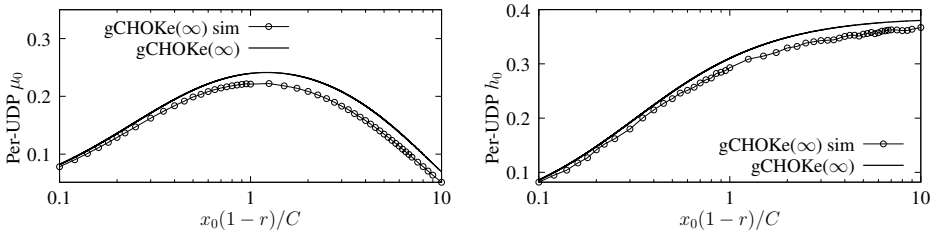


(a) gCHOKe(∞): μ_0 (b) gCHOKe(∞): h_0



(c) gCHOKe(2): μ_0 (d) gCHOKe(2): h_0

Figure 4.7.: Model validation with a single UDP flow ($\nu = 1$): μ_0 and h_0 under gCHOKe with `maxcomp` $m = \infty, 2$.



(a) gCHOKe(∞): μ_0 (b) gCHOKe(∞): h_0

Figure 4.8.: Model validation involving $\nu = 2$ UDP flows. Per-UDP flow μ_0 and h_0 under gCHOKe with `maxcomp` $m = \infty$.

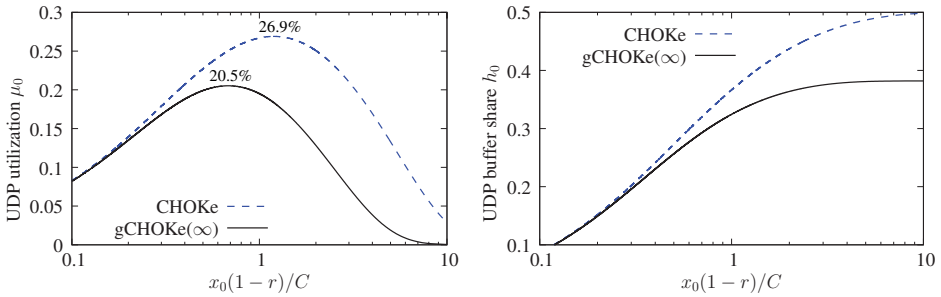
Likewise, the graphs in Fig. 4.8 validate the model presented in Sec. 4.4.3 using $\nu = 2$ simultaneous UDP flows. As can be seen, the model fairly matches the simulation results, despite the bottleneck being subjected to an aggregate UDP traffic rate of up to 20× the link capacity. Remarkably, the two UDP flows together take up about 75% of the buffer space, but receive a total throughput less than 10% of the link capacity.

4.6. Results and Observations

4.6.1. Main Results and Observations

In this section, we are concerned with observations regarding the main performance metrics of this chapter—UDP utilization and buffer shares. Unless stated otherwise, the following observations are based on the analytical model and results. We refer the reader to Fig. 4.9, which concisely shows the comparative flow control performance of CHOKe and gCHOKe(∞) (for other m, see the end of this section). We highlight the following salient points:

- UDP utilization bounds: CHOKe (26.9%) and gCHOKe(∞) (20.5%); UDP buffer share bounds: CHOKe (50%) and gCHOKe (38.2%). Both gCHOKe(∞) bounds are tighter by over 20% over CHOKe's. The corresponding figures for gCHOKe(2) are $\mu_0^{max} = 21.8\%$, $h_0^{max} = 41.4\%$.
- For low and moderate UDP traffic rate, there is no significant difference between CHOKe and gCHOKe in terms of UDP utilization or buffer share levels.
- Increasing UDP input rate initially increases the buffer share, but does not allow complete buffer monopoly (see Fig. 4.9(b)). UDP's buffer shares eventually stabilize around their peaks (see also Fig. 4.5).
- In sharp contrast to Drop-Tail and RED (e.g., see Fig. 4.2), while indefinitely increasing UDP input traffic will increase the corresponding buffer share, it may not yield higher utilization for UDP in both CHOKe an gCHOKe(∞) (see Fig. 4.9(a)). Particularly, increasing the input rate beyond $0.682C$ backfires in gCHOKe(∞) since each incoming UDP packet potentially triggers a sequence of successful comparisons at soaring rate. UDP utilization peaks at 20.5% in gCHOKe(∞) (*vs.* 26.9% in CHOKe). This occurs when 29% of the packets in buffer are UDP (*vs.* 39% for CHOKe). This means, as UDP buffer ratio increases from 29% to 38.2% due to increased input, $\mathrm{p}_{gCHOKe}(\infty) \to 1$ and UDP utilization steadily declines from the maximum 20.5% to almost 0.
- The contrast between high UDP buffer occupancy and low UDP utilization (see Fig. 4.9(a) and Fig. 4.9(b)) at high incoming UDP rate implies nonuniform UDP packet distribution in the queue where most of the UDP packets may have been clustered closer to the tail of the queue.

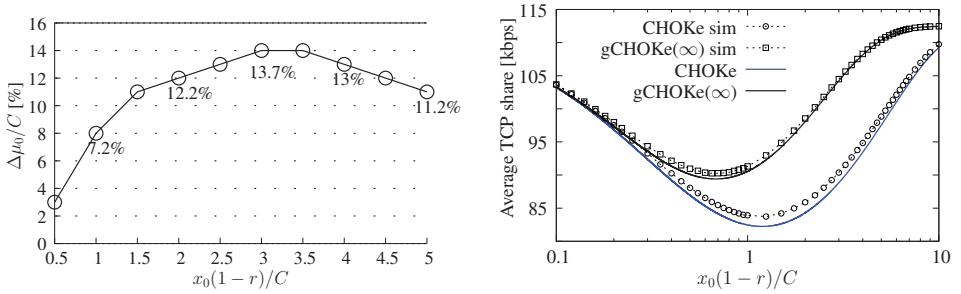


(a) u_0 versus $x_0(1-r)/C$. (b) h_0 versus $x_0(1-r)/C$.

Figure 4.9.: UDP share u_0 and h_0 *vs.* UDP incoming rate x_0.

An intuitive explanation of the last property is as follows. It is easy to see that packets enter the queue tail at a rate $[x_0(1-h_0)+Nx_1(1-h_1)](1-r)$, but exit only at the rate of C. When x_0 is high, since $h_1 \approx 0$ (see Eq. 4.6) and r is assumed insignificant (see Fig. 4.6), most of the UDP packets must have been dropped by matching as they advance towards the head of the queue. Assuming a fluid model, we say the UDP flow is *thinned* by flow matching. The power of thinning depends on the UDP flow rate x_0. The flow controlling or thinning powers of the schemes are explicitly captured by Eq. (4.20). An arriving UDP packet gets enqueued at queue tail with probability $(1-r)(1-h_0)$. The UDP flow gets thinned due to flow matching by factors of $(1-1/b)^{\tau x_0(1-r)}$ and $(1-1/b)^{\tau x_0(1-r)/(1-h_0)}$, respectively for CHOKe and gCHOKe(∞), before the flow packet can reach the head of the queue. When h_0 is excessively high, due to high input UDP traffic rate x_0, the thinning exponent of gCHOKe(∞) increases faster. This allows gCHOKe(∞) to restrict the UDP bandwidth and buffer shares to lower levels. It is worth highlighting, however, the average number of matching trials is reasonably low. For gCHOKe(∞), the average is $1/(1-h_0)$ as implied by (4.3) for geometric distribution. In the extreme case when $h_0^{max} = 0.382$, each arriving UDP packet in gCHOKe(∞) performs, on average, $1/(1-h_0^{max}) \approx 1.62$ matching trials.

Fig. 4.10(a) illustrates the difference between UDP utilizations obtained under the CHOKe and gCHOKe(∞) schemes for selected UDP traffic rates. While it is difficult to derive $\Delta\mu = \mu_{choke}(x_0) - \mu_{gCHOKe}(x_0)$ in closed form, the individual UDP utilization values can be obtained numerically (or read from Fig. 4.9(a)) for a given UDP rate x_0. The resulting $\Delta\mu_0$ is plotted in *percentage of* C in Fig. 4.10(a). As can be seen, gCHOKe wields better control on high bandwidth flows. Remarkably, up to 14% of the total link bandwidth can be saved by switching to gCHOKe(∞). To give numbers, consider the scenario with $x_0 = 3C$. The UDP utilizations are found to be 21.02% and 7.36%, respectively, under CHOKe and gCHOKe(∞). This corresponds to a saving of 13.66% of C or a decent improvement of 65% over CHOKe. As a consequence, gCHOKe provides better throughput to

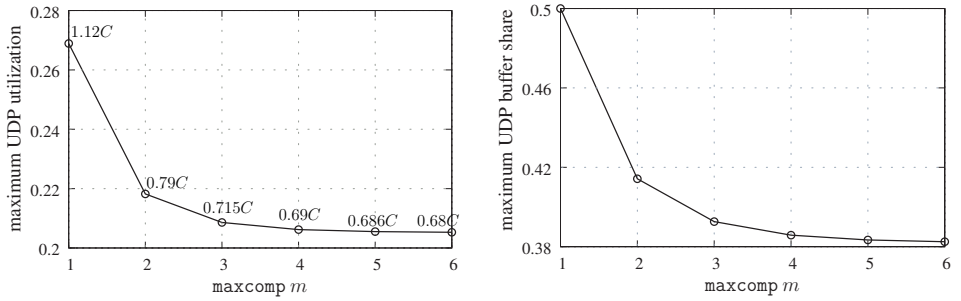


(a) (b)

Figure 4.10.: More bandwidth is available for TCP flows under gCHOKe(∞).

TCP flows, as demonstrated in Fig. 4.10(b). At any level of UDP source rate, the average perflow TCP throughput obtained from simulation fits the theoretical TCP throughput obtained from (4.5) very well.

Finally, we turn to results when $m \neq \{1, \infty\}$. Fig. 4.11 demonstrates the bounds in UDP bandwidth and buffer space occupation and complements the results in Fig. 4.5. Both the maximum bandwidth and buffer occupancy that can be claimed by the unresponsive flow decline with increasing m. For each point in Fig. 4.11(a), we indicate in units of C the input traffic rate x_0 which results in the maximum UDP utilization. For instance, for CHOKe ($m = 1$), the UDP utilization cannot exceed 26.9% and this can be achieved when input UDP traffic rate is $x_0 \approx 1.12C$. The figures clearly illustrate that the controlling power of gCHOKe improves with m.



(a) (b)

Figure 4.11.: Maximum possible (a) bandwidth and, (b) buffer space used by the unresponsive flow under gCHOKe(m).

4.6.2. Additional Results and Observations

The focus of the previous section has been on UDP bandwidth and buffer shares. To gain a deeper understanding of gCHOKe, we comment on loss rates and queueing delays in this section.

Overall TCP and UDP loss rates: It is important to note that TCP flows and the unresponsive flow have drastically different total loss rates. Since flow matching is negligible, TCP flows suffer losses only due to congestion, i.e., r. However, the unresponsive flow suffers both congestion based loss and gCHOKe-based loss. That is,

- Total TCP loss rate $p_1 = r$
- Total UDP loss rate $p_0 = r + (1 - r)\mathrm{p}_{gCHOKe}(m)$

Since $r << 1$ (especially for low input rates, see Fig. 4.6), TCP flows rarely suffer losses while losses to the unresponsive flow are largely due to flow matching. As depicted in Fig. 4.6, gCHOKe(∞) has the least TCP loss rate r.

Queueing delay τ: Recall from (4.8) that the queueing delay experienced by TCP/UDP packets in leaky gCHOKe is $\tau_{gCHOKe} = \frac{b(1-h_0)}{C(1-\mu_0)}$) where b is the steady-state backlog. For a nonleaky RED queue, quite simply, $\tau_{RED} = b/C$. Obviously, b depends on the load and the dropping scheme (that is, `maxcomp`). When UDP load x_0 increases indefinitely in RED, $\tau_{RED} = B/C$, where B is the buffer size. In gCHOKe, however, there are two factors tugging the τ_{gCHOKe} in opposite directions. Increasing the x_0 increases b but decreases the factor $(1 - h_0)/(1 - \mu_0)$.[6] Generally, initial increases of x_0 quickly ramp up the b, with the net effect of increasing τ_{gCHOKe}. When x_0 is much higher, the b increases are milder, and τ appears to be pulled down by the plummeting $(1 - h_0)/(1 - \mu_0)$. The simulation results are plotted in Fig. 4.12.

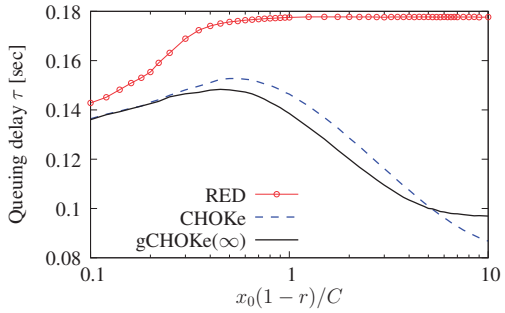


Figure 4.12.: Comparison of TCP queueing delays τ in RED, CHOKe, gCHOKe(∞).

[6] For example, for CHOKe, we proved in [25] that $(1 - h_0)/(1 - \mu_0) \approx 1/2$ at high x_0.

Summary of the results and observations: From the results so far, we can conclude the following. *Without tampering with the desirable features of CHOKe, gCHOKe offers an improved control on the use of network resources by (potentially) malicious flows. Coincidentally, rate adaptive flows do not only obtain better share of resources but also generally experience fewer packet losses and lower queueing delays.*

4.7. Discussion

4.7.1. General Discussion

This section reflects on miscellaneous topics useful for thorough understanding of gCHOKe.

- Processing and complexity: Having inherited its design principle from CHOKe, gCHOKe is simple and completely stateless, and its service discipline is FIFO. Flow matching may be done with a simple implementation in hardware. For dropping packets, the authors of CHOKe [78] proposed marking them in their header, rather than plucking them out of their linked list. When these packets reach at queue head, instead of being transmitted, they are dropped.

  gCHOKe routers are configured with `maxcomp` which is a delicate tradeoff between protection level and packet processing overhead. However, we see that protection with $m \geq 4$ is approximately the same as that of $m = \infty$, and that even with $m = \infty$, the average matching trials per arrival in the worst case is only 1.62 as discussed in Sec. 4.6.1. Furthermore, only the rogue flow incurs matching on its packet arrival. The rest of the flow population requires processing no further than RED.

- gCHOKe vs. M-CHOKe: Note that gCHOKe is not M-CHOKe. M-CHOKe [78] is a CHOKe extension whereby M packets are randomly sampled from the buffer and compared to the arriving packet. The matching packets are all dropped together with the arrival. Using the system setup and assumptions of this chapter for M-CHOKe, the M-CHOKe based dropping from a flow with buffer share h_i can be approximated by,

$$\sum_{k=1}^{M}(k+1)\binom{M}{k}h_i^k(1-h_i)^{M-k}. \tag{4.30}$$

  As can be seen, the number of M-CHOKe flow matching trials per arrival is a Binomially distributed random variable. With large M, the Binomial coefficients rapidly increase the flow matching probabilities. Consequently, the TCP matching probabilities may not be too insignificant to be ignored. While this maybe desirable to control the rogue flow, it may degrade TCP

throughput by incurring multiple TCP packet losses. Further work is required to understand the true behavior.

4.7.2. Multiple Unresponsive Flows and Multi-link Situations

Sec. 4.4.3 and Fig. 4.8 present some insights on gCHOKe in the presence of multiple UDP flows. Fig. 4.13 demonstrates a simulative performance comparison of gCHOKe(∞) against CHOKe when there are two UDP flows of equal source rate x_0 (see also Fig. 4.8). The advantage of gCHOKe is very clear, particularly when the UDP flows are highly aggressive.

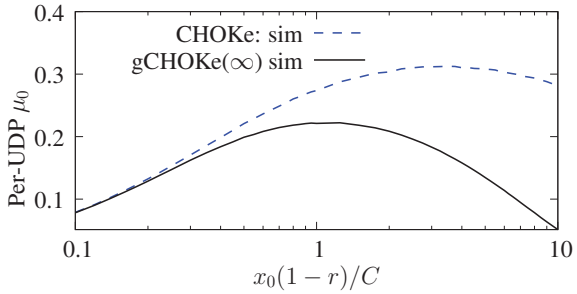


Figure 4.13.: Comparison of gCHOKe(∞) and CHOKe in the presence of two UDP flows, each of rate x_0.

Nevertheless, in the presence of many unresponsive flows, the controlling power of gCHOKe may decline as gCHOKe matching becomes less frequent. In such cases, much like in CHOKe, we may define M-gCHOKe(m). Upon a packet arrival, the scheme picks M packets. For those matching packets, the matching continues in parallel until each is aborted by a no-match or as limited by m serial attempts. It may be interesting to study the comparative performance and complexity of this scheme in relation to M-CHOKe, but we leave this as a future work.

Another way to deal with unresponsive UDP flows is to treat them as a single aggregate flow when they arrive at the gCHOKe queue. This is indeed simpler to implement since identifying individual UDP flows compares several header fields, e.g., the 5-tuple (source and destination IP addresses and ports, and protocol type). Identifying the UDP aggregate, on the other hand, needs to consider only the protocol type field. For the multiple-congested-links simulation experiment shown in Fig. 4.14, the latter way is adopted. The eight connected links $L_1, \cdots, L_8$ all adopt the gCHOKe scheme, and each has capacity $C = 45$Mbps, buffer size 1MB, and link delay of 10ms. The number of TCP flows on each link is 200. One hundred of those TCP flows, together with one UDP flow, pass through end-to-end across the connected links $L_1, \cdots, L_8$. The other 100 TCP flows are cross flows that start at the sources at the bottom and end a single hop away at the sinks on the upper

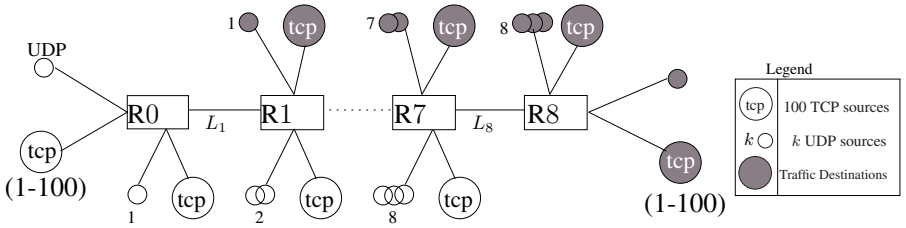


Figure 4.14.: Multiple unresponsive flows under multiple congested gCHOKe links.

part of the figure. There are also an increasing number of cross UDP flows as we move from the first link L_1 to L_8; specifically, link L_i has i cross UDP flows. That means, there are a total of $i+1$ UDP flows traversing link L_i. Each UDP flow has a rate of $C/4$ at its source.

Table 4.3 compares the theoretical values with the simulation averages for the UDP utilizations across each link. Note that the UDP utilization on link L_i, together with the $i+1$ cross UDP flows, forms the UDP input to L_{i+1}. As shown, the theory results predict the simulation results reasonably.

Table 4.3.: gCHOKe with multiple UDP flows under multiple congested links.

Link	1	2	3	4	5	6	7	8
UDP Input (%C)	50	69.9	95.5	120	143	167	190	214
μ_0 (theory, %)	19.9	20.5	19.7	18.3	16.7	15.1	13.5	12
μ_0 (simulation, %)	19.5	20.0	20.1	19.2	17.6	15.9	14.2	12.6

4.7.3. Differences with MLC(I) [88]

A CHOKe-like stateless scheme related to gCHOKe is Multi-level Comparison with Index l, MLC(l), which is proposed for providing max-min fairness in networks controlled by TCP flows [88]. Recall that achieving max-min fairness without a complex stateful perflow mechanism has been a difficult research problem. Before highlighting the differences between gCHOKe and MLC, we present the MLC scheme briefly. MLC maintains several global parameters: block size l, iteration variable h_M, refresh interval Δ_0, target link utilization μ_t, multiplicative increase/decrease factor γ. MLC works as follows. When a packet arrives to an MLC queue, $l-1$ packets are randomly sampled from the queue: if *all the* l packets match, the arriving packet is dropped but the sampled packets remain intact. However, if any of the sampled packets do not match the arriving packet, the above matching experiment is repeated. The number of repetitions is dictated by h_M: It is $\lceil h_M \rceil$ with probability $h_M - \lfloor h_M \rfloor$, and $\lfloor h_M \rfloor$ with the remaining probability of $\lceil h_M \rceil - h_M$. h_M is dynamically adapting to the link utilization. Every Δ_0, the link utilization

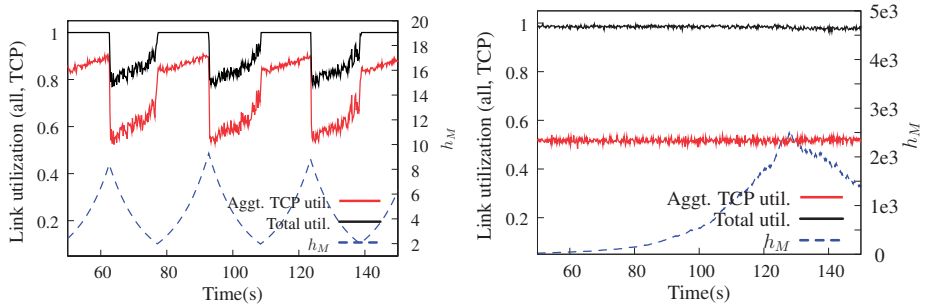
is compared to the target μ_t. If utilization falls below μ_t, the new h_M decreases to h_M/γ. If utilization is higher than μ_t, h_M increases to $h_M \times \gamma$.

Note that the basic idea and objectives behind gCHOKe are different from those of MLC. In gCHOKe, repetition of the matching experiment per arrival is governed by the success of the previous matching experiment. The number of matching retrials is linked to the probability of matching itself, leading to the geometric distribution. A successful "first" matching in gCHOKe drops *at least two* packets. That is how gCHOKe controls the unresponsive flows. However, in MLC, repetition of the matching experiment per arrival is governed by the failure of the previous matching experiment. The number of matching retrials is linked to an external parameter (namely, the target link utilization). Unlike gCHOKe, MLC drops only the arriving packet, but none of the matched sampled packets. This makes MLC more difficult to drop the same amount of traffic.

To highlight the differences, we implemented MLC(l) in ns-2 and conducted simulation experiments with one high rate UDP flow and many TCP flows in a dumbbell topology. We simulate a total of 100 (99 TCP plus 1 UDP) flows. Overall round-trip propagation delays for the 100 sources are uniformly distributed on [40,140]ms. The scheme at the bottleneck is an MLC queue with buffer size 0.5MB, link delay 10ms, and link speed 20Mbps. The MLC configurations are l=2, $\mu_t = 0.98$, and $\gamma = 1.01$. Δ_0 is configured as $\min(\min(RTT_i), 0.1)$. Fig. 4.15 demonstrates how MLC performs with respect to an unresponsive UDP flow of various source rates. To avoid diverging the focus of this chapter, only selected results with two cases of UDP source rates— given in percentages of the bottleneck capacity— are presented.

For UDP rate $x_0 = 25\%C$, Fig. 4.15(a) shows that the TCP link utilization fluctuates. One possible reason is as follows. Due to the aggressive UDP flow, the link utilization is usually over the target μ_t, which in turn increases h_M. Since the packet drop/matching attempts are persistently retried, a high h_M increases the chance of TCP packets getting dropped. Consequently, many of the TCP flows suffer losses simultaneously, and back off. This reduces the overall link utilization. When the utilization falls below the target μ_t, h_M starts to decrease. This cyclic growth of link utilization and h_M continues. Overall in this case, h_M, i.e. the number of trials, remains at a reasonably low level. In addition, UDP traffic gets close to 20% share of the capacity.

For UDP rate $x_0 = 75\%C$, Fig. 4.15(b) shows that h_M can reach at an alarming 2500. Indeed, similar evolution of h_M over time as seen in Fig. 4.15(a) can be observed, but with much larger cycle period. In addition, the total link utilization and the aggregate TCP link utilizations are just below 100% and 55%, respectively, and UDP traffic grabs the remaining 45% of the capacity. In contrast to the first case, the fluctuating TCP link utilization is hardly seen for the increased UDP rate case. This contrast may be interesting to study and is left as a future work.



(a) UDP rate $x_0 = 25\%$ of C.

(b) UDP rate $x_0 = 75\%$ of C.

Figure 4.15.: Total and TCP link utilization and the dynamic evolution of h_M under MLC(2) in the presence of a UDP flow.

Overall, in MLC, h_M is sensitive to the source rate of the UDP flow and can reach a very high value when the UDP source rate increases. Furthermore, when UDP source rate increases, there is no clear limit on the UDP link share, and when there is, the limit may be too loose to protect TCP flows from unresponsive UDP traffic.

4.8. Related Work

Probably due to their impact[7] on the performance of TCP flows, which shape much of the Internet flow dynamics, a lot of interesting queue management schemes have been proposed and studied by the research community. This section is devoted to a discussion on some of the related queue management mechanisms, with emphasis on those intended to improve upon RED and CHOKe.

The classic drop policy Drop-Tail is characterized by severe under-utilization of network resources, see Sec. 2.4.2. To correct this shortcoming and others, researchers proposed and recommended RED for widespread deployment in the Internet [10, 38, 36]. RED is characterized by a packet loss probability which is computed as a function of the average queue size avg and applied indiscriminately to all flows. As explained in Sec. 2.4.2, this does not result in fairness since flows generally have different responses / throughput to RED-type *equal* drop rates.

The research community have therefore proposed RED extensions and other dropping policies to improve flow protection and fairness. Recall Flow RED (FRED) [64], RED with Preferential Dropping (RED-PD) [66], and Stabilized RED (SRED) [74].

[7] Packet losses are implicit indicators of congestion, whereupon most TCP flows back off and decrease their sending rates.

Unlike RED, these policies are based on identifying the high bandwidth flows. Unlike CHOKe, all of them maintain state for all active or in-progress flows, making them partially stateful. By bookkeeping perflow information, the high bandwidth flows can be identified and the shares of their buffer space and bandwidth can be selectively restricted.

Of the above schemes, CHOKe is closest to SRED which is particularly designed to maintain the queue size at a preset value. However, SRED bases the dropping rates on two factors: instantaneous queue size q and the number N of active flows. It maintains a flow cache ("zombie list") which is a list of M flows recently seen at the router. When a packet arrives, SRED randomly samples a record from the cache and determines whether there is a match (Hit 1) or a mismatch (Hit 0). Based on the frequency of hits, computed as an exponential moving average, the number of active flows can be estimated. We highlight two concerns here: (1) there may be lack of quality perflow dropping rates since both q and N are global variables; (2) N estimation is based solely on TCP flow assumption, and this may lead to inaccurate drop rate computation in the presence of non TCP-friendly flows. We illustrate the last concern with a concrete example in Sec. 6.1.

In the remaining part of this section, we turn to some works pertaining to CHOKe extensions, including XCHOKe [15] and REHOKe [42], MLC [88], and CHOKe analysis [25, 77, 96, 98]. We already discussed MLC [88] in Sec. 4.7.3, so we will discuss XCHOKe and RECHOKe in the following.

Both XCHOKe and RECHOKe forgo the design principle of CHOKe since they lack the *stateless* property. They maintain a lookup table, where the flow identities and hit counters of potential misbehaving (malicious) flows are kept. The table is refreshed every time-to-live (TTL) period. When a packet arrives to an XCHOKe/RECHOKe queue, its flow ID is first checked in the table. If there is a table entry for the flow (i.e., a table hit), the packet is marked for dropping with a probability $p^* = \min(1, r \times 2^n)$, where n stands for the flow's hit counter maintained in the same table and r is the RED drop rate. The packet is then sent to the CHOKe module for matching. One main difference between XCHOKe and RECHOKe is how the flow's hits n are counted. In XCHOKe, n depends solely on CHOKe hits. However n in RECHOKe represents: (1) table hit, when the packet's flow ID is found in the table; (2) CHOKe hit when the arriving packet's ID matches that of the randomly chosen packet; (3) RED hit, when the packet is chosen for dropping / marking with the RED drop probability r.

Most of the analytic studies on CHOKe assume a fluid model and generally proceed along two distinct lines of approach: (i) computation of the overall loss rates, e.g., [98], and (ii) derivation of the spatial distributional properties of the queue [25, 77, 96]. The first approach derives the steady state UDP utilization and buffer shares by deriving the overall flow dropping probabilities as shown in this chapter. The second approach is based on obtaining the spatial distribution

of flows (e.g., packet velocity at any position of a CHOKe queue in steady state) using queue properties at the tail and head as boundary conditions.

4.9. Conclusion

Regardless of the outcome of the matching experiment, CHOKe maximally triggers a single trial of flow matching per packet arrival. Therefore, flow protection in CHOKe is flat. In this chapter, we present a suite of active queue management schemes, collectively named gCHOKe, as a lightweight generalizing framework of CHOKe. gCHOKe offers an additional power of protection and at the same time retains the statelessness and simplicity of CHOKe. It rewards a successful flow matching with a bonus trial. Depending on the recent admission history of the flow, or the ratio of flow packets queued in the buffer, a single packet arrival may trigger a succession of packet droppings. Each instance of the gCHOKe framework is indexed by a parameter called `maxcomp` which controls the maximum number of matching trials that can be tried per arriving packet. The quality of protection generally improves with `maxcomp` without significantly increasing the per packet processing complexity. In this way, a flow with relatively many recent arrivals can be controlled further, leaving the network resources (buffer and link bandwidth) to responsive flows. Compared to CHOKe, the flow protection performance of gCHOKe is superior ensuring tighter upper bounds in the use of network resources by unresponsive UDP traffic.

In the next chapter, we investigate the transient behaviors of CHOKe when the UDP traffic rate x_0, the sole independent parameter of all CHOKe steady state models discussed so far in the literature (to the best of our knowledge), is changing. This provides a more thorough understanding of the scheme under more realistic and dynamic situations. As we will see shortly, the steady state properties studied in this chapter are very important for modeling the transient behaviors of the CHOKe queue.

5. Analysis of the Transient Behavior of CHOKe

Previous works including the analysis in Chapter 4 have proven that CHOKe is able to bound both the bandwidth and buffer shares of (a possible aggregate) UDP traffic on a link without introducing complex operational overhead. However, these studies consider, and pertain only to, a steady state where the queue reaches equilibrium in the presence of many TCP flows and an unresponsive UDP flow of fixed arrival rate. If the steady state conditions are perturbed, particularly when the UDP traffic rate changes over time, it is unclear whether the protection property of CHOKe still holds. Indeed, it can be observed, for example, that when the UDP rate suddenly becomes 0 (e.g., when the flow ends), the unresponsive flow may assume close to full utilization in sub-RTT scales, potentially starving out the TCP flows. To explain this apparent discrepancy, this chapter investigates CHOKe queue properties in a transient regime, which is the time period of transition between two steady states of the queue, initiated when the rate of the unresponsive flow changes. Explicit expressions that characterize flow throughput in transient regimes are derived. The results in this chapter provide more in-depth understanding of CHOKe (and by extension, all other gCHOKe[1] variants) in realistic and dynamic network environments, and give some explanation on its intriguing behavior in the transient regime.

The rest of the chapter is structured as follows. Sec. 5.1 explains the motivation using illustrative examples and summarizes the research contributions made. The transient phase or regime is defined as the transition period between two steady states, and we find that understanding the steady state queue models is as relevant. To that end, we review the background required on steady state CHOKe models and queue properties in Sec. 5.2. Sec. 5.3 presents the system setup, basic assumptions and notations used in this chapter. The studied setup is similar to the one in Chapter 4. Our theoretical models are presented in Sec. 5.4. In particular, Sec. 5.4.1 derives the insightful *rate conservation argument* and obtains further simplifying assumptions required for the transient analysis; Sec. 5.4.2 lays out the theoretical foundation on the *spatial distribution model* just before rate change; Sec. 5.4.3 tracks the UDP link utilizations and derives its properties during the transient period. Sec. 5.5 presents model validation and simulation results. Sec. 5.6 concludes the chapter.

[1]The analysis in this chapter can easily be extended to other variants of gCHOKe.

5.1. Motivation and Contribution

5.1.1. Motivating Examples

We clarify our motivation using two example scenarios employing the network setup depicted in Fig. 4.3. There are $N = 100$ TCP sources and a UDP flow whose arrival rate is dynamically varying. The simulation parameters are described in full in Sec. 5.5.

Example 1: We conduct two separate experiments. The initial UDP arrival rate x_0 is $0.5C$ and $0.25C$ for Experiment 1 and Experiment 2, respectively, where C is the link capacity. At $t = 21$s, the x_0 suddenly jumps by factors of 4 and 12 to $2C$ and $3C$, respectively, and then returns back to $0.5C$ and $0.25C$ at $t = 22$s. We conduct 500 replications of the two experiments and the resulting average UDP flow utilizations (as measured using time intervals of 10ms) are shown in Fig. 5.1.

Example 2: The initial UDP arrival rate is $10C$. Starting at $t = 21$s, the input UDP rate alternates between $1C$ and $10C$ every 250ms, as shown in Fig. 5.2. The figure shows UDP utilization averaged over 1000 replications, with measurements taken every 1ms.

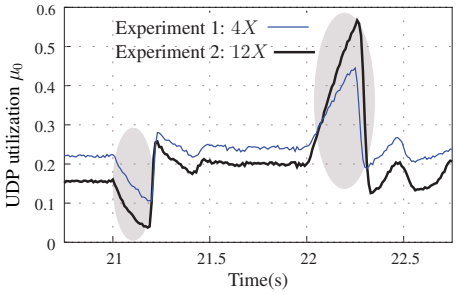


Figure 5.1.: Transient UDP utilizations as x_0 changes by a factor of 4 and 12.

5.1.2. Observation and Objective

In this chapter, we are interested in the peculiar behaviors (e.g., shaded in Fig. 5.1) when UDP arrival rate varies. The behaviors can represent transitions of a CHOKe queue from one steady-state to another and typically lasts no more than a full queueing delay. We note during this transient phase that,

- While UDP buffer and utilization bounds stipulated in earlier works ([96, 98] and Chapter 4) hold in the steady-state, they are easily violated during the transient phase. For example, CHOKe UDP utilization $\mu_0 \leq 27\%$ for $\forall x_0$ during steady-state (refer to Fig. 5.3(a) or Fig. 4.9). However, during the

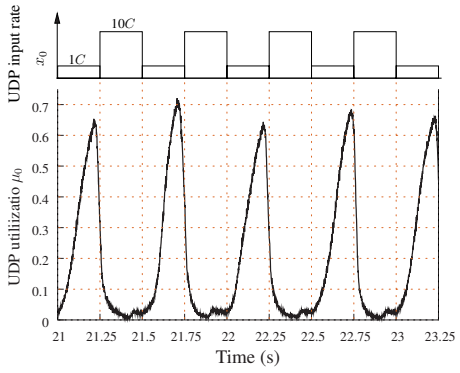


Figure 5.2.: Transient UDP utilizations when x_0 flaps between $1C$ and $10C$.

transient phase, μ_0 may increase abruptly by several factors. In Fig. 5.2, for instance, $\mu_0(21.7s) = 72\%$.

- The queue does not simply undergo a *smooth* transition between two steady states. Surprisingly, the change in transient UDP utilization is not determined by the μ_0 values of the two steady states. Rather, it follows the direction opposite to the arrival rate change. Specifically, if UDP arrival rate goes up, its utilization dips and vice-versa. For example, consider Experiment 2 shown in Fig. 5.1. Instead of steadily increasing from a steady state value $\mu_{0,0.25C} \approx 16\%$ to another steady state value $\mu_{0,3C} \approx 21\%$, the transient UDP link utilization first whittles down to 3.75% at $t \approx 21.18$s. (See Fig. 5.3(a) to find out steady state CHOKe UDP utilization μ_0 values.)

What are the lowest and highest UDP utilizations during the transient phase? Note that these extreme UDP utilizations during the transient regime depend on the measurement interval / window. With large windows, the transient behavior gets diffused and evened out by the adjacent steady-state results. The objective of this work is to obtain the *extreme packet-level UDP utilizations* (i.e., the highest or lowest utilizations as measured for each packet) by analysis.

These transient behaviors have important consequences on the performance of competing flows. For instance, Internet flows, mainly dominated by short Web transfers, see fluctuating available bandwidth. When the UDP arrival rate drops, for example, the strong increase in UDP link utilization may suddenly inflate the completion times of competing flows, or decrease the number of those completing service.

Since the transient regime represents a departure from a stable queue state, its behavior may be influenced by the earlier steady state, as we shall soon see. Therefore, it is fitting to briefly discuss the steady state behaviors, which we do in Sec. 5.2.

5.1.3. Our Contributions

To the best of our knowledge, all previous analytical studies on CHOKe [96, 98, 77, 27, 26] are restricted to the steady state where the traffic rate of the UDP flow is assumed constant. However, this assumption is too restrictive, limiting more in-depth understanding of CHOKe. The focus of this chapter is therefore to contribute to comprehensive understanding of CHOKe (and, by extension, all other gCHOKe variants) in the face of dynamically changing UDP rates while simultaneously generalizing its steady state properties. In particular, this chapter investigates CHOKe queue properties in a transient regime, which is defined as a period of transition between two steady states of the queue and is started when the rate of the unresponsive flow changes.

From a modeling perspective, the study of the transient behavior of CHOKe is an arduous task for two main reasons: (1) the *leaky* nature of the queue, meaning that packets already in queue may be dropped later, (2) the continuous state transition of the queue in the transient regime. Due to (1), the delay of a packet is not merely the backlog the packet sees upon arrival divided by the link capacity as in non-leaky queues. Besides, the spatial packet distribution of a flow in the queue can be nonuniform throughout the queue. Due to (2), many parameters that characterize the queue (e.g., flow matching probability, backlog size, skewed packet distributions of flows in the queue) are likely to be dynamically changing. Both constraints prohibit us from making "safe" simplifying assumptions in analyzing the transient behavior of CHOKe.

Two specific problems are dealt with in this study: (1) how UDP utilization evolves in transient time, and more importantly (2) the limits, i.e., how far the UDP utilization goes up or down in transient time. As we noticed earlier, as the UDP arrival rate goes up or down, its transient transmission rate goes to the opposite direction in a dramatic fashion. For example, when the UDP rate sharply decreases, its utilization rapidly soars, potentially exceeding the steady state limits studied in Chapter 4. Extreme transient behaviors are observed when a very high rate UDP flow abruptly stops. In such cases, UDP transient utilization can suddenly jump from 0% to over, say, 70% (see the examples in the previous section). This intriguing phenomenon cannot be explained with literature results.

Our contribution in this chapter is several-fold. First, this study is the first, to the best of our knowledge, to report the "perplexing" transient CHOKe queue behaviors in the aftermath of change in the UDP traffic rate. Second, for any given UDP traffic rate, we derive the queue parameters that characterize the spatial properties of the queue in a steady state. Third, by leveraging the queue parameters derived and abstracting the UDP rate change by a factor, both the evolution and the extreme points of UDP throughput in transient regime are analytically formulated for any arbitrary UDP rate change. Last but not least, we obtain generic plots that succinctly summarize both transient and steady state UDP throughput behaviors. Extensive simulations confirm the validity of the theoretical results.

5.2. CHOKe Steady State Models and Properties

Since the transient regime is a transition between two steady states, some background on steady state models and properties of the CHOKe queue is necessary.

5.2.1. Steady State Models

Generally, two types of steady state CHOKe models are relevant: (1) the *overall loss model* presented in Chapter 4 (see [98, 27] for special cases of this model), and (2) the *spatial distribution model* [96]. Both models assume that the CHOKe dropping and RED-based dropping are reversed for analytic simplicity, as depicted in Fig. 4.4. The only independent parameter in both models is the UDP flow arrival rate, x_0.[2] A change in x_0 causes a departure from current UDP utilization and kicks start the transient regime.

Overall Loss Model (OLM)

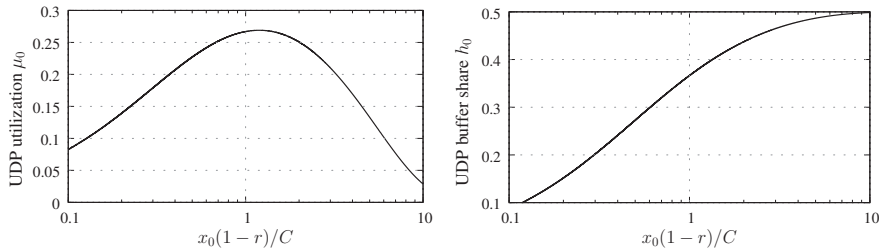
The OLM model derives the steady-state UDP utilization μ_0 and buffer shares h_0 by first deriving the flow loss probability incurred by both CHOKe and RED parts of the CHOKe queue, see the gCHOKe model in Chapter 4. For CHOKe with single input UDP flow, we substitute $\nu = 1$ and `maxcomp` $m = 1$ into the model summarized in Table 4.2. We obtain nonlinear numerical equations between UDP link utilization μ_0, UDP buffer share h_0 and UDP input rate x_0 as formulated by (5.1), (5.2) and graphically demonstrated by Fig. 5.3 (also Fig. 4.9).

$$\mu_0 = \frac{\ln\left(\frac{1-h_0}{1-2h_0}\right)}{\left(\frac{1-h_0}{1-2h_0}\right) + \ln\left(\frac{1-h_0}{1-2h_0}\right)} \tag{5.1}$$

$$\frac{x_0(1-r)}{C} = \frac{\mu_0}{1-2h_0} \tag{5.2}$$

A glance at Fig. 5.1 shows that the steady state simulation results (i.e., the areas outside the shade) match the theoretical μ_0 values plotted in Fig. 5.3(a).

[2] For our model in Chapter 4, the number of UDP flows ν is another independent parameter.



(a) UDP utilization μ_0.

(b) UDP buffer share h_0.

Figure 5.3.: Steady-state UDP utilization μ_0 and buffer share h_0 under CHOKe.

Spatial Distribution Model (SDM)

Tang, Wang and Low developed a novel ordinary differential equation model [96] of the CHOKe queue in steady state. The model captures the spatial distribution of flows in a CHOKe queue using queue properties at the tail and head as boundary conditions. This spatial distribution includes the packet velocity and the probability of finding a flow packet at a position in the queue. An important concept introduced is *thinning* which refers to the decaying of UDP packet velocity as the packet moves along towards the queue head. This steady state model, and the notion of thinning, are useful for studying the transient properties as well. We present the basic model, and provide pertinent assumptions that enable us to extend the model to the transient regime in Sec. 5.4.

5.2.2. Summary of Queue Properties

The two analytic models of CHOKe reveal the following interesting and peculiar steady-state properties of the queue. It is also possible to infer or imply some of these properties from the plots in Fig. 5.3 or Fig. 4.9.

- *Limits*: An unresponsive UDP flow cannot exceed certain limits in both buffer and link bandwidth shares. The maximum UDP bandwidth share is $(e + 1)^{-1} = 26.9\%$ of link capacity, and the maximum buffer share is 50%.
- *Asymptotic property*: As the UDP rate increases without bound, its buffer share can asymptotically reach 50% and its queueing delay decreases, but its link utilization drops to *zero*.
- *Spatial distribution*: The spatial packet distribution in the queue can be highly nonuniform [96]. The probability of finding a packet belonging to a high rate flow in the queue diminishes dramatically as we move towards the head of the queue. Correspondingly, the flow distribution in the queue is skewed with

most packets of high rate flows found closer to the queue tail while packets of low rates are found closer to the queue head. See the last observation on Page 57.

5.3. System Model and Notation

We use the system setup of Chapter 4 together with the same assumptions and notations. The studied setup is depicted in Fig. 4.3, where N rate-adaptive similar TCP flows share a link with a single unresponsive / aggressive UDP flow. Due to the additional notations we adopted in this chapter and for the sake of clarity and self-containment, we present the assumptions and notations fully in this section.

As before, flows are indexed from 0, $\cdots$, N, where 0 denotes the UDP flow. Since TCP flows are assumed to be similar, 1 hereafter denotes a typical TCP flow. As in Chapter 4, the steady-state backlog size in packets is denoted by b. We assume large N and b.

The full notation is summarized in Table 5.1. The key performance metric is the flow (link) utilization denoted by $\mu_i,\ i \in [0, 1]$. Given UDP link utilization μ_0 and link capacity C, the throughput of all flows can be computed.

A queue parameter that is not indexed with a control variable, say time t, designates the value of the parameter in the steady state. Otherwise, the parameter is dependent on that control variable. For example, μ_0 designates UDP utilization in a steady state, while $\mu_0(t)$ is changing with t, more likely during a transient regime.

Table 5.1.: Notation

Symbol	Description
N	number of TCP flows
r	congestion(RED)-based dropping probability, or ambient drop probability (common to all flows)
x_i	source rate of flow i
μ_i	link utilization of flow i
b_i	amount of flow i packets in buffer
b	total backlog $b = \sum_{i=0}^{N} b_i$ in packets
h_i	the ratio b_i/b (matching probability)
$y \in [0, b]$	position in queue
$v(y)$	packet velocity at y
$\tau(y)$	queueing delay to reach at y
$\rho_i(y)$	prob. of finding flow $i \in [0, 1]$ at position y

As in (4.6), for a TCP flow, the packet matching in CHOKe is approximated by,

$$h_1 = \frac{b_1}{b} = \frac{b_1}{b_0 + Nb_1} \leq \frac{1}{N} \approx 0 \tag{5.3}$$

5.4. Modeling the Transient Regime

Before delving into the transient model, we explain the transient queue behaviors in Sec. 5.4.1 using an argument which we call the *rate conservation argument*. This argument also provides an insight to extending the SDM to transient regimes. The SDM extension is then treated in Sec. 5.4.2.

5.4.1. Rate Conservation Argument

Before a UDP packet can be admitted into a CHOKe queue, it must survive both the RED and the CHOKe based dropping. The probability of packet admission into queue is then $(1 - r)(1 - h_0(t))$. Once in the queue, the UDP packets can still be lost. This is because incoming packets that evade RED-based dropping (with probability $1 - r$) may trigger flow matching (with probability $h_0(t)$) and cause dropping of the matched packets. In addition, UDP packets can also leave the queue due to transmission with rate $\mu_0(t)C$. Summarizing, we get a system invariant that captures the rate of change in UDP buffered packets as follows:

$$\begin{aligned}\frac{db_0(t)}{dt} &= x_0(t)(1-r)(1-h_0(t)) - x_0(t)(1-r)h_0(t) - \mu_0(t)C \\ &= x_0(t)(1-r)(1-2h_0(t)) - \mu_0(t)C \end{aligned} \tag{5.4}$$

Let us call (5.4) the *rate conservation argument*. That is, the rate of change in UDP buffer occupancy is the difference between the flow queueing rate $x_0(t)(1 - r)(1 - h_0(t))$ and the outgoing rate. The outgoing rate in turn is the sum of the departure/transmission rate given by $\mu_0(t)C$ and the leaking rate given by $x_0(t)(1-r)h_0(t)$. Here the leaking rate denotes the rate with which a queued UDP packet matches the incoming packet and is consequently dropped.

The corresponding equation for a TCP flow is, as $h_1 = 0$ (see (5.3)),

$$\frac{db_1(t)}{dt} = x_1(t)(1-r) - \mu_1(t)C, \tag{5.5}$$

where, since the link is fully used,

$$\mu_1(t) = \frac{1-\mu_0(t)}{N}. \tag{5.6}$$

Since, trivially, $b(t) = b_0(t) + Nb_1(t)$, we get

$$\frac{db(t)}{dt} = \frac{db_0(t)}{dt} + N\frac{db_1(t)}{dt}. \tag{5.7}$$

Remark During a stable / steady state $db_i(t)/dt \approx 0,\ i \in (0, 1)$ (see Fig. 5.4).

Now, let us assume an abrupt change in UDP arrival rate x_0 and note the following in the immediate aftermath: *TCP flows react slowly in response to the sudden change in UDP arrival rate. Notably, TCP flows react in an RTT timescale, but the transient behavior lasts for sub-RTT scales.* Hence, during the transient phase,

$$\frac{db_1(t)}{dt} \approx 0 \quad \Rightarrow \quad \frac{db(t)}{dt} \approx \frac{db_0(t)}{dt} \tag{5.8}$$

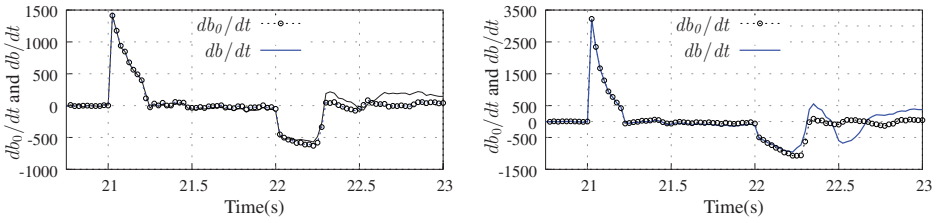
Fig. 5.4 plots the simulation results for the two experiments of *Example 1* in Sec. 5.1.1. The figures confirm the argument above and Eq. (5.8). As can be seen, during the transient parts of the simulations, the changes in total buffer occupancy b are almost solely due to changes in UDP buffer occupancy b_0.

We warn the reader that (5.8) may not hold outside the transient regime. For instance, when the UDP flow arrival rate plummets at $t = 22$s, TCP flows respond by increasing their sending rates after around a round-trip delay (see especially Fig. 5.4(b)). Subsequently, $db_1/dt \neq 0$, but $db_0/dt \approx 0$. From (5.7), $db/dt \approx N db_1/dt$. That means, db/dt swings from db_0/dt during the transient phase to $N db_1/dt$ following the transient phase. However, since the TCP flows are largely in a congestion avoidance phase, the rise in $N db_1/dt$ is not as significant as that of db_0/dt in the transient phase. After absorbing TCP bursts for a while (say, a few round-trip cycles), the queue eventually settles to a new steady state determined by the new UDP arrival rate, and then $db_i/dt \approx 0,\ i \in (0, 1)$ once again.

Remark Combining (5.5) and (5.8), we conclude that TCP packet arrival rate to the CHOKe queue matches its transmission rate during the transient regime.

Now we are in a position to explain the transient behaviors shaded in Fig. 5.1 and Fig. 5.2 from the perspective of the rate conservation argument. Rearranging (5.4), we get

$$\mu_0(t) = \frac{x_0(t)(1-r)}{C}(1 - 2h_0(t)) - \frac{1}{C}\frac{db_0(t)}{dt} \tag{5.9}$$



(a) Experiment 1 ($0.5C/2C$). (b) Experiment 2 ($0.25C/3C$).

Figure 5.4.: Transient regime: $db/dt \approx db_0/dt$ and $db_1/dt \approx 0$. Steady state: $db/dt \approx 0$, $db_0/dt \approx 0$, $db_1/dt \approx 0$.

Interestingly, (5.9) captures all pertinent behaviors of the system. In the steady state, $db_0(t)/dt \approx 0$, and

$$\mu_0 = \frac{x_0(1-r)}{C}(1-2h_0) \tag{5.10}$$

which is (5.2). We have reproduced one of the key equations in the OLM steady state model depicted in Fig. 5.3.

Now, let us return to the transient behaviors. We explain only the dips in μ_0 shown in Fig. 5.1 but similar arguments follow for the peaks as well. An abrupt injection of UDP rate x_0 at $t = 21$s rapidly ramps up the second term on the right hand side (r.h.s.) of (5.9) (see also Fig. 5.4). Despite the rapid rise of $x_0(1-r)/C$, its contribution to $\mu_0(t)$ is approximately counteracted by a corresponding rise in $h_0(t) = b_0(t)/b(t)$ (see Fig. 5.5 where $h_0(t) \to 45\%$). Therefore,

Remark *The transient UDP utilization in CHOKe $\mu_0(t)$ is mainly influenced inversely by the rate $db_0(t)/dt$.*

See the contrast between Fig. 5.4 and Fig. 5.1. Specifically, focusing on Fig. 5.1 and using measurement intervals of 10ms, we observed the following : when x_0 increases 12-fold at $t = 21$s, $db_0/dt \to 3500$ (as shown in Fig. 5.4(b)) but the utilization $\mu_0 \to 3.75\%$. Conversely, when x_0 slackens by a factor of 12 at $t = 22$s, the db_0/dt falls to 1000 below 0 (as shown in Fig. 5.4(b)) but the UDP link bandwidth share soars to 56.5%.

While the rate conservation argument explains the transient queue dynamics qualitatively, due to several dynamically changing parameters ($h_0(t)$, $b_0(t)$, $b(t)$) in the transient regime, it is difficult to derive quantitative μ_0 results directly from (5.9). Nevertheless, some of the insights we gained prove to be useful for extending the SDM model to the transient regime, as we shall soon see.

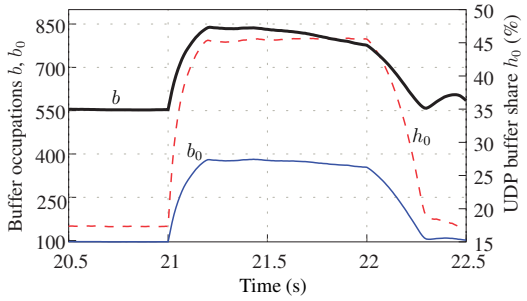


Figure 5.5.: At $t = 21$, UDP buffer share h_0 jumps radically, nullifying the impact on μ_0 of the sudden change in x_0.

5.4.2. Modified Spatial Distribution Model

This section is dedicated to the development of the SDM in concert with the transient regime. A schematic diagram of this model is illustrated in Fig. 5.6.

5.4.2.1. Model parameters

The SDM can be described by a few key parameters (see Fig. 5.6 and Table 5.1). The parameters are queue positions/points/slots $y \in [0, b]$, the packet velocity $v(y)$ at y, the probability $\rho_i(y)$ of finding a flow i packet at y, and the queueing delay $\tau(y)$ for the packet at the tail to arrive at slot y. Queue position y is indexed from tail to head as $\{0, \cdots, b\}$. The packet velocity $v(y)$ is the speed with which packets move towards the head of the queue, and is defined as:

$$v(y) = dy/dt. \tag{5.11}$$

The packet velocity at queue tail $v(0)$ is simply the full queueing rate $\sum_{i=0}^{N} x_i(1-r)(1-h_i)$ (see Fig. 4.4). At queue head, however, $v(y)$ is merely the link capacity, i.e., $v(b) = C$. The packet velocity $v(y)$ is related to the queueing delay $\tau(y)$ accumulated in going from tail $y = 0$ to slot y as follows,

$$dt = dy/v(y) \qquad \Rightarrow \qquad \tau(y) = \int_0^y \frac{1}{v(s)} ds \tag{5.12}$$

where the equation on the left side is obtained from (5.11).

Of course, the full queueing delay is $\tau(b)$. Alternatively, $\tau(b)$ can be derived using queueing principles. The rate of departure of TCP packets is $C(1-\mu_0)$, and

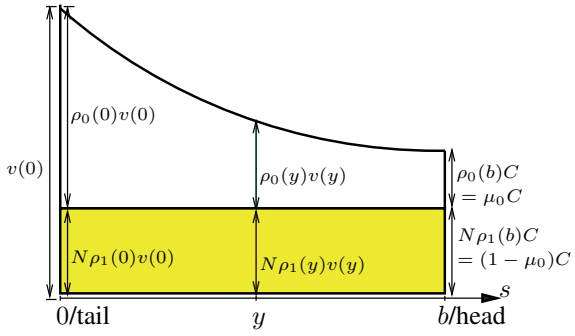


Figure 5.6.: Schematic diagram: Decay of UDP velocity $\rho_0(y)v(y)$ in queue. Look at the similarity to Fig. 5.7(b) when $x_0 = 10C$.

average number of TCP packets in queue is given by $b(1-h_0)$. As far as the TCP flows are concerned, the model is a non-leaky queue (since $h_1 = 0$). Therefore, we apply Little's law to obtain,

$$\tau(b) = \frac{b(1-h_0)}{C(1-\mu_0)}. \tag{5.13}$$

Another useful spatial parameter is $\rho_i(y)$—the probability of finding a flow i packet at slot y. Trivially,

$$\rho_0(y) + N\rho_1(y) = 1 \qquad y \in [0, b] \tag{5.14}$$

Here, $\rho_i(y)$ is closely related to the packet velocity $v(y)$: It quantifies the fraction of flow i's packet velocity at y to the total packet velocity $v(y)$. For instance, at queue head where $y = b$, $\rho_i(b) = \mu_i$, i.e., the probability is simply the flow link utilization.

Summarizing the two important parameters $\rho_i(y)$ and $v(y)$ at queue tail and head, the following boundary conditions apply, which are also illustrated in Fig. 5.6. Here, we ignore TCP flow matching since from (5.3) $h_1 \approx 0$.

$$\begin{aligned} v(0) &= x_0(1-r)(1-h_0) + Nx_1(1-r)(1-h_1) \\ &\approx x_0(1-r)(1-h_0) + Nx_1(1-r) \end{aligned} \tag{5.15}$$

$$\rho_0(0) = \frac{x_0(1-h_0)(1-r)}{[x_0(1-h_0)+Nx_1](1-r)} \approx \frac{x_0(1-h_0)}{x_0(1-h_0)+Nx_1} \tag{5.16}$$

$$\rho_1(0) = \frac{x_1}{x_0(1-h_0)+Nx_1} \tag{5.17}$$

$$v(b) = C, \qquad \rho_0(b) = \mu_0, \qquad \rho_1(b) = \mu_1 = \frac{1-\mu_0}{N} \tag{5.18}$$

Note that $\rho_1(b)$ in (5.18) is the same as (5.6). As mentioned in Sec. 5.4.1, TCP transmission rates do not change during the transient regime, resulting in constant TCP packet velocities throughout the queue. In a congested CHOKe queue with high UDP rate x_0, however, the total packet velocity $v(y)$ is continuously decreasing because UDP arrivals trigger packet drops through flow matching. In fluid terms, we say the UDP fluid gets *thinned* as it moves along the queue. We formalize the notion of thinning and use it to derive the slot parameters next.

5.4.2.2. Ordinary Differential Equation model

The UDP portion of the packet velocity at the tail ($y = 0$) is given by $\rho_0(0)v(0) = x_0(1-h_0)(1-r)$ and the amount of UDP fluid in small time dt at the tail by

$\rho_0(0)v(0)dt$. The corresponding values at y are $\rho_0(y)v(y)$ and $\rho_0(y)v(y)dt$, respectively. Traveling from queue tail to y takes $\tau(y)$ during which time $x_0(1-r)\tau(y)$ new packets would arrive to the queue. Each arrival triggers a flow matching trial and drops the small volume of fluid with success probability $1/b$. The probability that the UDP volume escapes matchings by all arrivals is $(1-1/b)^{x_0(1-r)\tau(y)}$. The UDP packet velocity at y is therefore thinned or weakened as,

$$\rho_0(y)v(y) = \rho_0(0)v(0)(1-1/b)^{x_0(1-r)\tau(y)} \tag{5.19}$$

Note the similarity to (4.4), where $m = 1$ for CHOKe. On the other hand, there is no thinning for TCP and hence a constant TCP packet velocity throughout.

$$N\rho_1(y)v(y) = N\rho_1(0)v(0) = (1-\mu_0)C \tag{5.20}$$

During the transient regime, (5.20) is still valid due to the slow reaction of TCP congestion control, as discussed in Sec. 5.4.1.

Rearranging (5.20) and using (5.14), we get

$$v(y) = \frac{\rho_1(0)v(0)}{\rho_1(y)} = \frac{(1-\mu_0)C}{N\rho_1(y)} = \frac{(1-\mu_0)C}{1-\rho_0(y)} \tag{5.21}$$

Define parameters a and β as follows, which will often be used throughout the rest of the chapter:

$$a := \frac{1-\rho_0(0)}{\rho_0(0)}$$
$$\beta := \ln(1-1/b)$$

Using (5.14), (5.15), (5.16) and (5.20), a can equivalently be rewritten as,

$$a = \frac{1-\rho_0(0)}{\rho_0(0)} = \frac{N\rho_1(0)}{\rho_0(0)} = \frac{N\rho_1(0)}{\rho_0(0)}\frac{v(0)}{v(0)} = \frac{(1-\mu_0)C}{x_0(1-r)(1-h_0)}.$$

Taking logarithm of key equation (5.19) first and then differentiation w.r.t. y, we get

$$\begin{aligned} \ln(\rho_0(y)v(y)) &= \ln(\rho_0(0)v(0)) + x_0(1-r)\beta\tau(y) \\ \frac{\rho_0^{'}(y)}{\rho_0(y)} + \frac{v^{'}(y)}{v(y)} &= 0 + \frac{x_0(1-r)\beta}{v(y)} \end{aligned} \tag{5.22}$$

where $\tau'(y)=1/v(y)$ from (5.12).

Applying (5.21) for $v(y)$ and $v^{'}(y)$ and inserting into l.h.s. of (5.22), the following ordinary differential equation (ODE) is obtained.

$$\frac{\rho_0^{'}(y)}{\rho_0(y)} + \frac{\rho_0^{'}(y)}{1-\rho_0(y)} = \frac{x_0(1-r)\beta}{v(y)} \tag{5.23}$$

Eq. (5.23) establishes a foundation for the analysis in the remaining part of the chapter.

First, solving $\rho_0(y)$ from ODE (5.23) (see the Appendix for the proof), we have the following relation between $\rho_0(y)$ and $\tau(y)$:

Lemma 5.4.1

$$\rho_0(y) = \frac{e^{x_0(1-r)\beta\tau(y)}}{a + e^{x_0(1-r)\beta\tau(y)}}. \tag{5.24}$$

In addition, substituting (5.21) for $v(y)$ in (5.23) gives,

$$\frac{\rho_0^{'}(y)}{\rho_0(y)} + \frac{\rho_0^{'}(y)}{1-\rho_0(y)} = \frac{x_0(1-r)\beta}{(1-\mu_0)C}(1-\rho_0(y)).$$

Dividing by $(1-\rho_0(y))$, we obtain

$$\frac{\rho_0^{'}(y)}{\rho_0(y)(1-\rho_0(y))} + \frac{\rho_0^{'}(y)}{(1-\rho_0(y))^2} = \frac{x_0(1-r)\beta}{(1-\mu_0)C}$$

Upon integrating w.r.t. y, we get

$$\int_0^y \frac{x_0(1-r)\beta}{(1-\mu_0)C}ds = \int_0^y \frac{\rho_0^{'}(s)}{\rho_0(s)(1-\rho_0(s))}ds + \int_0^y \frac{\rho_0^{'}(s)}{(1-\rho_0(s))^2}ds$$

$$\frac{x_0(1-r)\beta}{(1-\mu_0)C}s\Big|_0^y = \ln\left[\frac{\rho_0(s)}{1-\rho_0(s)}\right]\Big|_0^y + \frac{1}{1-\rho_0(s)}\Big|_0^y$$

Let $K = \frac{x_0(1-r)\beta}{(1-\mu_0)C}$. We then get:

$$Ky = \ln\left[\frac{\rho_0(y)}{1-\rho_0(y)}\frac{1-\rho_0(0)}{\rho_0(0)}\right] + \frac{1}{1-\rho_0(y)} - \frac{1}{1-\rho_0(0)} \tag{5.25}$$

from which, the following relation between y and $\rho_o(y)$ is easily established:

Lemma 5.4.2

$$y = \frac{1}{K}\ln\left[\frac{a\rho_0(y)}{1-\rho_0(y)}\right] + \frac{1}{K}\frac{\rho_0(y)-\rho_0(0)}{(1-\rho_0(y))(1-\rho_0(0))} \tag{5.26}$$

where $K = \dfrac{x_0(1-r)\beta}{(1-\mu_0)C}$ *and as above,* $a := \dfrac{1-\rho_0(0)}{\rho_0(0)} = \dfrac{(1-\mu_0)C}{x_0(1-r)(1-h_0)}$.

Remark Lemma 5.4.1 and Lemma 5.4.2 are crucial, through which, the slot and its associated parameters are inter-related as defined by $\rho_0(y)$ (5.24), y (5.26), $v(y)$ (5.21) and $\tau(y)$ (A.4). However, from the model in [96], it is difficult to obtain for each slot y corresponding explicit expressions of the spatial parameters $\rho_0(y)$, $v(y)$ and $\tau(y)$, hence limiting the application of results in [96] to the transient analysis. This is because $\rho_0(y)$ and $v(y)$ were given in terms of queueing delay $\tau(y)$, but no explicit relations were afforded between queue slot y and anyone of the triplets $(v(y), \rho_0(y), \tau(y))$ in [96].

Remark The analytical principles leading to Lemma 5.4.1 and Lemma 5.4.2 also apply to the transient regime, making later transient analysis possible.

5.4.2.3. Properties of Queue Dynamics

This section briefly outlines properties of queue dynamics $\rho_0(y)$, $v(y)$ and $\tau(y)$. The detailed proofs are in the Appendix.

We start with $\rho_0(y)$. To prove the properties of $\rho_0(y)$ in Lemma 5.4.4, we need the following intermediate result.

Lemma 5.4.3

$$1 \leq \frac{1-\mu_0}{1-h_0} \leq 2$$

The following properties hold for $\rho_0(y)$, and follow from (5.24) and Lemma 5.4.3.

Lemma 5.4.4 *Given $x_0 > 0$,*

(i) If $x_0 \leq C/2$, $\rho_0(y)$ is strictly convex decreasing.

(ii) Otherwise, $\rho_0(y)$ is concave decreasing from $\rho_0(0)$ to ρ_0^*, and convex decreasing from ρ_0^* to $\rho_0(b)$ where the critical point $(y^*,\ \rho_0^*)$ is given by

$$\rho_0^* = \rho_0(y^*) = \frac{1}{3}$$
$$y^* = \frac{1}{K}\ln\left(\frac{a}{2}\right) + \frac{1}{K}\frac{1-3\rho_0(0)}{2(1-\rho_0(0))}.$$

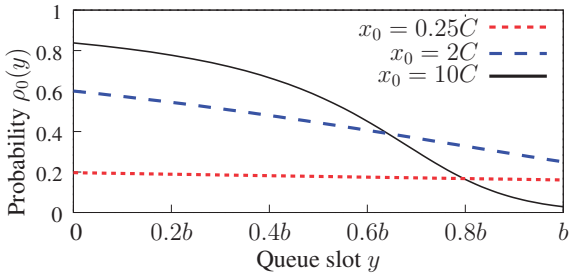
For $v(y)$, applying (5.24) to (5.21), the following property can be verified:

Lemma 5.4.5 *Given $x_0 > 0$, and boundary values $v(0)$ and $v(b)$ defined respectively by* (5.15) *and* (5.18)*, the packet velocity $v(y)$ is convex decreasing throughout the queue.*

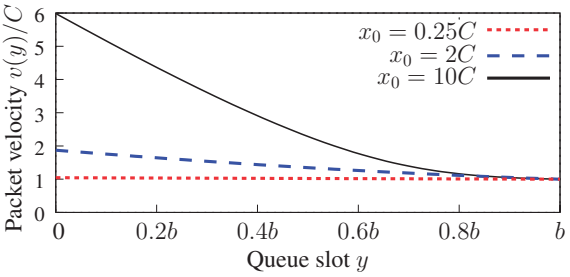
For $\tau(y)$, the following property holds.

Lemma 5.4.6 *Given $x_0 > 0$, $\tau(0) = 0$ and $\tau(b)$ defined by (5.13), queueing delay $\tau(y)$ is strictly convex increasing.*

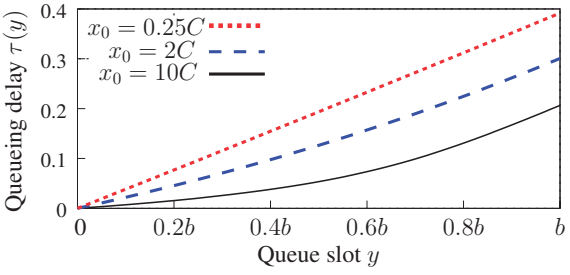
To visualize the queue dynamics, Fig. 5.7 plots for each queue position y the probabilities $\rho_0(y)$, the packet velocities $v(y)$ and the queueing delays $\tau(y)$. For low and moderate rate x_0, the spatial properties $v(y)$ and $\rho_0(y)$ look uniform in queue,



(a) UDP packet probability vs queue slot.



(b) Packet velocity at queue slot.



(c) Queuing delay to reach at a queue slot.

Figure 5.7.: Spatial characteristics of a CHOKe queue under different intensities of input UDP arrival rates.

and the queueing delay $\tau(y)$ is approximately linearly rising, like in a regular non-leaky queue. With increasing rate x_0, however, the spatial distribution becomes increasingly asymmetrical in queue, and most of the UDP packets are piled up closer to the tail. Additionally, the packet velocity sharply decreases to C as we move towards the head of queue. In such cases, since the packet velocities are nonuniform in queue, the queueing delays are also nonuniform. When $x_0 \to \infty$, the queueing delay becomes $\frac{1}{2}\frac{b}{C}$—fictitiously half of the queueing delay possible in a non-leaky queue with the same backlog size b.[3]

5.4.3. Analysis on the Transient Behavior

The transient behavior is a transition between two stable queue states excited by a change in UDP arrival rate. Without loss of generality, let us assume that the UDP arrival rate changes from x_0 to x_{02} at time $t = 0$. At that instant, the snapshot of the queue exhibits the steady state characteristics defined by x_0 (see Fig. 5.6). We are interested in transient behaviors that surface within a time of $\tau(b)$ (the full queueing delay) after rate change. Thereafter, we assume the queue enters the steady state defined by the new rate x_{02}. We shall focus on queue properties during the transition regime $[0, \tau(b)]$.

While Lemma 5.4.1 and Lemma 5.4.2 lay a foundation for the analysis, we still need to make assumptions to proceed. We *boldly* assume during $[0, \tau(b)]$ that,

(1) the buffer occupancy remains constant at b, and

(2) TCP arrival rates remain the same.

Remark It is worth highlighting that while Assumption (2) is valid or accurate based on our observation of TCP reactions in Sec. 5.4.1, Assumption (1) may be strong (see Fig. 5.5). Incidentally, this assumption may be the cause of approximation errors in the analysis. Nevertheless, results based on this assumption are very satisfactory.

We are now ready to conduct the analysis. Consider at time $t = 0$ the UDP fluid at position y. It has a velocity $\rho_0(y)v(y)$, and a remaining age in the queue of $\tau(b) - \tau(y)$. Owing to FIFO, the UDP flow transmission rate at time $\tau(b) - \tau(y)$ is due to the transmission of this UDP fluid. By transmission time $\tau(b) - \tau(y)$, the UDP velocity has been thinned according to the new rate x_{02} as:

$$\rho_0(y)v(y)(1 - 1/b)^{x_{02}(1-r)(\tau(b)-\tau(y))}. \tag{5.27}$$

Remark (5.27) is similar to (5.19). Unlike (5.19), however, there are two distinct parts of thinning for the UDP fluid located at slot $y \in [0, b]$ at $t = 0$. First, in

[3]Queueing delay does not precisely reduce by half because the backlog size b varies with x_0.

going from tail to y, the thinning is according to x_0, and the duration of thinning is the queueing delay so far, i.e., $\tau(y)$. This part is reflected in (5.27) by $\rho_0(y)v(y)$. Second, for the remaining duration $\tau(b)-\tau(y)$, the thinning is due to the new rate x_{02}. Overall thinning, before eventual transmission, of the UDP fluid found at slot y at $t=0$ is then,

$$\rho_0(0)v(0)(1-1/b)^{x_0(1-r)\tau(y)}(1-1/b)^{x_{02}(1-r)(\tau(b)-\tau(y))}.$$

Above, we use the same RED / congestion-based drop probability r even when the UDP arrival rate changes. In fact, our extensive simulations show that r is often insignificant and can be ignored altogether (recall similar observations in Chapter 4, especially Sec. 4.5.1). CHOKe's excessive flow dropping keeps the average queue size *avg* in check, and this in turn lowers the r in comparison to that in plain RED. Hereafter we choose to ignore r. That is,

$$r \approx 0. \tag{5.28}$$

With Assumption (2), the total packet velocity v at time $\tau(b)-\tau(y)$ is the sum:

$$\begin{aligned} v(\tau(b)-\tau(y)) = {} & \rho_0(y)v(y)(1-1/b)^{x_{02}(\tau(b)-\tau(y))} \\ & + (1-\mu_0)C \end{aligned} \tag{5.29}$$

where the second term represents the velocity contributed by the TCP flows, given by (5.20).

The instantaneous UDP link utilization μ_0 follows from (5.27) and (5.29) simply as,

$$\mu_0(\tau(b)-\tau(y)) = \frac{\rho_0(y)v(y)(1-1/b)^{x_{02}(\tau(b)-\tau(y))}}{v(\tau(b)-\tau(y))} \tag{5.30}$$

Note that, from (5.21), which holds for both the steady-state and the transient regime, we can express the UDP packet velocity, $\rho_0(y)v(y)$, as

$$\rho_0(y)v(y) = \frac{\rho_0(y)}{1-\rho_0(y)}(1-\mu_0)C \tag{5.31}$$

with which, we further obtain

$$\begin{aligned} \mu_0(\tau(b)-\tau(y)) &= \left[1+\frac{1-\rho_0(y)}{\rho_0(y)}\left(1-\frac{1}{b}\right)^{-x_{02}(\tau(b)-\tau(y))}\right]^{-1} \\ &= \left[1+\frac{1-\rho_0(y)}{\rho_0(y)}e^{-x_{02}\beta(\tau(b)-\tau(y))}\right]^{-1} && \text{(5.32a)} \\ &= \left[1+ae^{-x_{02}\beta\tau(b)+\beta\tau(y)(x_{02}-x_0)}\right]^{-1} && \text{(5.32b)} \end{aligned}$$

In obtaining the reduced forms (5.32a),(5.32b), we used (5.31) and (5.24) respectively.

(5.32b) captures the evolution of UDP utilization during the transient regime. We summarize it in the following lemma.

Lemma 5.4.7 *Assume at $t = 0$, UDP arrival rate changes from x_0 to x_{02}. The UDP link utilization at time $\Delta T \in [0, \tau(b)]$ is given by,*

$$\mu_0(\Delta T) = \left[1 + ae^{-x_{02}\beta\tau(b)+\beta(\tau(b)-\Delta T)(x_{02}-x_0)}\right]^{-1}.$$

where, b is the backlog size at $t = 0$ and $\tau(b)$ is given by (5.13).

It is trivial to see that $\mu_0(0) = \mu_0$. In addition, it is easy to prove that when $x_{02} = x_0$, $\mu_0(\Delta T) = \mu_0$ for $\forall \Delta T \in [0, \tau(b)]$. The proof is similar to the proof of Lemma 5.4.9.

Theorem 5.4.8 *Assume a CHOKe queue characterized by steady state UDP probabilities $\rho_0(y)$, $y \in [0, b]$ and input UDP rate x_0. Further assume the UDP rate changes to $x_{02} \geq 0$ at $t = 0$.*

(a) The transient UDP utilization is upper bounded by $\rho_0(0)$.

(b) This upper bound can be achieved when $x_{02} = 0$ and at time $t = \tau(b)$.

Proof Since $\tau(b) \geq \tau(y)$ (see Lemma 5.4.6) and $\beta < 0$ in (5.32a), we have $\exp(-x_{02}\beta(\tau(b) - \tau(y))) \geq \exp(0) = 1$ and

$$\begin{aligned}\mu_0(\tau(b) - \tau(y)) &\leq \left[1 + \frac{1 - \rho_0(y)}{\rho_0(y)}\right]^{-1} \\ &\leq \rho_0(y) \leq \rho_0(0) \end{aligned} \tag{5.33}$$

In (5.33), we used the property that $\rho_0(y)$ is a decreasing function (see Lemma 5.4.4) to state that $\mu_0(\tau(b) - \tau(y)) \leq \rho_0(0)$. When $x_{02} = 0$ in (5.32a), $\mu_0(\tau(b) - \tau(y)) = \rho_0(y)$. See Fig. 5.8 for the relationship between $\rho_0(y)$ in queue and transient UDP rate. ∎

Theorem 5.4.8 states that if the UDP flow stops ($x_{02} = 0$), it will attain exactly the utilizations $\rho_0(y)$ in reverse order of time shown in Fig. 5.6 (resulting in the 'mirror reflection' Fig. 5.8). That means, the steady-state probabilities $\rho_0(y)$ associated with the "old" UDP input rate x_0 successively turn out as transient utilizations when the flow stops. Therefore, the transient utilizations increase from $\rho_0(b) = \mu_0$ at $t = 0$ to $\rho_0(0)$ at $t = \tau(b)$.

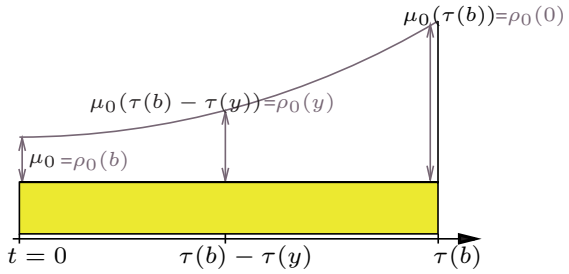


Figure 5.8.: Relationship between steady-state $\rho_0(y)$ and transient UDP utilization $\mu_0(t)$ when the flow stops, i.e., $x_{02} = 0$.

Theorem 5.4.8 vindicates our choice to leverage steady-state CHOKe model for understanding the transient behavior.

So far we have discussed the two special cases when $x_{02} = x_0$ (no change) and $x_{02} = 0$. Now consider a scenario where $x_{02} \notin \{0, x_0\}$, in particular $x_{02} \to \infty$. It is plausible that when x_{02} is very high, the exponential terms in (5.32a), and in Lemma 5.4.7, may become so large that the utilization may quickly plunge to very low values. From this observation, we remark that the *UDP utilization in the transient phase moves in the opposite direction to the change of UDP input rate that triggers the phase.* The is similar to the observation in Sec. 5.4.1 that the UDP utilization moves in a direction opposite to that of db_0/dt.

We now generalize our findings and obtain the extreme values. Note that from Lemma 5.4.7, $\mu_0(\Delta T)$ is decreasing or increasing with ΔT, depending on whether x_{02} is greater than x_0 or not, so the extreme is obtained when $\Delta T = \tau(b)$. In other words, the extreme (lowest or largest) values occur at a time of $\tau(b)$ after the rate change. That means, the last packet of the old rate x_0 is transmitted with the extreme utilization $\mu_0^* = \mu_0(\tau(b))$. The next theorem gives the maximum or minimum value. First, we start with Lemma 5.4.9 which captures the special case when $\alpha = 1$.

Lemma 5.4.9 *For $\alpha = 1$, $\mu_0(\tau(b)) = \mu_0$.*

Theorem 5.4.10 *Assume the current steady-state utilization μ_0 when the input UDP rate is x_0. If $x_{02} = \alpha x_0$ at $t = 0$, $\alpha \in [0, \infty)$, then the extreme (minimum / maximum) UDP utilization during the transient regime is given by*

$$\mu_0(\tau(b)) = \left[1 + a\left(\frac{1-\mu_0}{a\mu_0}\right)^{\alpha}\right]^{-1}. \tag{5.34}$$

See the Appendix for proofs of Theorem 5.4.10 and Lemma 5.4.9.

Corollary 5.4.11 *For $\alpha = 0$, or when the UDP flow stops, $\mu_0(\tau(b)) = \rho_0(0)$.*

Theorem 5.4.10 is a key finding in this work and can be visualized using Fig. 5.9. Note that the figure illustrates both steady-state and (extreme) transient UDP utilizations for the selected UDP arrival rates. When $\alpha = 1$, by Lemma 5.4.9, the values shown are the steady-state utilization μ_0 for the given UDP arrival rate x_0. When the UDP arrival rate x_0 changes (i.e., $\alpha \neq 1$), the graph shows how far the UDP utilization can go up/down when x_0 abruptly decreases/increases to x_{02}. For example, assume an initial UDP rate of $x_0 = 2C$. The utilization for this input is read from the figure at $\alpha = 1$ (also from Fig. 5.3) as $\mu_0 = 25.0\%$. When $x_{02} = 0.2C$ which corresponds to rate change by a factor of $\alpha = 0.1$, the transient utilization surges to $\mu_0(\tau(b)) = 56.5\%$. If the flow stops, by Corollary 5.4.11, the utilization instead jumps to $\rho_0(0) = 60\%$ (see also Fig. 5.7(a)). Similarly, when a flow of initial UDP arrival rate $x_0 = 3C$ stops, the transient utilizations can surge to a whopping 67% from the initial 21%.

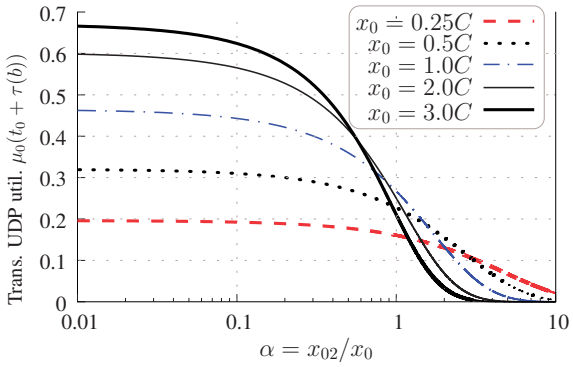


Figure 5.9.: The impact of the multiplicative factor $\alpha = x_{02}/x_0$ on extreme UDP utilizations. Five previous inputs $x_0 \in \{0.25C, 0.5C, 1C, 2C, 3C\}$ are shown.

5.5. Performance Evaluation

In this section, we validate the results using simulations performed in ns-2.34. The network setup shown in Fig. 4.3 with the following settings is used: $C = 20Mbps$ or 2500 pkt/sec, link latency $1ms$, buffer size 1000 packets, $N = 100$ TCP flows each of type SACK, RED buffer thresholds (in packets) $\min_{th} = 20$ and $\max_{th} = 1000$. Packet sizes are 1000 bytes. Flows start randomly on the interval [0,2] sec.

We conducted extensive experiments, each simulation replicated 500 times if not otherwise highlighted. The 95% confidence intervals are so small that they are not reported. We remark that in computing the simulation results, unless stated otherwise, we have used a time window of 1ms. Since C=2500 pkts/sec, this is 2.5

times more than the per packet transmission interval assumed by the model, but the error due to this disparity is small and can be ignored. Sec. 5.5.1 presents the validation of the model, and Sec. 5.5.2 presents additional simulation results.

5.5.1. Model Validation

In this section, we validate the two important results of this work: Theorem 5.4.10 and Lemma 5.4.7.

5.5.1.1. Validation of Theorem 5.4.10

In Fig. 5.10, we show for selected initial UDP arrival rates the impact of rate change by factor $\alpha \in [0.01, 10]$. As can be seen, the simulation results accurately match the model predictions. For instance, for $x_0 = 3C$ and $x_{02} = 0.03C$, or $\alpha = 0.01$, the maximum utilizations obtained by the model and simulations are 67% and 65%, respectively.

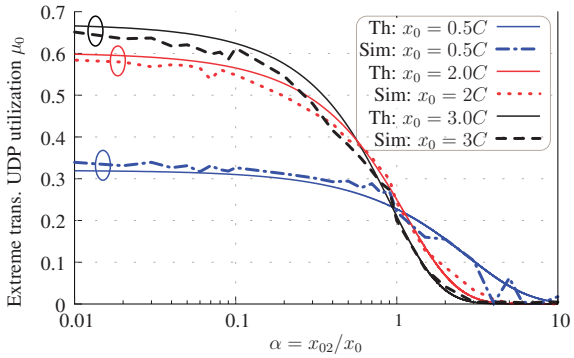
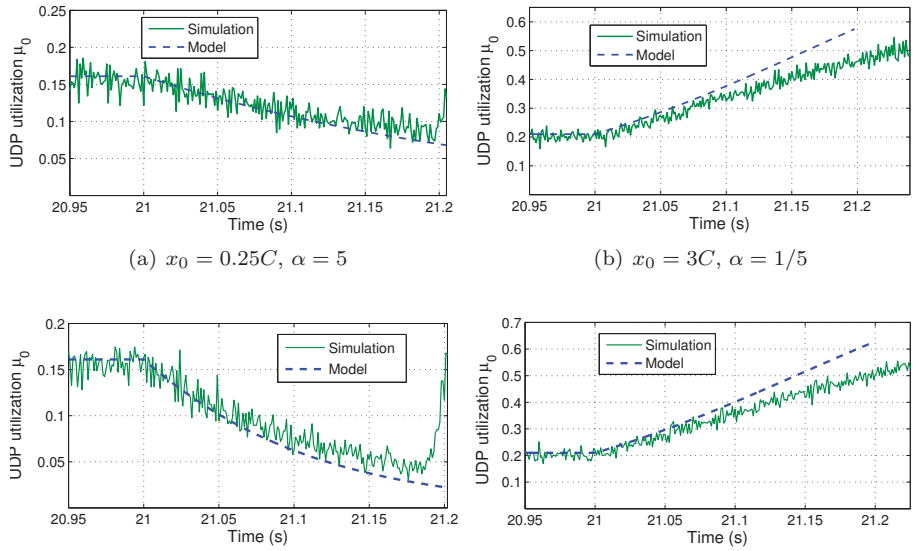


Figure 5.10.: Validation of extreme utilization stated by Th. 5.4.10.

5.5.1.2. Validation of Lemma 5.4.7

For rate factors of $\alpha = 5,\ 1/5,\ 10,\ 1/10$, Fig. 5.11 shows the evolution of transient UDP utilizations obtained through simulation and the analytical model as stated by Lemma 5.4.7. The steady state backlog size b required for the theoretical plot is taken from the steady state simulation just before the rate change.

As discussed earlier, there may be two sources of approximation errors for the theoretical results. First, the assumption of constant backlog size b during the transient phase may be strong. Second, the difference in measurement intervals.



(a) $x_0 = 0.25C$, $\alpha = 5$

(b) $x_0 = 3C$, $\alpha = 1/5$

(c) $x_0 = 0.25C$, $\alpha = 10$

(d) $x_0 = 3C$, $\alpha = 1/10$

Figure 5.11.: Validating the transient utilization Eq. (5.32b)

The model results are shown for each packet transmission time (0.4ms), while simulation results are averages over 1ms intervals. Despite these differences, the model and simulation results are reasonably matching, even more so for moderate initial x_0 values (see Fig. 5.11(a)). In all figures, the approximation errors are negligible at the beginning. As we move further in time during the transient phase, however, the buffer occupancy b changes and the errors become larger as a result. Still, those errors are not significant.

Probably the more important results are the extreme UDP utilizations of the transient regime (i.e., the lowest utilization in Figs. 5.11(a) and 5.11(c), and highest utilization in Figs. 5.11(b) and 5.11(d)). These extreme model results, which can also be verified from the generic utilization plots shown in Fig. 5.10, are very close in value to the simulation results.

5.5.2. Miscellaneous Results

The next three subsections present additional results. The first two return to the motivational examples discussed in Sec. 5.1. The results in Sec. 5.5.2.3 are based on a different traffic model—Web Traffic.

5.5.2.1. Results on Example 1

For the two experiments of Example 1 discussed in Sec. 5.1.1, Table 5.2 tabulates three sets of extreme UDP utilization values: theoretical values based on Theorem 5.4.10; and two sets of simulation values averaged over measurement windows of 0.4ms and 10ms. Note that since transient utilization is continuously changing, the time granularity of measurement windows are critical (see also Sec. 5.1.2). In Fig. 5.1, the curves are based on a 10ms window. Compared with simulation results, the theoretical values in Table 5.2 seem to represent *lower* utilization bound when $\alpha > 1$ and *upper* utilization bounds when $\alpha < 1$. Nevertheless, the theoretical and simulation values based on packet transmission time window are remarkably close, and the bounds are tight.

Table 5.2.: Lowest and highest transient UDP utilizations for the experiments of Example 1 in Sec. 5.1.

Scenarios			Extreme UDP utilization		
x_0/C	x_{02}/C	α	Model	Simulation	
				W=0.4ms	W=10ms
0.50	2.0	4	6.7%	7.5%	10.5%
2.0	0.50	1/4	50.8%	49.3%	44.5%
0.25	3.0	12	1.3%	2.0%	3.8%
3.0	0.25	1/12	63.2%	61.4%	56.7%

5.5.2.2. Results on Example 2

Next, for the motivational example in Sec. 5.1.1 where the UDP arrival rate alternates between 1C and 10C, the two extreme utilization values using Theorem 5.4.10 are found to be 0.015% (for change from 1C to 10C) and 75% (for change from 10C to 1C). Similar to what has been observed in the previous section, the extreme values are tight bounds. Even using a gross measurement window of 10ms for Fig. 5.2, we still observe a peak utilization of 72%. In addition to the extreme values, the model allows us to explore the average utilization in a time period within the transient regime. To demonstrate this, Table 5.3 shows average utilizations (per every 500ms) using four methods: (1) steady state utilization corresponding to average UDP arrival rate 5.5C, (2) average of steady state utilizations corresponding to UDP arrival rates 1C and 10C, (3) average utilization based on the model, i.e., Lemma 5.4.7, and (4) simulation results, using measurement time windows of 1ms. Steady state values for the first two methods are based on the OLM model described in Sec. 5.2.1. For using Lemma 5.4.7, the values for backlog size b at each 250ms interval are required. Our extensive simulations show that the backlog size is wildly changing over time. We took the backlog size at the onset of rate change and fed it as the input b_0 into Lemma 5.4.7. For example, at $t = 21$ $b_0 = 765$

and at $t = 21.25$ $b_0 = 675$. As the table shows, the steady-state analytical bounds are far from the simulation results. On the contrary, the transient analysis nicely represents the picture of the queue even under radically changing traffic conditions.

Table 5.3.: Average UDP utilizations for Example 2 in Sec. 5.1.

Time	Average UDP utilization in %			
Interval	$\mu_{0,5.5C}$	$\frac{\mu_{0,1C}+\mu_{0,10C}}{2}$	Model	Simulation
$[21, 21.5)$	11.7	14.8	17.6	18.9
$[21.5, 22)$	11.7	14.8	19.1	19.4
$[22, 22.5)$	11.7	14.8	17.6	18.9
$[22.5, 23)$	11.7	14.8	18.3	19.4
$[23, 23.5)$	11.7	14.8	18.0	19.3

5.5.2.3. Results using Web Traffic

Since the Internet flow dynamics is heavily shaped by short Web transfers, we conducted a 500-replicated experiment explained below. The UDP arrival pattern is the same as in Fig. 5.2 except that $x_0 = 10C$ for $t < 21$ and $x_0 = 1C$ for $t > 23$. The Web traffic is modeled as follows: Starting from $t = 20$s, each of the 100 TCP sources (see Fig. 4.3) generates a Poisson process with an average arrival rate of 25 Hz. The size of each session (file) is Pareto-distributed with an average size of 10kB (about 10 packets) and a shape parameter of 1.3. This model captures the heavy tailed nature of Web file sizes and their transmission times [19]. Simulation lasts for 25 seconds and over 9000 Web sessions have been generated. The result is illustrated in Fig. 5.12.

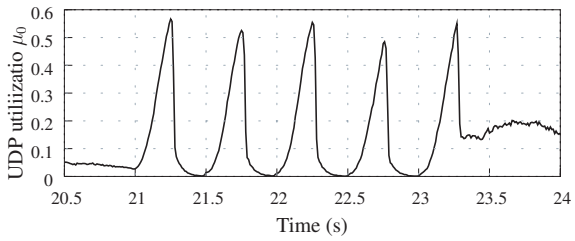


Figure 5.12.: UDP utilization in the presence of Web flows.

Due to the huge and highly bursty Web traffic generated, the buffer is always full. Like in the long-lived TCP scenarios, UDP exhibits widely fluctuating throughput patterns during transient regimes. However, the extreme points of transient regime are generally lower in value. For example, UDP utilization can get as low as 0.08%, and as high as 56%. Nevertheless, in both cases, they are bounded, albeit *loosely* by the analytical extreme values 0.015% and 72% as discussed in Sec. 5.5.2.2. The

smaller simulation values are probably due to the high ambient drop rate r which becomes significant as RED deals with persistent full buffer occupancy. Further study is still required to refine these findings.

The results in this section show that the analytical results and observations made in this chapter may apply to a wider context than studied here.

5.6. Conclusion

While existing works on CHOKe reveal interesting structural, asymptotic and limit behaviors of the queue, their results are limited to the steady state when the queue reaches equilibrium in the presence of many long-lived TCP flows and constant rate UDP flows. Unfortunately, they lack showing properties of the queue in a possibly more realistic network setting where the exogenous rates of unresponsive flows may be dynamically changing, and consequently the model parameters, rather than being static, may be continuously evolving.

This is the first study on CHOKe behavior in the aftermath of rate changes in UDP traffic arrival. In particular, we are concerned with CHOKe queue behaviors during the transient regime which we model as a transition from one steady queue state to another. We found that the performance limits predicted by existing steady state models do not hold for such transient regimes. Depending on the nature of rate change, the queue exhibits instant fluctuations of UDP bandwidth sharing in reverse direction. This behavior has ramifications on the smooth operation of the Internet where most flows are rate-adaptive. Such flows may see fluctuating available link bandwidth and degrade in performance. By extending and leveraging the spatial distribution model, we analytically (1) determine the extreme points of UDP utilization (observed within an order of queueing delay after rate change), and (2) track the evolution of the transient UDP utilization following rate change. In addition, the model allows us to obtain generic UDP utilization plots that help explain both the transient extreme characteristics and steady-state characteristics. The analytic results have been rigorously validated through extensive simulations. The analytical approach used in this chapter can be easily extended to studying transient behaviors of other leaky queues, including other gCHOKe variants.

6. Statelet Fair Queue

Motivated by the fairness performance of S-SFQ in simplified network topologies (see Chapter 3), we seek to extend the algorithm for arbitrary networks and generalize it as a statelet fairness framework. To this end, we present a specific router fairness algorithm we call *Approximate Fairness through partial Finish Time (AFpFT)* in this chapter. AFpFT addresses two important limitations of S-SFQ. First, S-SFQ is stateful and hence a naive extension into inner routers would be infeasible. AFpFT solves this problem by keeping flow state only for a subset of the flows. This is generally acceptable since completely stateless flow protection schemes such as those in Chapter 4, despite being powerful, cannot impose high quality fairness among competing flows. Secondly, S-SFQ can suffer the unfortunate fate of loss synchronization which completely obliterates its fairness quality. AFpFT solves this problem by leveraging the statelet flow information into a simple, yet effective, drop policy that in turn eliminates the buffer and link usage discriminations. Using extensive simulations, we show that the scheme is highly fair and potentially scalable unlike other proposed schemes.

The rest of the chapter is organized as follows. We start by presenting the motivation of this work in Sec. 6.1 followed by research contributions in Sec. 6.2. Our proposed scheme AFpFT is fully explained using pseudocodes and illustrative examples in Sec. 6.3. Sec. 6.4 provides our simulation results and performance comparisons of the scheme against RED, FRED and CSFQ. Sec. 6.5 discusses this work in the general context of other related works. Sec. 6.6 concludes the chapter.

6.1. Motivation

Replicating the fairness of perflow fair queueing algorithms such as SFQ [44] and SCFQ [41] without maintaining perflow states is a difficult challenge [54]. This is because the perflow parameters (e.g., the virtual flow finish times) are not stand-alone but depend on all flows traversing the router. Due to this dependency, encoding the perflow parameters of SFQ and SCFQ into packets at ingress nodes, and using them later for scheduling inside the network is not possible.

The key objective of AFpFT is *to retain the nice fairness qualities of perflow schemes without keeping states for all flows.* In this section, we state the limitations of the existing approaches which motivate this work. In a broader sense,

these limitations impact the scalability for high-speed implementation, or their widespread acceptability among the various network stakeholders.

Recall from Sec. 2.3 that the two broad approaches to impose flow fairness in networks are either end-to-end based and router-based. Fairness based on e2e schemes is dependent on *cooperation* or universal adoption. Universal adoption may not be possible for reasons outlined in Secs. 1.2.1 and 2.3. As explained, with increasing heterogeneity in deployed end-to-end based congestion control schemes, equitable resource sharing among Internet flows may not be possible.

The second approach for flow fairness is router-based as described in Sec. 2.4. There are two sub-categories in this regard: (1) perflow fair queueing and (2) queue management mechanisms. The flow protection in the former comes with a large number of physical queues, or else with expensive buffer partitioning and dynamic scheduling states (e.g., the pointers to each packet queues) required to maintain the resulting queue structures [57]. Another binding constraint is that the perflow schemes are inherently stateful requiring the maintenance of flow-level state or history [92], see Sec. 2.4.1. Other commonly cited constraints are sorting of queues that may precede a packet transmission and, in some cases, packet dropping. At each transmission epoch, the scheduler may need to identify the queue or flow from which to send the next packet. In SFQ and SCFQ, for example, it may be from the queue whose head-of-line packet has the smallest service tag. Likewise, when a packet arrives to a full buffer, the policy for dropping may be preceded by sorting the queues. The schemes may discard a packet from the most backlogged queue (i.e., Longest Queue Drop LQD [94]).

Queue management mechanisms, on the other hand, offer simplicity at the expense of generality and quality flow protection. A flow-myopic solution of applying equal drop probabilities indiscriminately to all flows, as in RED, does not result in a *fair* allocation, as demonstrated by the simulation results of the RED queue in Chapter 3. As pointed out in Sec. 2.4.2.2, several statelet RED extensions that *identify* the high-bandwidth flows and apply additional perflow drop rates have been proposed. We raise two performance problems with these schemes. First, lacking the powerful flow isolation of perflow queueing, the flow protection of QMs is generally poorer even with the additional perflow drop rates. Secondly, the flow identification policy assumes that the flows conform to the Standard TCP. Because of this, they may perform sub-optimally in the presence of other traffic types. Examples of mechanisms that rely on this assumption are FRED, Stabilized RED (SRED) [74], and RED-PD. For example, the authors of FRED recommend $\min_q = 3$ since a Standard TCP sends no more than 3 packets back-to-back. RED-PD identifies and punishes at a pre-filter those flows having more recent drops than a reference Standard TCP with round-trip time RTT_t. We demonstrate the pitfall of the Standard TCP assumption with an example using SRED.

Example. SRED is a queue management scheme designed to keep queue size at a preset value, say Q_0. The average congestion window W of TCP congestion

avoidance algorithms in the steady state is linked to the loss event rate p as $W \sim p^{-d}$, where $0.5 \leq d \leq 1$ is a constant dependent on the TCP algorithm. For example, $d = 0.5$ for the traditional TCP. See also Sec. 7.2. Inflight packets of the N flows occupy the target buffer occupation Q_0. SRED assumes all N flows adopt the traditional TCP algorithm. That means, $Q_0 \approx N \times p^{-0.5}$, or the drop rate becomes $p \approx (\frac{N}{Q_0})^2$. N is approximated as $1/P(t)$, where $P(t)$ is the exponential weighted hitting frequency. The p computation is not accurate for other non-traditional TCP traffic. As a result, the N or $P(t)$ estimation would be grossly imprecise in mixed traffic situations. In order to clarify, we run two separate experiments each with 200 sources: (i) all TCP, and (ii) all *except* 1 UDP source are TCP (UDP source rate is 50% of link capacity C). Fig. 6.1 shows the error in N approximation, hence the flaw in the dropping rates p computation in the second experiment. The all-TCP experiment approximates $N \approx 160$, while in the second experiment $N \approx 6$ which significantly decreases the drop computation. As a result, the UDP takes up almost as much bandwidth as its sending rate (50% of link capacity).

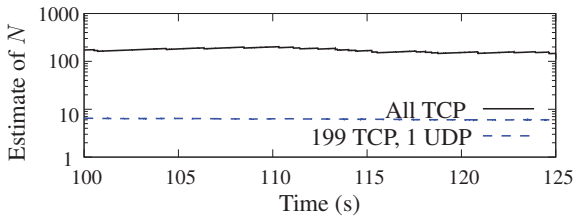


Figure 6.1.: Flaw in estimation of flow count N.

The idea of ensuring flow fairness with no or limited perflow state is not new. A notable example in this regard is the Core-Stateless Fair Queueing CSFQ [92], see Sec. 2.4.3. CSFQ core routers compute the local max-min fair share rate Φ_{share} before inserting the new outgoing flow rate $\min(\Phi_{share}, \Phi_i)$ into packet headers and forwarding the packets to downstream routers. It is easy to see that the computation of the fair share, and hence flow dropping rates, is based on implicit trust of upstream nodes in the network. A faulty router along the flow's path can therefore insert inconsistent values into packet headers and severely undermine the fairness, and the overall performance, of CSFQ [93]. Therefore, CSFQ can neither transcend network boundaries nor withstand malfunctioning or wrongly configured routers.

Summary: While technically superior, many of the existing approaches (1) may be stateful and complex, (2) target specific traffic type and hence may lack generality, (3) are based on implicit trust and hence lack robustness to cope misconfigurations. In a broader sense, one or more of these shortcomings can limit their scalability and potential widespread deployability.

6.2. Contributions

In this chapter, we propose a powerful flow fairness router mechanism called *Approximate Fairness through partial Finish Time* (AFpFT) which has the following features: (1) a single-aggregate queue shared by all flows, (2) a drop policy free of a particular TCP assumption, (3) *statelet*, or keeps state for a limited number of flows which become listed in a flow list $\mathcal{FL}$, and (4) packet sorting upon arrival but not upon packet dequeueing and dropping. AFpFT is a specific example of *statelet fair queue*, see Sec. 6.3.3. Due to the nature of packet sorting or tag computation following packet arrival to queue, packets of low or moderate rate flows obtain "favorable" tags which push them closer to the queue head, while those packets coming from persistent high rate flows are placed near queue tail, probably incurring losses or delays before transmission. The favorable tagging allows low rate flows to obtain time (or scheduling) and space (or buffering) priorities. Incidentally, the flows whose states are kept in the flow list $\mathcal{FL}$ at inner routers are relatively higher in arrival rates and fortunately fewer in number, making the scheme *statelet*. We conduct extensive simulations under different operating conditions. The results demonstrate that AFpFT is superior in fairness to other related schemes such as FRED and CSFQ. It can, for instance, restrict the high bandwidth flows (e.g., TCP flows with small round-trip times or aggressive flows that lack e2e congestion control) to a common fair share rate.

6.3. The Scheme

6.3.1. Conceptual Design of AFpFT

When a packet of a flow arrives to an AFpFT queue, it is assigned a timestamp or tag which defines its position in the queue. Packets are sorted in ascending order of their tag values from head to tail of the queue. Regardless of whether the buffer is already full or not, the arriving packet is always sorted first in the queue before any decision to drop a packet is made. If the buffer becomes full, the packet at the queue tail will be dropped. More than one packet may be dropped since the arriving packet may be larger than the one(s) at the tail. The pseudocode is shown in Fig. 6.2. Note that the dequeue operation is trivial and not shown: the packet at the queue head is always the one that is dequeued for transmission.

Apart from the sorting queue, the most important functional components of AFpFT are the *tag computation*, the *flow list* $\mathcal{FL}$ and the *drop policy*, which we describe shortly.[1]

[1] After a period of inactivity, idle flows expire and are removed from the flow list $\mathcal{FL}$.

```
1: Upon packet p arrival to queue
2: if  previous packet p̂ in queue then
3:    // expensive tag
4:    Γ_p = Γ_p̂ + Δ_p̂
5: else
6:    // cheap tag: note p⃗ is packet in transmission
7:    Γ_p = Γ_p⃗
8: end if
9: Put p in ascending order of Γ_p in queue
10: while queue full do
11:   drop a packet from the queue tail
12: end while
```

Figure 6.2.: Conceptual pseudocode of AFpFT enqueueing and dropping

6.3.1.1. Tag Computation

As can be seen there are two ways of tag computation for an arriving packet, labeled *cheap* and *expensive* in Fig. 6.2. Expensive tag is applied when a previous packet $\widehat{p}$ of *the same flow* is found in queue, in which case the tag of the arriving packet Γ_p is a small increment $\Delta_{\widehat{p}}$ over that of $\widehat{p}$.[2] If multiple previous packets of the flow are found in queue, only the one that arrives last is considered. If no previous packet of the flow is found, however, the tag Γ_p just takes up the tag of the packet in transmission, $\overrightarrow{p}$, regardless of whether or not $\overrightarrow{p}$ belongs to the same flow as the arrival. This latter case is cheap tagging for the flow. Cheap tagging allows the arriving packet to get sorted closer to the head of the queue, in the process receiving higher priority than those with expensive tagging.

Illustrative Example: Consider an AFpFT queue at an arbitrary busy time t when a flow f_2 packet is in service (i.e., being transmitted), see Fig. 6.3. At time t, let us assume that the queue serves five flows $f_1, \cdots, f_5$ which have 0, 0, 1, 2, 1 packets in the queue, respectively. The horizontal bars show the flows' timestamps, i.e., the tag of the last (previous) packet of the flows in queue. Let us also assume that all packets have equal length.

Since packets are sorted in ascending order of tags, f_1's tag Γ_{f1} must be smaller than all others, and that the tag Γ_{f4} must be the greatest of all the five flows. If a packet of flow f_1 (or f_2) were to arrive at time t, since the flow has no packet in queue, it would invoke cheap tagging that allows the arriving packet to obtain the tag of the packet in transmission, i.e., Γ_{f2}. On the contrary, however, if a packet of flow f_3 were to arrive, it would receive an expensive tag of $\Gamma_{f3} + \Delta_p$. Similar expensive tagging is invoked for packet arrivals of flows f_4, and f_5. While an f_1 arrival obtains the least tag and gets sorted to the front of the queue, incoming packets belonging to flows f_3, f_4 and f_5 are placed away from the head. If the buffer becomes full, these arrivals—particularly those of f_4—may even be dropped.

[2] $\Delta_p = L_p/L_M$, where L_p and L_M are the sizes of packet p and the maximum packet, respectively.

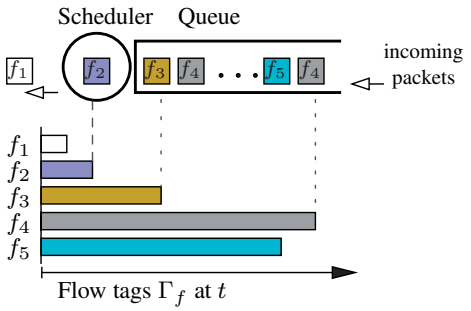


Figure 6.3.: Structural view of the queue in operation.

Observation 1. *All packets already buffered in AFpFT queue at any time t must have tags greater than that of the packet in transmission.*

The rationale for the differential tag computations is as follows.

- Consider a high-bandwidth flow. It is more likely that such a flow is bottlenecked since it naturally sends many packets, some of which may reside in the buffer. The incoming packets of such a flow are likely to incur expensive tag computation, see line 4 in Fig. 6.2.
- Consider a flow with a rate lower than the fair share rate, or a newly started or restarted flow. Such a flow is unlikely to see previous packets of own flow upon arrival. Its arriving packets invoke cheap or "favorable" tagging which allows them to be scheduled close to the queue head.

A consequence of the differential tag computation is that the distribution of flow packets in AFpFT queue is most likely to be non-uniform.[3] High bandwidth flows have their packets clustered closer to the queue tail, and packets from lower rate flows closer to the head of the queue.

It is trivial to see that the tag computed for a packet is only meaningful for local packet scheduling decisions. A wrong service tag computed at an arbitrary AFpFT node does not impact the tag computation at downstream nodes, making AFpFT more robust against router errors, unlike CSFQ [93, 90]. The tag encoding at a router is merely based on the availability of flow's entry in the list.

6.3.1.2. AFpFT as a Statelet Scheme

From an implementation perspective, AFpFT is a statelet scheme, keeping state for a subset of the active flows in its flow list $\mathcal{FL}$. The amount of state in $\mathcal{FL}$ is limited by the buffer size. For the non-bottlenecked flows, incoming packets can simply

[3] Other queues exhibiting this behavior are CHOKe [78, 98] and Geometric CHOKe in Chapter 4.

pluck their tagging directly from the packet in transmission. For bottlenecked flows, however, AFpFT needs to keep the $\Gamma_{\widehat{p}}$—the tag of the last packet of the flow found in the queue. The flow list size in $\mathcal{FL}$ is thus limited by the buffer size.

The two important perflow fields in $\mathcal{FL}$ are $count_i$ which stores the number of flow i packets in the queue and $finish_i$ which stores the finish tag $\Gamma_{\widehat{p}} + \Delta_{\widehat{p}}$ of the last arrived packet of flow i, respectively. As packets arrive to queue, both fields are updated to reflect the change. $finish_i$ is a *relative* measure of the amount of service received by flow i (amount of flow traffic transmitted in units of L_M). For example, a high bandwidth flow i has relatively frequent arrivals and enqueueing, and this increases the $finish_i$ (or the corresponding horizontal bar in Fig. 6.3) significantly. When packets of flow i arrive, they assume increasingly larger service tags which in turn push them to the queue tail. On the other hand, packets of new flows would start with smaller $finish$ as they have not received any service.

$\mathcal{FL}$ lists mostly the bottlenecked flows which are often the relatively high bandwidth flows. Numerous measurement studies [105, 83, 66, 76] report that Internet flow distribution is highly skewed with respect to sizes, and rates of flows. Accordingly, a large proportion of Internet flows are indeed short-lived web traffic that end up in TCP slow start phase. And, the "heavy-hitters", or large and fast flows in the Internet, are very few in number but contribute the lion share of Internet traffic in volume. Other studies based on real network traces, e.g., [60], show that even though the number of flows in progress can be very high at a router, the actual number requiring buffering is limited to few hundreds both at very high-speed and residential links. Using AFpFT, therefore, we mainly need to manage in $\mathcal{FL}$ the few (potentially "heavy-hitting") flows that regularly have packets queued in the buffer. By keeping state only for a very small subset of in-progress flows, AFpFT makes sure that the few relatively high rate flows cannot get more than the fair share. In addition, most low rate flows often use cheap tagging to get service ahead of the high rate flows. Since small flows are naturally disadvantaged, this preferential treatment is probably a desirable quality of router mechanisms [21]. AFpFT inevitably fulfills that objective without making it an explicit design goal.

6.3.1.3. Drop Policy

Packet dropping in AFpFT is preceded by packet enqueueing. That is, when a packet arrives to an AFpFT queue whose buffer capacity or threshold is reached, the arriving packet must first be enqueued based on its tag. Then, the packet(s) found at the tail are dropped. More than one packet may be dropped per arrival. This is because the arriving packet may be larger in size than the one(s) that are dropped. But this is only half of the story.

When packets are dropped, their corresponding flows receive no service. A dropped packet has earlier updated the flow's *finish* time when it arrived to the queue. Therefore, a packet dropping should immediately be followed by some cor-

rection of the flow finish time *finish*, and also *count*, in the $\mathcal{FL}$. In order to understand the correction required, consider a congested router serving well-behaved flows and an aggressive high rate flow f. Flow f often finds its packets queued near the tail. When a flow f packet arrives, it first updates $finish_f$ in $\mathcal{FL}$ before being queued near the tail. Later, the packet may be pushed to the tail and dropped because of arrivals from other competing flows. In the long run, the flow's finish time no longer, as it should, indicates the flow's transmission progress (or service received), but the sum of the service received plus the dropped traffic ("obituaries") of the flow. Therefore, when a packet p of a flow is dropped, since the packet's contribution to the flow's throughput is *zero*, the flow's state (e.g., $finish_f$) should be as if the packet had never arrived to the queue in the first place. In short, we must cancel out the contribution of p's arrival to the flow finish time. Finding the exact contribution is generally not possible because the $finish_f$ updating is done upon p's arrival, usually earlier than the time of its dropping. Since we lack information about the previous $finish_f$ values, we cannot recover the value of $finish_f$ when p arrived. Therefore, we fairly approximate this contribution by Δ_p. Summarizing, when a packet p belonging to a flow f is about to be dropped, we update the flow state as follows. If there are no other packets of the same flow in the queue, the flow is removed from the flow list $\mathcal{FL}$. Otherwise, $count_f$ decrements by 1, and the current value of $finish_f$ is decremented by Δ_p.

In order to emphasize the importance of the flow finish time correction following packet drops, we provide two examples. The first is the loss synchronization problem of Sec. 3.3.1.

Example 1: We use the same terminology as in the original example and let us assume that $\Delta_p = 1$. From that example, service tag $S(p_f^k) = k - 1$ for both flows $f \in \{1, 2\}$. By time $t = 2$, packets p_1^1 and p_2^1 have been transmitted, and p_1^2 is under transmission, and packets p_2^2 and p_1^3 occupy the queue. Packet p_2^3 arrives surplus to the buffer capacity with a tag of 2. At $t = 2$, the finish tags of both flows would be $k = 3$. Since p_2^3 is dropped, $finish_2$ is reduced to 2. By $t = 3$, p_1^1, p_2^1, p_1^2 are all gone, p_2^2 under transmission, p_1^3 occupies one slot in buffer, and packets p_1^4 and p_2^4 just arrive to queue. One of the last two arrivals must be dropped. Due to finish time correction of flow 2, p_2^4 obtains a smaller tag than p_1^4 and gets ahead of it, even though it arrives slightly later. The sequence of packet drops is then alternating, and the full fairness restored as shown in Fig. 6.4.

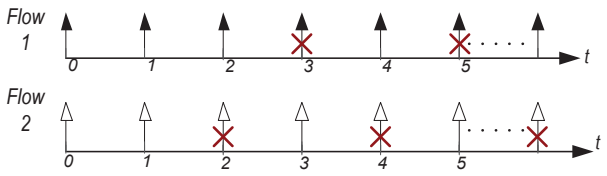
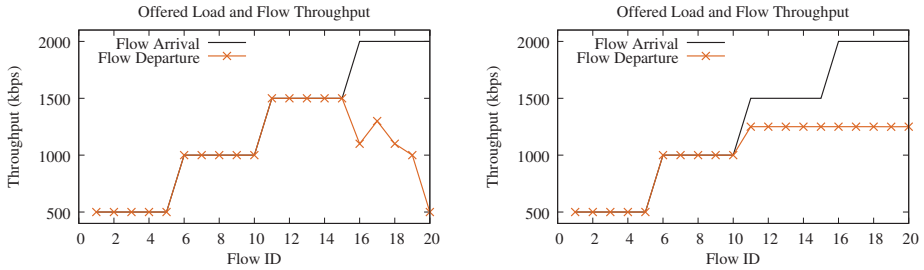


Figure 6.4.: Riddance of loss synchronization.

Example 2: Figure 6.5 shows the result of 20 CBR sources sending at four different flow rates of 0.5Mbps, 1Mbps, 1.5Mbps, and 2Mbps for a total traffic rate of 25Mbps. The capacity of the AFpFT link is 20Mbps. Flow groups 3 and 4 send more than the fair share rate of 1.25Mbps. As can be seen, without the finish time correction following packet drops, the flows in the last group are punished while the third flow group is transmitting at full incoming rate (see Fig. 6.5(a)). With the correction, both high rate flow groups are limited to their fair share (see Fig. 6.5(b)).



(a) AFpFT without $finish_f$ adjustment. (b) AFpFT with $finish_f$ adjustment.

Figure 6.5.: Offered Load and Flow Throughput of 20 CBR sources under AFpFT.

6.3.1.4. Router Roles

AFpFT can be put into a DiffServ-like framework with edge and inner/core router implementations. Since the number of flows are often limited at the edge, edge routers can afford to add all in-progress flows to the list $\mathcal{FL}$ and maintain the full perflow information; that is, edge routers can indeed perform perflow fair queueing such as SFQ [44].[4] This allows all traffic entering the network to be conditioned at the local router. On the other hand, an inner router maintains flow information only if the flow has some packets in the queue, as explained before. The question is: *how to differentiate the routers?* One solution is for traffic sources to tag packets with invalid or *negative* values. The first router where the flow enters the network acts as edge / ingress upon reading the negative tag values. If the packet tag value is nonnegative, the router assumes that the tag must have been assigned by an upstream node and therefore acts as an inner node to the flow. From the above description, we see that there is no hard-and-fast router configuration as such. The router acts as an edge to a flow where that particular flow enters the network, and acts as inner router to the flow otherwise.

[4]The flows can be removed from $\mathcal{FL}$ of edge routers through flow timeouts.

6.3.2. Full Design of AFpFT

Figs. 6.6 and 6.7, respectively, demonstrate more complete pseudocodes of the AFpFT enqueueing and dequeueing operations at inner nodes. The following notations are used in the pseudocodes.

Flow list $\mathcal{FL}$ variables	
$count_f$	count of flow f packets in AFpFT queue $\mathcal{Q}$
$finish_f$	finish timestamp of flow f
Functions and other variables	
Γ_p	service tag of packet p
V	tag of packet in service / transmission
$conn(p)$	flow/connection id of packet p

Variable V keeps track of the service tag of the packet under transmission. Note that V is the smallest among all tags of queued packets. When a packet p of flow f arrives to queue, the following router actions occur. If the flow is not listed, a flow entry is created before packet p is assigned a tag. Then, f's flow parameters are updated in the flow list $\mathcal{FL}$ and p is enqueued. If the buffer overflows, the packet(s) found at the tail are dropped. Let us assume that the dropped packet(s) belong to flow i. Flow i parameters are updated in $\mathcal{FL}$ following packet loss(es). If the $count_i$ decrements to *zero*, flow i is removed from the flow list. For transmission, on the other hand, a packet at the head of the queue is chosen. The tag of the dequeueing packet is copied to V, followed by updating of $count_i$ in the flow list, and a potential removal of flow entry in $\mathcal{FL}$ if the flow has no more queued packets. When the queue is empty, V is reset and all flows entries are removed.

6.3.3. Generalizing the Scheme

AFpFT is a *statelet fair queue*—our generic term for AFpFT-like single queues shared by all flows. Such queues sort arriving packets based on packet tags, and have drop policies similar to that of AFpFT. They keep state only for those flows having packets in queue. By varying the tag computation, we can define other statelet fair queues. A generalized tag computation is given below. When packet p of flow i arrives to queue, the packet tag assumes either of the two values:

$\Gamma_p = \Gamma_{\overrightarrow{p}} + f_1(\Delta_p, \Delta_{\overrightarrow{p}})$ // cheap tagging (no flow i packet in queue)

$\Gamma_p = \Gamma_{\widehat{p}} + f_2(\Delta_p, \Delta_{\widehat{p}})$ // expensive tagging (previous flow i packet $\widehat{p}$ in queue)

where $f_1(\cdot)$ and $f_2(\cdot)$ are some functions. For AFpFT, $f_1(\cdot) = 0$, and $f_2(\cdot) = \Delta_{\widehat{p}}$.

As in AFpFT, packets are sorted into the queue in ascending tag values, and hence queued packets have higher tags than the packet in service. A possible future work is the close examination of other statelet fair queue tag computations and their performance comparisons to that of AFpFT.

```
1: Upon receiving p
2: f ⟸ conn(p)
3: if (f ∉ FL) then
4:    Add f to flow list FL; initialize count_f = 0
5: end if
6: if (count_f ≥ 1 ) then
7:    Γ_p ⟸ finish_f   //expensive tag
8: else
9:    Γ_p ⟸ V   //cheap tag
10: end if
11: //The next two lines update the f variables in FL
12: finish_f ⟸ Γ_p + Δ_p
13: count_f ⟸ count_f + 1
14: enque p into Q   // based on Γ_p value
15:
16: //Block A—drop policy when queue Q becomes full
17: while (Q-size ≥ Q-limit) do
18:    Draw packet p from Q tail
19:    i ⟸ conn(p)
20:    if (i ∈ FL) then
21:       count_i ⟸ count_i − 1
22:       finish_i ⟸ finish_i − Δ_p   //discrediting "obituaries"
23:    end if
24:    if ( count_i < 1) then
25:       remove flow i from FL
26:    end if
27:    drop p
28: end while
```

Figure 6.6.: AFpFT packet enqueueing and dropping.

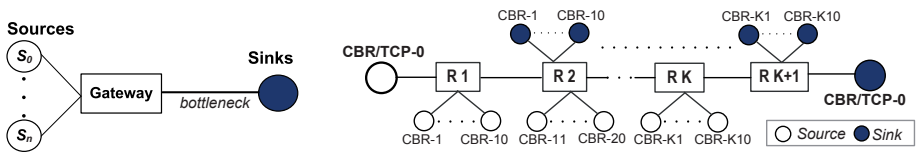
6.4. Performance Evaluation

6.4.1. Topologies and Parameters

We implemented AFpFT in ns-2. Unless stated otherwise, the topologies shown in Fig. 6.8, and simulation parameters summarized in Table 6.1 are used for evaluation. AFpFT flow fairness and link utilization are compared with Adaptive RED [36], FRED [64], and CSFQ [92]. FRED and CSFQ implementations are freely available [91]. As described in Chapter 2, FRED fairness mechanism is through fair allocation of buffer space to flows, while CSFQ adopts perflow dropping rates $(\Phi_i - \Phi_{share})/\Phi_i$ to bring down the outgoing rates of bottlenecked flows to Φ_{share}. K and K_α are averaging constants (time windows) used for estimating incoming flow rates and

1: *Draw* $\overrightarrow{p}$ *from* $\mathcal{Q}$ *head*
2: **if** ($\overrightarrow{p}$ exists) **then**
3: // Block B— $\overrightarrow{p}$ is packet in service
4: $V \Longleftarrow \Gamma_{\overrightarrow{p}}$
5: $i \Longleftarrow conn(\overrightarrow{p})$
6: **if** ($i \in \mathcal{FL}$) **then**
7: $count_i \Longleftarrow count_i - 1$
8: **end if**
9: **if** ($count_i < 1$) **then**
10: **remove** flow i from $\mathcal{FL}$
11: **end if**
12: **else**
13: //Block C— $\mathcal{Q}$ empty; reset the queue & remove flows from $\mathcal{FL}$
14: $V \Longleftarrow 0.0$
15: $\forall j \in \mathcal{FL}$: **remove** flow j from $\mathcal{FL}$
16: **end if**
17: **return** $\overrightarrow{p}$

Figure 6.7.: AFpFT packet Dequeueing.



(a) Single Congested Link. (b) Multiple Congested Links.

Figure 6.8.: Topology used for evaluation.

the fair share Φ_{share}, respectively. In AFpFT, the only free parameter is L_M which is set to 1kB.

6.4.2. Single Congested Link

Fig. 6.8(a) shows 32 long-lived TCP (New Reno) flows and a single CBR flow competing for a scarce 1Mbps bottleneck. The CBR flow sends at full bottleneck capacity of 1Mbps, and the TCP windows are unlimited. The bottleneck buffer size is 100kB. Figures 6.9 shows the results averaged over 30 replications. The fair share rate in the above scenario is 30.3kbps per flow. AFpFT is an extremely fair scheme, providing an average of 29.5kbps to each TCP flow, close to the ideal fair share. It also manages to restrict the nonadaptive UDP flow to the fair share. The same cannot be said for RED which allows unfair link domination (in excess of

Table 6.1.: Default Settings used for Evaluation.

General Parameters		Algorithms	
Link and Buffer		**RED**	
Link speed	1Mbps	$\min_{th}$	25% Buf lim.
Prop. delay	1ms	$\max_{th}$	75% Buf lim.
Buf. limit	50kB	`adaptive`	yes
Traffic Source		**FRED**	
TCP	New Reno	$\min_{th}$	25% Buf lim.
TCP segment size	960 B	$\max_{th}$	75% Buf lim.
UDP packet size	1000 B	**CSFQ**	
Flow start time	$t \in [0.0, 5.0)$	K	100 ms
Simulation		K_α	200 ms
Simul. Duration	50s	Buf thresh.	50% Buf lim.
Results taken	2^{nd} half	Flow weights	Equal
Replications	30 or 100	**AFpFT**	
Confidence level	90%	L_M	1kB

a whopping 70%) by the unresponsive flow. Average TCP flow throughput is a meagre 8.4 kbps in RED. All flows are uniformly punished even though only UDP is responsible for congestion. FRED significantly improves upon RED due to its per flow dropping rates. Nevertheless, UDP still gets twice as much as an average TCP flow (57kbps vs 28 kbps). In addition, despite the fact that all TCP flows are identical with respect to round trip times, congestion control algorithm and receiver window sizes, a maximum difference of 12kbps ($\approx$ 40% of ideal share) is noted between TCP flow throughput values in FRED. Using CSFQ and AFpFT this difference is, however, less than 3kbps and 1kbps, respectively. The Jain's fairness index (see Sec. 2.2.4) scores between these identical TCP flows are: 0.9329 (RED), 0.9905 (FRED), 0.9994 (CSFQ) and 0.9999 (AFpFT). While CSFQ is better than FRED in fair distribution of bandwidth among TCP flows, its performance with regards to TCP bandwidth quota is generally poorer than FRED. Average TCP flow throughput is 26kbps and UDP takes up more than three times the fair share. This may largely be explained by the inadvertent packet dropping scheme employed by CSFQ and its consequential impact on TCP throughput. CSFQ's per flow dropping rate is based entirely on the incoming flow rate Φ_i and fair share Φ_{share} and is given by $\frac{\Phi_i - \Phi_{share}}{\Phi_i}$. CSFQ's objective is to limit the throughput of a congested flow i to the fair share Φ_{share}. Dropping with the above rate can, due to the unresponsive nature of UDP, successfully bring down UDP flow output rates to Φ_{share}. However, dropping TCP flows with the above rate[5], rather than limit the

[5] A TCP friendly drop rate in the steady state should consider the desired average TCP throughput and round trip delays ([75, 35, 66]). See also Eq. (2.3).

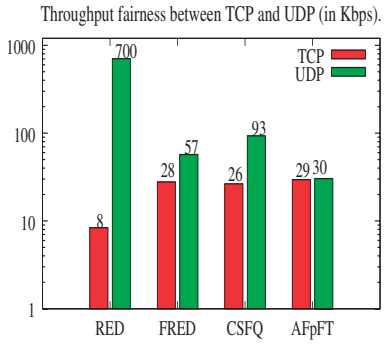
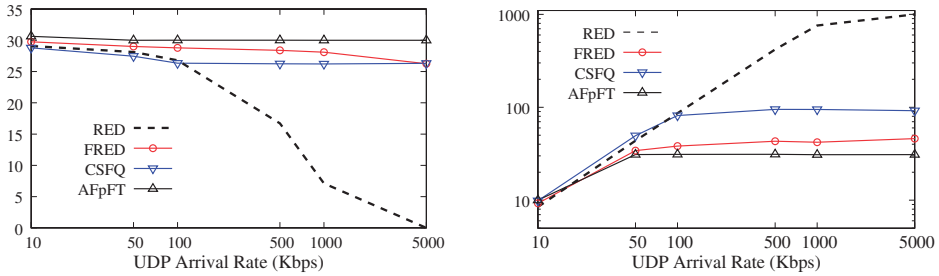


Figure 6.9.: UDP and average TCP throughput [kbps].



(a) Average TCP Throughput Comparison. (b) UDP Throughput Comparison.

Figure 6.10.: Average TCP and UDP Throughput [kbps] as incoming UDP rate is varied.

flow rate to the fair rate Φ_{share}, may potentially reset the congestion windows. This occasional but unfortunate situation can degrade average TCP flow throughput.

What happens to TCP throughput if we vary the incoming rate of the UDP flow? In Fig. 6.10, the UDP source rate increases by a factor of 500 along the x-axis. Most values are in logarithmic scale. Performance difference becomes clearer with higher incoming UDP rates. RED is very poor at restricting high bandwidth flows. The linear RED curve in 6.10(b) indicates that the UDP share is a linear function of the incoming UDP rate. When the UDP arrival rate is 5Mbps, over 99.99% of link capacity is used by UDP, completely starving out the well-behaved TCP flows. As before, CSFQ offers approximately 100kbps to UDP which is several times larger than the TCP share. While FRED is better at controlling high bandwidth flows than CSFQ, the TCP share slightly decreases with increasing UDP traffic rate. AFpFT provides precisely the same fair share to TCP at all levels of incoming UDP noise.

Another set of experiments consisting of only CBR flows is carried out, and the results are shown in Figure 6.11. A total of 32 flows (with flow i sending at

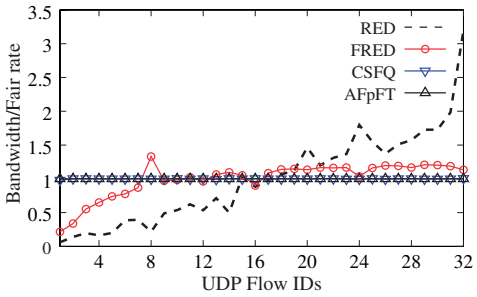


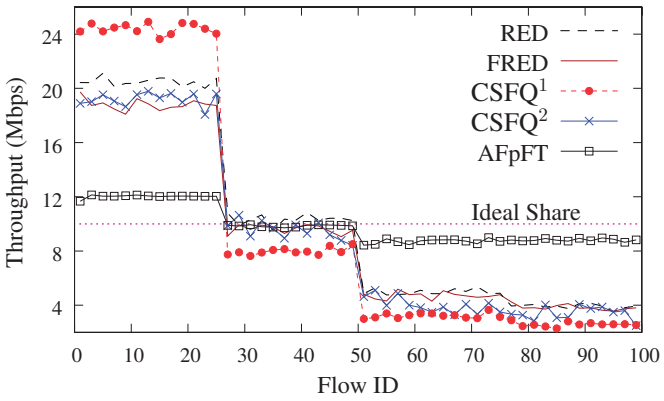
Figure 6.11.: Normalized throughput allocated for UDP flows.

$i \times 0.3125$Mbps) are simulated. Following our argument earlier in this section, this all UDP scenario is the natural environment for CSFQ's fair bandwidth allocation. AFpFT performs as good as CSFQ, allocating exactly the fair share of 0.3125Mbps to all flows even though the last flow, for instance, sends 32 times as fast as the first one. FRED, by contrast, fails to deliver the fair bandwidth allocation; indeed, the maximum allocation can be several times larger than the minimum allocation.

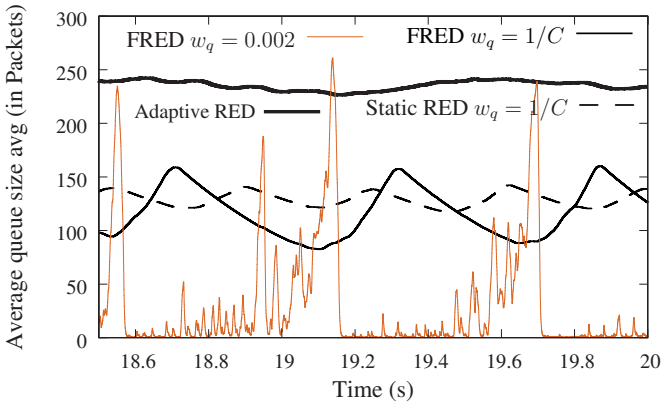
6.4.3. Link Scalability and Different RTTs

This section considers fairness when TCP flows have largely distinct RTT delays in a high-speed environment. The link speed at bottleneck is set to 1Gbps and the link latency to 5ms. Now we have a total of 100 TCP flows divided, based on their RTTs, into four equal groups: G_1, G_2, G_3, G_4. The RTTs are respectively 20ms, 40ms, 80ms, and 100ms. The queueing delay at a gigabit link is insignificant. Flow i belongs to $G_{\lceil i/25 \rceil}$. We use 500 packets as our buffer size, which is a small fraction of the bandwidth-delay product (BDP). The BDP is traditionally used as a general rule of thumb to provision buffer size [6]. Under current routers (Drop-Tail) where synchronous packet losses could be common, average TCP throughput is inversely proportional to RTT [75]. Specifically from Sec. 2.2.2, the ratio between throughput of two TCP flows is given by $T_1/T_2 \sim \left(\frac{RTT_2}{RTT_1}\right)^{\alpha}$ [100], where $1 < \alpha < 2$. Letting $\alpha = 1$ (optimistic assumption), the flows in groups G_1,...,G_4 would attain throughput r_1, $r_1/2$, $r_1/4$, $r_1/5$, respectively, where the G_1 flow throughput $r_1 = 21.33Mbps$. Figure 6.12(a) illustrates per flow throughput under different schemes.

First, the most unexpected of the observations: with default $K = 100ms$ and $K_\alpha = 200ms$, CSFQ performance, labeled CSFQ[1] in the figure, is worse than RED and FRED. Using RED, for instance, the average throughput per G_1/G_4 flow are 20.6/3.9 Mbps, respectively. Corresponding CSFQ values are 24.3 and 2.6 in Mbps. We believe that the averaging constants are too relaxed for CSFQ to be able to accurately estimate and control the rates of low RTT high speed TCP flows. With tighter window constants of $K = 20ms$ and $K_\alpha = 40ms$, CSFQ fairness shown



(a) Flow Throughput Fairness over a Gigabit link



(b) Impact of *parameter sensitivity* on FRED's link utilization.

Figure 6.12.: Results for Sec. 6.4.3.

as CSFQ^2 is much better, e.g., the average G_1/G_4 per flow throughput are 19 Mbps and 3.4 Mbps. Of all schemes, however, only AFpFT is significantly fair: G_1 flows obtain only 20% more than the fair share and G_4 flows receive on average 12% less than the fair share. We find that FRED is unfair and only marginally better than RED. G_1 flows in FRED receive on average 4.8× than those of G_4. Therefore, fairness in a network with a mix of different RTT flows, as is the case in the Internet, could be very poor under FIFO, RED, FRED and CSFQ. In addition, FRED produces both the least total throughput and total link utilization of all schemes. In steady state, we obtain the following link utilizations: RED (99.9%), FRED (92.7%), CSFQ (95.7%) and AFpFT (98.9%).

The RED version imbedded in FRED is to blame for the poorer link utilization of FRED. FRED was proposed much earlier than the `adaptive` and `gentle`[6] parameters introduced to enhance RED's performance. FRED may therefore suffer from poor queue utilization caused by sensitivity to the parameters (see [36, §5.1]). The default FRED values $\max_p = 0.1$ and $w_q = 0.002$ are too conservative for the scenario in hand. RED's recommended queue averaging constant w_q, for instance, is $1 - exp(-1/C)$, where C is the link speed in *packets/sec*. Since C is very high—1.25×10^5 in this case—we can fairly approximate it as $w_q \simeq 1.0/C$.[7] Without the `adaptive` feature, the static parameters do not allow FRED to operate optimally. Adaptive RED, on the other hand, can improve throughput performance by self-tuning, enabling it to maintain a target *avg* away from $\max_{th}$. And `gentle` smoothly increases the dropping probability p_b when the *avg* exceeds $\max_{th}$. As a consequence, queue utilization in FRED, and generally in non-adaptive RED, is poor. Figure 6.12(b) shows the link utilization curves for non-adaptive RED, Adaptive RED with `gentle`, FRED with default $w_q = 0.002$, and FRED with $w_q = 1/C$ all in a typical run of heavy load scenario. Adaptive RED always attempts to maintain *avg* around $\frac{1}{2}(\max_{th} + \min_{th}) = 250$. We see that the default value $w_q = 0.002$ is too large, making FRED's *avg* extremely sensitive to the actual queue size. The behavior of FRED ($w_q = 1/C$) is similar to the non-adaptive RED (with default $w_q = 1/C$), i.e. both increase *avg* in cycles corresponding to alternating periods of high loss and low loss. Hereafter, we use $w_q = 1/C$ for FRED.

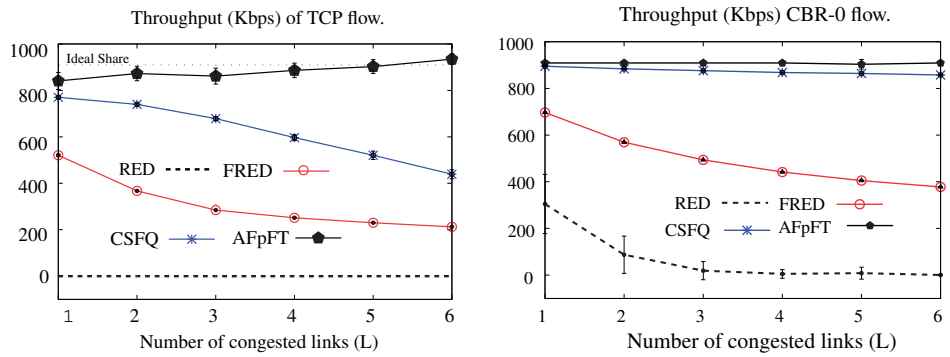
6.4.4. Multiple Congested Links

This section discusses how flow throughput fairness is affected across multiple congested links. Topology for this section is borrowed from [92] and shown Fig. 6.8(b). The links connecting hosts to routers are 20Mbps, and routers to routers 10Mbps. Each 10Mbps link is traversed by 11 flows: 10 UDP (CBR) cross flows each with rates 2 Mbps and a well-behaved flow 0. In the first experiment, flow 0 is TCP. In the second experiment, flow 0 is CBR sending close to the fair share of 0.909Mbps. We collect throughput of flow 0 as a function of the number of congested links L. Flows start times are uniformly distributed on $[0, 5.0]$. Results from 100 replications are shown in Fig. 6.13.

Regardless of whether the flow is TCP or CBR, its throughput decreases as it traverses multiple congested links. Since UDP does not react to congestion, its throughput is generally higher than that of TCP. There is a total lack of protection for TCP flow when RED is used, see Fig. 6.13(a). Corresponding UDP throughput *decays* with increasing number of links when using RED, see Fig. 6.13(b). The results here are generally consistent with those in [92, Fig. 8]. In FRED, UDP

[6] Recall from Sec. 2.4.2.1 that `adaptive` and `gentle` parameters can improve RED performance.

[7] We use the approximation: $(1 - \frac{1}{C})^C \simeq e^{-1}$. $w_q \approx 1/C$ for all realistic cases since C is usually large enough. Following this approximation, RED's *avg* computation can then be simplified as: $avg = (1 - w_q)avg + w_q q = ((C - 1)avg + q)/C$.



(a) Flow 0 is TCP.

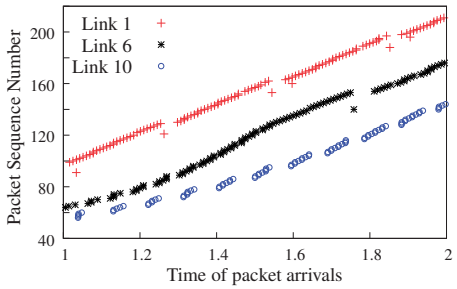
(b) Flow 0 is UDP with rate 1 Mbps.

Figure 6.13.: How flow throughput scales with number of congested links.

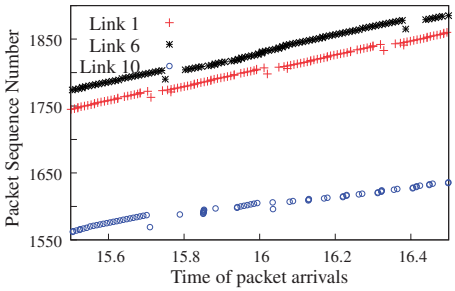
throughput decreases steadily with the number of congested links. CSFQ scales much better than both RED and FRED: It provides a stable throughput share to UDP with increasing links, and a slowly decreasing share to TCP. The only exception to the usual trend of throughput decline is AFpFT which provides reasonably fair share to the flow no matter what the kind of flow traffic or the number of links traversed. A very important, but seemingly discrepant, observation is the fact that TCP flow throughput slightly increases with L in AFpFT. This non-intuitive observation is in sharp contrast to all other schemes, and can be explained as follows. For small L, due to smaller propagation delays, many packets may arrive before previous packets of the flow have left the queue. The flow becomes listed in flow list $\mathcal{FL}$, and the arriving packets may be queued at or near the queue tail. This means there is a higher probability of packet drops and/or higher queueing delays. This in turn causes timeouts and slower TCP sending rates. For large propagation delays (i.e., multiple congested links), it is the opposite: When packets arrive, there may be no previous packets in the queue. Such packets use cheap tagging (see Sec 6.3.1.1) and are pushed to the front of the queue as soon as they arrive. The overall result is fewer packet drops, smaller average queueing delay, and higher flow throughput.

We verify the above observation by tracking TCP packet arrivals and departures at the R1-R2 link (Fig 6.8(b)). Three case studies are provided: when the total number of congested links L are 1, 6 and 10 (see Fig 6.14). For all cases of L, we consider the TCP flow packets on the link R1-R2.

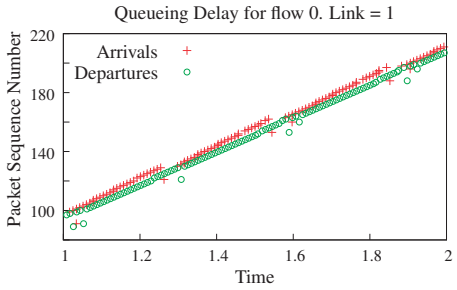
Fig. 6.14(a) shows that TCP packets arrive frequently when $L = 1$, since the propagation delay is the smallest of all cases. When the traffic destination is 10 multiple congested links away, windows of TCP packet arrive slowly and are distinctly separated by time delay. In $L = 1$ case, packets arrive before the previous packets leave the buffer (see Fig. 6.14(c)). The flow becomes listed and arriving



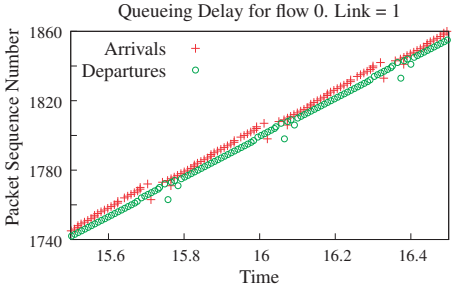
(a) Initial TCP flow arrival rate on R1-R2.



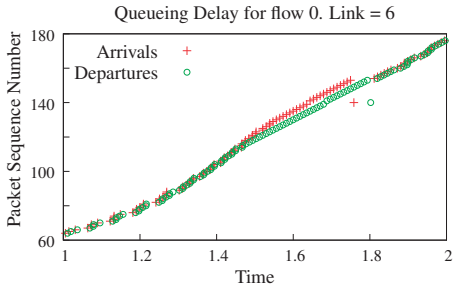
(b) Steady state TCP flow arrival on R1-R2.



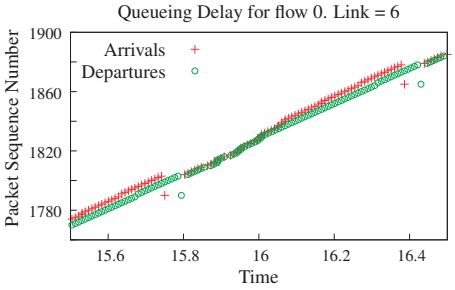
(c) Queuing delay $L = 1$ (initially).



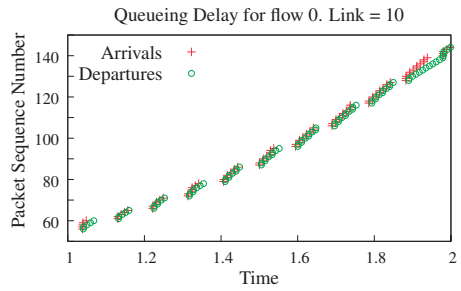
(d) Queuing delay $L = 1$ (steady state).



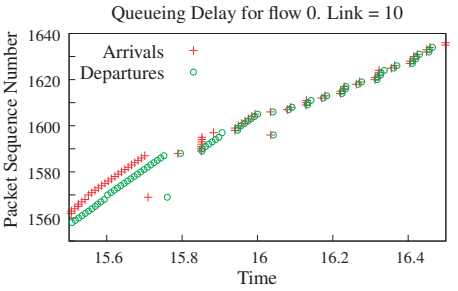
(e) Queuing delay $L = 6$ (initially).



(f) Queuing delay $L = 6$ (steady state).



(g) Queuing delay $L = 10$ (initially).



(h) Queuing delay $L = 10$ (steady state).

Figure 6.14.: Study of TCP flow under multiple congested links.

packets get queued near the tail, and therefore both the average queueing delay and packet drops are the highest for $L = 1$. For our simulation run of 50s: the average queueing delays are 40ms ($L = 1$), 24ms ($L = 6$), 17ms ($L = 10$) and packet drops are 300, 85 and 55 respectively. As can be seen in Fig. 6.14(g),(h), most TCP packets leave the queue as soon as they arrive for $L = 10$. Even though the flow for $L = 10$ is not listed most of the time, TCP throughput still suffers because of large propagation delays, or equivalently large round-trip delays. That means, TCP throughput share when $L = 10$ (not shown) is less than the TCP throughput share when $L = 6$.

When $L = 6$, on the other hand, the flow has moderate round-trip delay. This occasionally allows flow's previous packets to empty the queue before new flow's packets arrive (see Fig. 6.14(f)). When such packets arrive, they are queued near the front of the queue and are scheduled for transmission immediately.

6.4.5. Other Traffic Models

6.4.5.1. Web Traffic Model.

We consider performance of Web traffic which forms the most dominant portion of Internet traffic. Such flows are typically short-lived and often end up during the slow start phase. We model such traffic as a Poisson arrival process with an average of 25 new sessions per second, and the size of each session (file) Pareto distributed with average size of 30kB (about 30 packets) and shape parameter 1.3. The model captures the heavy-tailed (highly variable) nature of Web file sizes and their transmission times [19]. Session statistics such as mean transfer times are important for such flows. We simulate the flows together with a 5Mbps CBR flow under a dumbbell topology: 10Mbps, 1ms link with buffer size of 100 packets. The results are summarized in Table 6.2.

AFpFT performs the best in terms of fulfilling the short transfer demands of *mice* flows. Half of the web flows finish the transfers under 50ms. The mean transfer times of flows are 640ms (RED), 150 ms (FRED), 220ms (CSFQ) and 110ms (AFpFT). A flow, on the average, completes its web transfer twice as fast in AFpFT as in CSFQ. In all schemes, over a thousand flows have completed their transfers within 50s of simulation. The number of flows that complete transfers is larger in AFpFT than in any other scheme.

6.4.5.2. ON-OFF Traffic Model.

The bottleneck is now used by $N - 1$ CBR sources sending at the fair rate, and 1 bursty ON-OFF source. We choose $N = 20$. The ON and OFF periods are taken from exponential distributions with means of 0.2s and $(N - 1) \times 0.2s = 19 \times 0.2s$,

Table 6.2.: Web Session Statistics under Different Router Schemes.

Scheme	Percentage of Flows With Transfer Times					
	< 0.05s	< 0.5s ≥ 0.05	< 1.0s ≥ 0.5	< 2.0s ≥ 1.0	< 5.0s ≥ 2.0	≥ 5.0s
RED	6.70	64.75	16.00	6.50	4.80	1.30
FRED	26.90	69.50	2.28	0.89	0.33	0.09
CSFQ	24.60	67.74	5.00	1.50	0.97	0.16
AFpFT	50.20	47.23	1.46	0.74	0.28	0.08

respectively. During ON period, the ON-OFF source sends at full link capacity of 10Mbps, making it highly bursty. Then it goes idle during the OFF period. ON-OFF sources are normally challenging for AFpFT because at the start of an ON period a packet potentially arrives after packets of previous ON periods have left the buffer. Our interest here is how well the algorithms can restrict this bursty source. Of all packets sent by the ON-OFF source, 92% (RED), 22% (FRED), 28% (CSFQ) and 21% (AFpFT) have been delivered. The result confirms that AFpFT matches FRED in restricting the bursty ON-OFF source.

6.5. Discussion and Related Works

One major motivation for this work is the widespread consensus that *perflow fair queueing algorithms are both stateful and complex*. We believe that AFpFT is relatively simpler and more scalable as it is based on a single aggregate queue, and its design lacks the complex buffer partitioning and associated dynamic scheduling states. AFpFT's packet transmission is a very simple operation of dequeueing the packet at the head of the queue. The only complex operation seems to be the sorting of packets upon their arrival. But the complexity of this operation, and also the amount of flow-level state, is limited by the buffer (backlog) size. Note that the buffer size is a trade-off between performance and complexity. With increasing buffer sizes, AFpFT closely approaches the perflow queueing mechanisms in amount of flow state, operational complexity and quality of flow fairness. Nevertheless, in a more mature form of AFpFT, several optimizations can be applied to simplify the packet sort operation. For instance, since we have two kinds of tag computation, we can define two queues: one for the cheaply tagged packets and another one for the expensively tagged packets. The former is apparently a FIFO queue having a higher priority than the latter. The sort operation is then restricted only to the latter queue. We discuss other optimizing enhancements shortly.

Recent studies on real network traces and analysis based on the statistical nature of Internet flows, e.g., [60], have contradicted the earlier claims of complexity associated with perflow fair queueing algorithms. Accordingly, even though the

number of flows in progress can be very high at a router, the actual number of flows requiring buffering or scheduling (i.e., active flows) is limited to few hundreds both at very high-speed and residential links. Still, implementations may require that the buffer space be partitioned among the active flows or that each active flow be assigned a physical queue. An example of perflow fair queueing proposed for high-speed implementation is [47] which adopts the latter approach: It dynamically maps a physical queue to an active flow. In light of the finding above that "only few flows are active and need buffering", AFpFT and statelet fair queues in general, rather than being dismissed, can be regarded as simple alternative implementations of perflow fair queueing algorithms operating on a small buffer size.

AFpFT sorts packets on arrival and always serves the packet at the head of the line. This kind of queue is not entirely new to our work. It has also been adopted in the Cross-protect architecture [61] which is probably the most related to this work. Cross-protect eliminates the explicit signaling of IntServ and packet marking of DiffServ in order to provide implicit service differentiation between best-effort and streaming flows. It has two mutually complementary functional components: perflow fair queueing and implicit admission control. The perflow fair queueing scheme adopted, called Priority Fair Queueing (PFQ), is an enhanced version of the SFQ algorithm. PFQ gives higher priority to packets if they are of smaller sizes or if their corresponding flows have incoming rates less than the fair share rate. This arrangement allows for streaming flows and lower rate elastic flows to obtain better QoS implicitly without signaling or packet marking. There are several similarities and differences between AFpFT and Cross-protect. We outline apparent similarities first: both can use SFQ for tagging packets; both maintain a list of active flows; both use similar type of sorting queues as mentioned above. One of the differences between AFpFT and Cross-protect is that packet dropping in the latter is Longest Queue Drop (LQD). The active flow list in Cross-protect maintains the flow backlog field. From this information, it is possible to identify the most backlogged flow from which to drop a packet. Another difference is how implicit service differentiation is realized. Cross-protect assigns higher priority for lower rate flows and explicitly to small-sized packets; AFpFT assigns cheap tagging to packets coming from the lower rate flows and this allows the packets to jump closer to the queue head.

Cross-protect is highly optimized. The implicit admission control enables the architecture to be scalable, stable and provides for service guarantees. In addition, the flow list is also protected from exhaustion by some probabilistic insertion of new flows to the list. Some of these optimizations can directly be borrowed to produce a more mature form of AFpFT. For example, listing flows (into $\mathcal{FL}$) based on a probabilistic criterion can be useful to simplify the per-packet operation and alleviate flow list exhaustion. In this regard, when packets of a newly started flow arrive to a queue, for example, we directly apply cheap tagging to those packets, and we list the flow into $\mathcal{FL}$ only with a small probability, say p. The majority of small flows are then scheduled without being entered into the flow list. Such

simplifying optimizations based on probabilities or sampling frequencies are also adopted in a number of other schemes in the literature [74, 76].

6.6. Conclusion

Designing core-stateless versions of fair perflow fair queueing is not an easy task [54]. One hurdle is that computation of perflow parameters is dependent on other interacting flows and it is generally not possible to determine these parameters at network edges. The goal of AFpFT is to approximate the fairness of per flow fair queueing algorithms with minimum states possible. Unlike some of the existing works [64, 66], we make no assumptions about the kind of traffic in the Internet, nor do we explicitly configure routers as core or edge as in [92]. Where flows enter the network, the first router acts as edge and keeps the flow state. This is generally a sound assumption since edge routers manage fewer flows and, being closest to the traffic sources, are ideally suited to provide flow level fairness. Inside the network where there are many more flows, we manage only a subset of those flows that have relatively high rates. Following the heavy-tailed Internet flow distribution, this subset is generally manageable in number. The state requirement in the core network can therefore be limited (from above by the buffer size). Extensive simulations show that the fairness performance of AFpFT is superior to related schemes such as CSFQ, RED, and FRED.

Chapter 7 continues the evaluation of AFpFT in enforcing flow fairness in the face of heterogeneity in TCP congestion avoidance algorithms at the traffic sources.

7. Embracing TCP Heterogeneity using Queue Mechanisms

A large portion of our work so far in this thesis is concerned with how router algorithms in the literature—including those proposed in this thesis—fare in terms of ensuring fairness among TCP flows or protecting TCP flows against unresponsive flows lacking congestion control (e.g., UDP flows). The considered TCP flows adopt the Additive-Increase-Multiplicative-Decrease (AIMD) algorithm [48]. During the last decade, however, the Internet has continuously been embracing a tremendous amount of heterogeneity in deployed TCP congestion avoidance algorithms. Having no standards to follow, these newer algorithms are ad hoc implementations adopted by various operating system vendors. This begs the question of whether these heterogeneous TCP algorithms are fair or compatible both to each other and to the classical TCP algorithm. The objective of this chapter is to investigate the fairness among commonly deployed TCP variants in the presence of several well-known queue management (QM) schemes at the bottleneck. Our simulation results show that most of the TCP algorithms are surprisingly highly unfair to each other or to the traditional TCP algorithm under several queue management schemes. The AFpFT scheme presented in Chapter 6, however, helps battle the TCP heterogeneity and enforce fairness among the various TCP variants considered.

This chapter has six sections. Sec. 7.1 introduces the unfairness problem among TCP algorithms and motivates the problem with an illustrative example, followed by an overview of the related works in Sec. 7.2. Sec. 7.3 provides some background on the various TCP congestion avoidance algorithms considered in this work. Our study is based on extensive simulation experiments, and Sec. 7.4 is devoted to clarifying the simulation environment and the default parameters of the router mechanisms used in the study, before presenting our results in Sec. 7.5. Performance of six different TCP congestion avoidance algorithms (traditional AIMD, Vegas, HSTCP, CTCP, BIC and CUBIC) are considered under two general scenarios. The first scenario compares the TCP friendliness of the various TCP algorithms, and the second scenario studies the full coexistence of all the TCP algorithms operating together. Our closing remarks are presented in Sec. 7.6.

7.1. Introduction

By allowing traffic sources to adapt their sending rates to the network congestion level, the traditional TCP algorithm has contributed greatly to the robustness of the Internet. Its window increase-decrease function is governed by AIMD(1, $\frac{1}{2}$) [48]. That means, TCP increases its sending window by 1 TCP segment size following a loss-free round-trip-time, but cuts its sending window by half upon a packet loss. This traditional TCP algorithm has been standardized by IETF and widely deployed at end hosts. Hereafter, we refer to this type of TCP as the Standard TCP, see Sec. 7.3. Due to its ubiquitous popularity, the Standard TCP algorithm has also shaped the design of several queue management mechanisms, such as RED [10, 38, 36], FRED [64], RED-PD [66], and SRED [74]. Refer to Sec. 2.4.2 and Sec. 4.8. Such mechanisms are often designed to be *TCP-aware* since their packet drop functions are optimized to the Standard TCP. That is, the drop policies implicitly assume that the flows traversing the routers conform to the Standard TCP and react to packet losses as such. For example, the packet drop principle of RED-PD recognizes that a packet drop does not merely result in the loss of the packet, but also abruptly reduces the TCP window by half.

Today, the Internet is no longer controlled by a single TCP algorithm. Due to its failure to scale and adapt, the Standard TCP congestion control algorithm is losing its default status in many of the newer operating systems designed for high speed environments.[1] The Standard TCP's window increase of 1 TCP segment per round-trip-time (RTT) is deemed too conservative to efficiently utilize the vast link capacities available in such networks, and especially so when the RTTs are large. In order to address this scalability problem, the research community responded with several high-speed variants of TCP such as HighSpeed TCP (HSTCP) [32], Scalable TCP (STCP) [56], Binary Increase Congestion Control (BIC) [100], CUBIC [45], Compound TCP (CTCP) [95], eXplicit Congestion control Protocol (XCP) [52] and FAST TCP [51]. These proposals claim to operate just like the Standard TCP in low and medium speed environments, to be compatible or fair to legacy TCP flows. When the window reaches a certain threshold, called the *low window*, they switch to their "scalable modes" whereby the window increases per RTT become significantly larger than 1. Little attention is paid to how these various proposed scalable TCP variants interact in networks [79], and how they behave in the presence of queue management algorithms designed with Standard TCP in mind.

In this chapter, we set out to study a rarely addressed problem: the *TCP inter-protocol* flow compatibility in networks.[2] Specifically, **how do the new congestion control schemes interact with Standard TCP and with each other in the presence of (i) Drop-Tail routers, or (ii) other queue schemes (that are normally designed for Standard TCP)?** End users have freely adopted

[1] Sample default TCP deployments are BIC and CUBIC in Linux distributions [2], and CTCP in some Windows operating systems.

[2] The general problem has recently generated a lot of interest on the end2end mailing list [67].

various end-to-end congestion control algorithms. In fact, a recent measurement study shows that only a small percentage of popular Web servers deploy the Standard TCP algorithms [102].[3] Due to the heterogeneity in end-to-end algorithms and the potential for incompatibility, the mandate of ensuring equitable sharing of scarce network resources may routinely fall on the network provider. This is usually accomplished by enabling the right queue mechanisms along the user's flow path. Since the drop policies of some classical queue management schemes are optimized to the Standard TCP, it is interesting to understand how the queue management schemes cope in the presence of heterogeneous TCP congestion control.

We motivate the TCP protocol incompatibility problem with a simple example. Fig. 7.1 depicts the bandwidth shares of six distinct TCP flows (i.e., Standard TCP, Vegas, HSTCP, CTCP, BIC and CUBIC) competing on a 500Mbps Drop-Tail bottleneck link, with a propagation delay of 10ms. We employed the Linux implementations [99] of the TCP variants in a 200-second ns-2 simulation. Significant Web and long-lived TCP flows were introduced as background traffic in both forward and reverse directions, to avoid phase effects [40]. As can be seen, current Internet routers may not have the required mechanism to enforce protection to competing flows, even when the flows are equipped with end-to-end congestion control algorithms. In this scenario, CUBIC and CTCP look compatible or fair to the Standard TCP. However, BIC obtains an average throughput of over 180 Mbps, which is more than the sum of the throughput of all the other five flows, or more than 50 times than the throughput of Vegas (not shown).

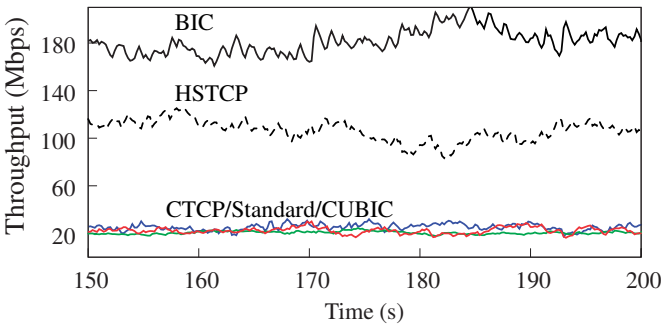


Figure 7.1.: Unfairness among different TCP congestion control algorithms.

7.2. Related Work

Most analytic works on high-speed TCP throughput are restricted to Drop-Tail routers where synchronous loss events are common [32, 100]. Along this line, the

[3] Approx. 45% of Web servers adopt BIC / CUBIC.

steady-state average TCP window W is formulated by the TCP response function

$$W = \frac{c}{p^d} \tag{7.1}$$

where c and d are protocol constants, and p is the loss event rate or probability. See plots of TCP response functions for selected TCP algorithms in Fig. 7.2. Recall from Eq. (2.4) that for Standard TCP $c = \sqrt{3/2}$ and $d = 1/2$. And for HSTCP, $c = 0.12$ and $d = 0.835$.

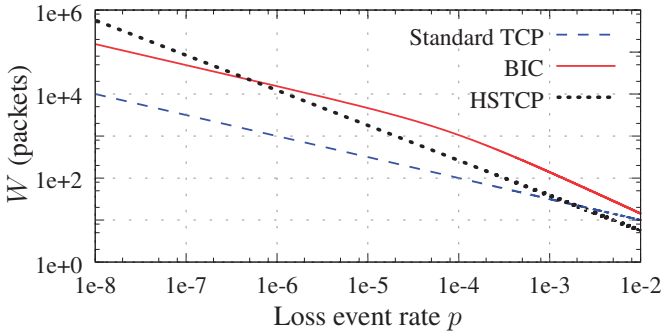


Figure 7.2.: TCP response functions. For BIC, $\beta = \frac{1}{8}$, $S_{\max} = 32$, $S_{\min} = 0.01$.

The original works of the high-speed TCPs [100, 95, 45] focus on the proposed protocol's (i) scalability, that is efficient use of the high link capacities available, (ii) TCP compatibility, that is whether they are fair to Standard TCP in medium or low speed environments, and (iii) intra-protocol RTT-fairness in the presence of *Drop-Tail* routers.

Though important for understanding the stability of the Internet, the study of inter-protocol fairness among high-speed TCP algorithms has received little attention, see [70] for survey-style review of such works. Often, the inter-protocol evaluation works consider a mix of just two high-speed protocols operating in Drop-Tail networks. To the best of our knowledge, there seems to be no comprehensive inter-protocol throughput performance study in the presence of other QM schemes. A recent exception is [101] which is an experimental evaluation of high-speed TCPs over certain queue management schemes operating in 10Gbps networks. The study is limited and does not, for example, consider CTCP and BIC which are two popular Web server deployments today [102]. In this chapter, the aim is to bridge the gap by studying whether the high-speed TCPs (1) are fair to the Standard TCP, (2) can coexist in the presence of the incumbent scheme, i.e., Drop-Tail, and other QM schemes that are probably proposed to be (Standard) TCP-aware. In a broader sense, the importance of the study is to (1) understand the stability of the Internet which evolves with an increasing heterogeneity of the deployed TCP algorithms, and (2) question the suitability of existing QM schemes in embracing the changing dynamic of Internet traffic.

7.3. Background

TCP congestion control is complex and consists of, among others, the slow start mechanism, congestion avoidance mechanism and loss recovery mechanism. Typical examples of loss recovery mechanisms are Reno [4], NewReno [37, 46] and SACK [68]. Examples of congestion avoidance algorithms are AIMD [48], Vegas [12] and BIC [100]. By diverse combinations of the constituent mechanisms, different flavors of TCP congestion control algorithms can be created. These TCP congestion control algorithms are customarily named after their loss recovery mechanism or the congestion avoidance algorithm. By Standard TCP in this chapter, we mean TCP adopting AIMD$(1,\frac{1}{2})$ and SACK as its congestion avoidance and loss recovery mechanisms, respectively. The focus of this section is the description of the various congestion avoidance algorithms compared in this work.

AIMD(α, β)

TCP's window is characterized by a linear increase and an exponential decrease,

$$W = \begin{cases} W + \alpha(W) & \text{// linear increase upon no packet loss} \\ W \times (1 - \beta(W)) & \text{// multiplicative decrease upon a packet loss} \end{cases}$$

Examples of AIMD schemes are Standard TCP and HSTCP. For Standard TCP, $\alpha(W) = \alpha = 1$ and $\beta(W) = \beta = 0.5$. That means, the window increases by 1 segment when there is no packet loss in the previous round, and cut by half otherwise. In HSTCP, however, $\alpha(W)$ and $\beta(W)$ are taken from a generated table of values based on the current W. Generally, as W increases above 16, $\beta(W)$ decreases from 0.5 towards 0.1, but $\alpha(W)$ increases from 1 to over 70 [32]. Example: Let a packet loss is detected when $W = 75,000$. Standard TCP responds by halving the window to $W_{std} = 75,000(1-\beta) = 75,000(1-0.5) = 37,500$, while HSTCP responds by $W_{hstcp} = 75,000(1-\beta(W)) = 75,000(1-0.1) = 67,500$ packets. However, if no packet loss (or "congestion") is detected in the previous round-trip-time, $W_{std} = 75,001$ and $W_{hstcp} = 75,000 + 69 = 75,069$. As a result, Standard TCP is very conservative in window increases whereas HSTCP is suited for high capacity links.

BIC and CUBIC

BIC window growth function is marked by binary search increases and additive increases. In order to understand BIC further, let us exemplify with a packet loss. Before a packet loss, the advertised window size is W_{max}, and the new window after the loss (or multiplicative decrease), is W_{min} which equals $(1-\beta)W_{max}$. BIC finds the midpoint between W_{min} and W_{max}. If the distance from the current / minimum window W_{min} to midpoint is larger than a configurable parameter S_{max},

the window increases additively by S_{max}. Otherwise, the window jumps to the midpoint. If there is no loss after attaining the new window, W_{min} takes on the current window. If there is a loss, however, the window before loss becomes the new W_{max} and the window after loss becomes W_{min}. This window function continues until the distance to midpoint falls below another parameter S_{min}.

CUBIC is proposed as an improvement over BIC. Its window is a cubic function of T or the time elapsed since the last loss event,

$$W = C(T - K)^3 + W_{max} \tag{7.2}$$

where C is a constant, $K = \sqrt[3]{\frac{W_{max}\beta}{C}}$, and W_{max} is—as in BIC—the window size just before the last multiplicative reduction. This real-time window increase of CUBIC allows for better TCP friendliness and RTT-fairness in low speed or short RTT situations.

Vegas

Note that AIMD, BIC and CUBIC are all *loss-based* approaches. That means, the network congestion is implicitly signaled to the TCP sources through the loss of packets. Vegas is a *delay-based* approach. It samples RTT variances to deduce network congestion. An increased round-trip-delay indicates congestion and is followed by a linear reduction of the window, and vice versa. If a stable RTT is maintained, the window size remains unchanged.

Vegas has better RTT-fairness than the Standard TCP, and yields better throughput in the absence of competition from the loss-based TCP flows.

CTCP

This is a synergy of loss and delay based approaches, or equivalently, the Standard TCP coupled with a scalable delay based component. The delay based component makes CTCP protocol RTT-fair and efficiently scale to large bandwidth. Just like in Vegas, the delay component senses the underlying network load from packet delay samples. If the network is not congested, the delay component quickly enables the flow to ramp up its sending rate or window.

7.4. Simulation Environment

This section outlines the simulation environment and default configurations used in this chapter. The system topology is shown in Fig. 7.3. There are N TCP sources competing over a bottleneck link of capacity $C = 1000$Mbps, buffer size B, and a

bottleneck link propagation delay D. Unless stated otherwise, B is dimensioned as $2 \times C \times D$, which is less than 100% of the bandwidth-delay product. We used the ns2 implementations of Standard TCP, Vegas, HSTCP, CTCP, BIC, and CUBIC ported from Linux [99] and without changing the default parameters. Since most end users have neither the expertise nor the desire to change TCP protocol parameters, deployments often operate with their default parameter values.

The bottleneck link implements one of the five schemes Drop-Tail, RED [10, 38, 36], CHOKe [78], FRED [64], and AFpFT presented in Chapter 6. Drop-Tail and RED are presented as baseline cases. CHOKe and FRED are extensions of the RED queue with differential flow drop rates. While CHOKe is completely stateless, FRED and AFpFT are statelet.

In order to avoid phase effects [40] in simulations, we introduce significant Web and Standard TCP traffic (with advertised windows ≤ 50) in both directions. For example, the background traffic constitutes about 11-18% of the link capacity for Scenario 1 experiments. The background traffic provide competition with TCP data flows in the forward direction, and with ACKs in the reverse direction. Default queue configurations are as follows. For RED, CHOKe and FRED: $\min_{th} = B/4$, and $\max_{th} = 3B/4$. Recall from Table 3.2 and the ensuing discussion that adjusting $\max_{th}$ and $\min_{th}$ to the available buffer size helps improve the throughput performance of Adaptive RED. For RED and the RED component embedded in CHOKe, both *gentle* [30] and *adaptive* [36] parameters are enabled; for FRED, queue averaging constant $w_q = 1/C$, see Sec. 6.4.3.

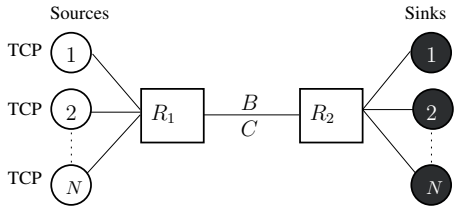


Figure 7.3.: System model.

7.5. Evaluation and Results

Our major performance metric in this study is flow *fairness* as quantified by Jain's fairness index [49] (see Sec. 2.2.4). Additional performance metrics is the *link efficiency* as measured by the total link utilization.

We conduct extensive experiments using two general scenarios. Scenario 1 investigates TCP friendliness, or whether or not the various TCP types can be fair to competing Standard TCP flows and is presented in Sec. 7.5.1. Scenario 2 considers the coexistence of all TCP types, and is presented in Sec. 7.5.2. Both scenarios are

carried out in the presence of different QM schemes at the bottleneck. In general, for a given QM scheme, we compare the inter-protocol fairness among the TCP variants when the flows experience similar and different round-trip-times, and bottleneck buffer capacities. The first 100 seconds were not considered when reporting steady-state results.

7.5.1. Scenario 1: TCP Friendliness

This scenario investigates the *compatibility of different congestion control algorithms with Standard TCP*. In each experiment, there are $N = 6$ TCP flows competing on the bottleneck link; three flows are of type Standard TCP and the remaining three belong to "Other" TCP flows of the same type, e.g., Vegas. The flows have three different round-trip-times, labeled as rtt_i, $i \in \{1, 2, 3\}$. Disregarding the queueing delays, the RTTs are $rtt_1 = 60$ms, $rtt_2 = 120$ms, $rtt_3 = 240$ms. One flow each from Standard TCP and the "Other" TCP has an RTT of rtt_i. That is, Standard-1 and "Other"-1 have rtt_1, and Standard-2 and "Other"-2 have rtt_2, and Standard-3 and "Other"-3 have rtt_3.

Figure 7.4 shows the result of the 25 experiments (five per QM scheme), averaged over 10 replications. A cluster of 3 stacked bars represent one experiment. "Standard" represents the Standard TCP. The type of the "Other" TCP is indicated in the x-axis, and so are the respective flow RTTs. For example, the first bar in Fig. 7.4(a) shows the throughput obtained (under Drop-Tail bottleneck queue) by the Standard (solid fill) and Vegas flows having rtt_1. Tables 7.1 and 7.2, respectively, show the Jain's fairness indices and the total link utilizations for the 25 experiments. From the tables and Fig. 7.4, the following key observations can be made (suffixing the flows with $-k$ to designate their rtt_k).

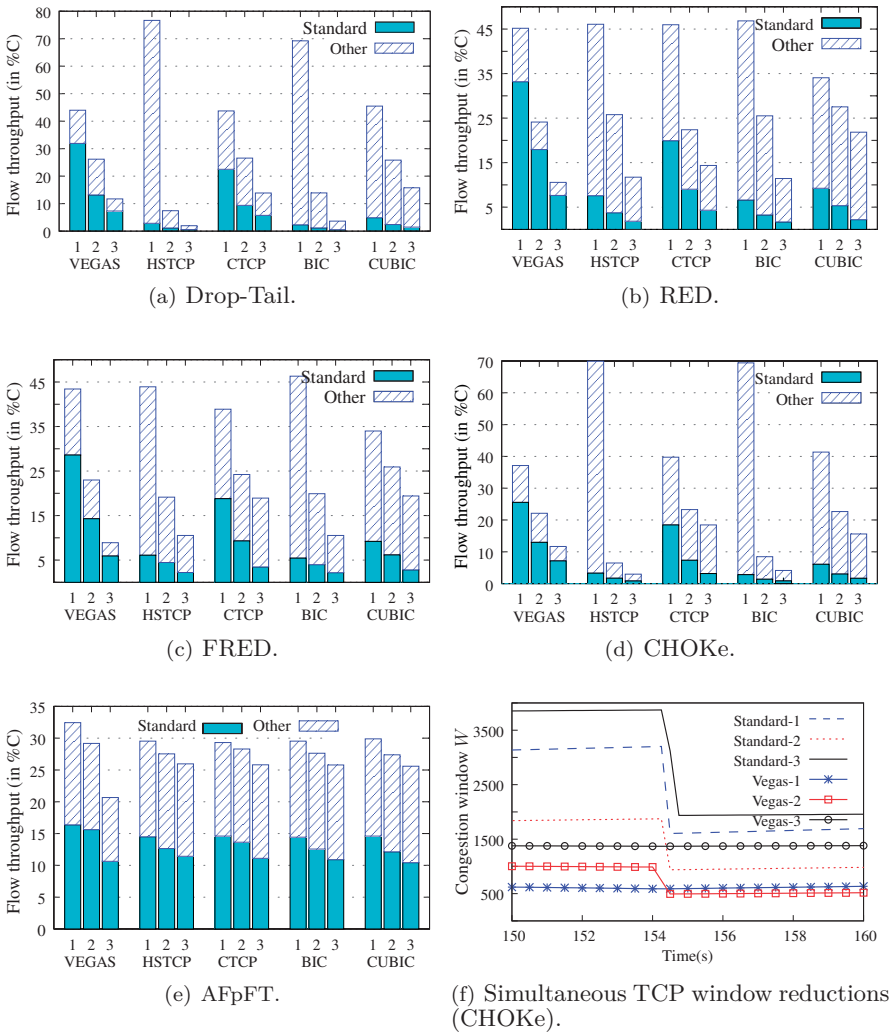
Table 7.1.: Fairness indices of 25 experiments.

"Other"	Scheme				
TCP	Drop-Tail	RED	FRED	CHOKe	AFpFT
Vegas	0.71	0.64	0.69	0.75	0.97
HSTCP	0.22	0.54	0.51	0.23	0.99
CTCP	0.82	0.78	0.85	0.82	0.99
BIC	0.27	0.52	0.48	0.25	0.99
CUBIC	0.52	0.72	0.74	0.56	0.98

7.5.1.1. Impact of QM schemes

AFpFT achieves the most superior fairness, regardless of the competing TCP types and their RTTs, without sacrificing the link utilization. By contrast, Drop-Tail and CHOKe have poor fairness. Surprisingly, FRED has only marginally better fairness

than RED. Even when all flows are of the Standard TCP type (result not shown), the fairness in FRED is just comparable to that of Drop-Tail. This observation is similar to the one in Fig. 6.12(a).



(a) Drop-Tail. (b) RED.

(c) FRED. (d) CHOKe.

(e) AFpFT. (f) Simultaneous TCP window reductions (CHOKe).

Figure 7.4.: Fairness among various TCP congestion avoidance schemes in the presence of Drop-Tail, RED, FRED, CHOKe or AFpFT at the bottleneck. (a)-(e) compare the flow bandwidth shares of Standard TCP with those of the indicated TCP type. (f) demonstrates the synchronization of packet losses among many TCP flows under CHOKe.

Table 7.2.: Link utilization in (%) of the 25 experiments.

Other TCP	Scheme				
	Drop-Tail	RED	FRED	CHOKe	AFpFT
Vegas	97	97	93	89	99
HSTCP	99	99	92	96	99
CTCP	99	99	99	98	99
BIC	99	99	95	98	99
CUBIC	99	99	97	97	99

7.5.1.2. TCP-compatibility

CTCP looks the most fair to Standard TCP, across all classic QM schemes. We conjecture that the dual nature of CTCP as loss and delay based scheme offers this compatibility benefit. On the other hand, HSTCP and BIC are the most unfair to Standard TCP. The HSTCP-1 and BIC-1 flows seem to overly dominate the link and starve out the legacy TCP traffic.

7.5.1.3. RTT-fairness

HSTCP and BIC have both very poor intra-protocol RTT-fairness. Except under AFpFT, HSTCP-1 and BIC-1 receive between 4-50 and 4-20 times the throughput of HSTCP-3 and BIC-3, respectively. The worst cases are observed under Drop-Tail and CHOKe. HSTCP and BIC window increases are inversely related to their flow RTTs. At small RTTs, the windows grow faster and quickly reach the *low window* after which point the window increases are more than 1 segment per RTT, aggravating the unfairness. The most RTT-fair high-speed protocols are CTCP and CUBIC. CTCP-1 receives around 1.3-2.6 times that of CTCP-3, and CUBIC-1 gets 1.3-3 times of CUBIC-3. Similarly, Vegas-1 throughput is around 2.5-5 times than that of Vegas-3.

7.5.1.4. Interaction between QM and TCP algorithms

CTCP and Standard TCP seem to interact very well under all QM mechanisms, resulting in both high fairness and link utilization. However, HSTCP + Standard and BIC + Standard combinations under Drop-Tail and CHOKe are literally broken as shared networks. The respective Drop-Tail and CHOKe fairness scores are low (less than 0.3). Under Drop-Tail, for example, the HSTCP-1 and BIC-1 flows obtain whopping link shares of 74% and 67%! And Standard-3 in those systems receive less than 1.0% of the link.[4] Indeed, the unfairness between Standard TCP

[4]Corresponding figures in AFpFT are around 11%.

and HSTCP or Standard TCP and BIC is even worse than the classical incompatibility of Standard TCP and Vegas flows under Drop-Tail. In the Standard+Vegas combination under Drop-Tail, the Standard-1 grabs 32% of the link capacity (see Fig. 7.4(a)), and Vegas-3 receives 4.4%.

7.5.1.5. Link utilization

AFpFT equals or betters the other mechanisms in link utilization, see Table 7.2. FRED is generally poorer in overall link utilization, probably due to the lack in FRED of *gentle* and *adaptive* performance tuning parameters, as explained in Chapter 6. CHOKe's link utilization in the Standard+Vegas is poor as well. The underlying reason, which is also an important observation on its own, seems to be the apparent synchronization of packet losses among the flows in CHOKe (see Fig. 7.4(f)).[5] Recall that each arriving packet triggers a flow matching trial in CHOKe. When a TCP flow dominates the CHOKe queue, the flow's matching probability, hence the packet loss probability, becomes higher. The loss of a packet is followed by a reduction in the TCP window and draining of its packets from the queue. This in turn escalates the matching / dropping probabilities of the other flow(s) in the queue. While the high-speed TCP variants can instantly compensate with rapid increases of congestion windows, Standard TCP and Vegas do not recover soon enough due to their conservative window increases. This multiple-flow reduction problem in CHOKe looks more severe than the global TCP synchronization problem in Drop-Tail. Note in Fig. 7.4(f) that flows with larger RTTs such as Standard-3 may look to have larger windows W, but not necessarily higher throughput. Average flow throughput is given as W/RTT.

7.5.2. Scenario 2: Full Coexistence

This section investigates the mix of the six different congestion control algorithms. In each experiment of this section, there are six flows—one from each type of TCP—all having similar round-trip-times. The results are shown in Fig. 7.5(a). There are a total of 15 experiments and each stacked bar represents the throughput of the six flows in one experiment. For each experiment, the link delay [ms] and the QM scheme at the bottleneck are clearly marked on the x-axis. For example, the first bar shows the "normalized" TCP throughput shares of the six flows, all traversing a Drop-Tail router and a bottleneck link delay of 10ms. The fairness indices of the 15 experiments are reported in Table 7.3.

The results demonstrate the exceptional fairness qualities of AFpFT, and the poor fairness under CHOKe and Drop-Tail where the BIC and HSTCP flows dominate the links. In general, with large buffer sizes, the fairness of AFpFT is close to that of perflow fair queueing schemes. In all other schemes, Vegas is starved out

[5] Similar to global TCP synchronization common in Drop-Tail routers.

Table 7.3.: Fairness indices of 15 experiments in Scenario 2.

Link delay	Scenario 2 experiments				
	Drop-Tail	RED	FRED	CHOKe	AFpFT
10ms	0.39	0.53	0.61	0.39	1.0
50ms	0.51	0.58	0.65	0.47	1.0
100ms	0.56	0.54	0.61	0.53	1.0

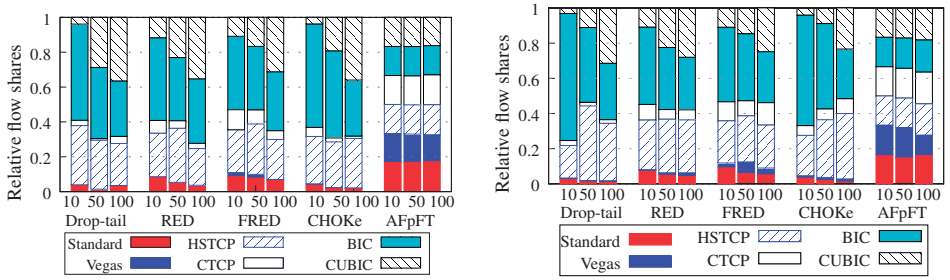


Figure 7.5.: The coexistence of the six TCP types under bottleneck buffer sizes of (a) bandwidth-delay product, and (b) a fixed 20Mb.

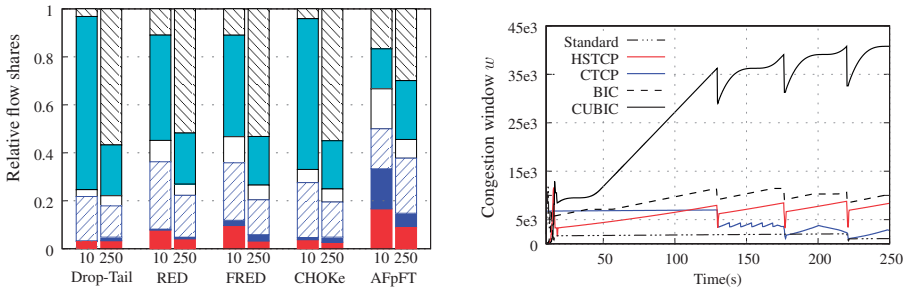
obtaining under 4% of the shares. But the throughput of CTCP once again looks proportional to those of Standard TCP. Surprisingly, CUBIC becomes increasingly aggressive with longer propagation delays, which is generally the opposite situation to that of the Standard TCP. The fairness scores approximate how many flows dominate the link. For example, for both Drop-Tail and CHOKe, the scores improve from 0.39 to more than 0.5 when bottleneck link delay increases from 10ms to 100ms. The is because the number of flows dominating the link has increased from two to three as CUBIC, at higher RTTs, joins the list of dominating flows (BIC and HSTCP). AFpFT aside, the best fairness score seen is a *poor* 0.65, achieved under FRED whose fairness is slightly better than that of RED.

The buffer sizes in the above experiments vary with the bottleneck link delays. As a result, those buffer sizes are probably very excessive. We repeat all of the experiments with fixed buffer sizes set at 20Mb. The results are demonstrated in Fig. 7.5(b). We saw no discrepancy to Fig. 7.5(a), except that the exceptionally high fairness scores of AFpFT start to decline.

As noted earlier, both Figs. 7.5(a) and 7.5(b) demonstrate increasing throughput shares of CUBIC as the bottleneck link delays increase from 10ms towards 100ms. In the rest of this section, we investigate the *increasing aggressiveness of CUBIC when the round-trip-times increase*. We conduct another set of experiments using a bottleneck link delay of 250ms, running for 300 seconds. The Standard TCP based

Table 7.4.: Fairness indices of Scenario 2 experiments with fixed buffer sizes.

Link delay	Scenario 2 experiments				
	Drop-Tail	RED	FRED	CHOKe	AFpFT
10ms	0.30	0.56	0.62	0.37	1.000
50ms	0.45	0.61	0.67	0.47	0.999
100ms	0.54	0.63	0.74	0.60	0.975
250ms	0.43	0.48	0.47	0.45	0.760



(a) Relative throughput shares of TCP flows.

(b) Typical TCP window runs under Drop-Tail and 250ms link delay.

Figure 7.6.: TCP CUBIC can be extremely aggressive in similar high RTT environments

cross-traffic takes up only 2% of the link capacity because of the long link delays. The fairness results are remarkably different; CUBIC turns into the most aggressive among the considered TCP types (see Fig. 7.6). Fig. 7.6(a) contrasts the relative TCP bandwidth shares with those obtained from the 10ms bottleneck delay shown earlier in Fig. 7.5(b). From the figure, it is clear that under all QM schemes except AFpFT, CUBIC bandwidth share equals or is more than the sum of all the five TCP shares. For example, under Drop-Tail, CUBIC share alone constitutes 56% of the whole sum! See also the last row of Table 7.4. The fairness index of AFpFT suffers as CUBIC joins the band of aggressive flows. An important question to pose is whether CUBIC is actually aggressive or merely grabs the otherwise free bandwidth? Another similar experiment without CUBIC demonstrates that the five flows together manage to grab 70% of the full link capacity under Drop-Tail. In the presence of CUBIC, however, the overall share of the five flows shrinks approximately to 40% of the link bandwidth. Even though the presence of CUBIC allows for very high link utilization (e.g., around 94% vs. 70% without), it can also become a bandwidth hog for the other TCP flows.

Next, we look for an empirical reasoning for the increasing CUBIC aggression, focussing on the Drop-Tail bottleneck link. Recall from Eq. 7.2 that the CUBIC window increase function is dependent on the time elapsed since the last packet loss (for clarity, let us call the interval between loss events *inter-loss interval or*

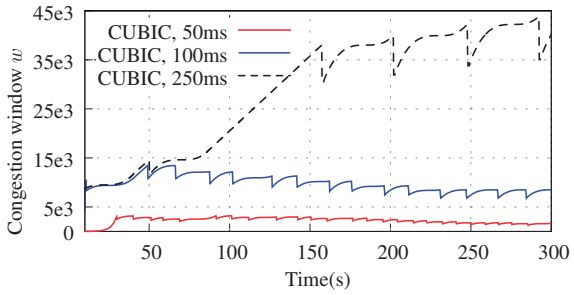


Figure 7.7.: CUBIC window runs under different inter-loss durations. Under homogenously long RTT environments, the inter-loss intervals are longer which in turn enables CUBIC to invoke the aggressive max-probing phase and rapidly ramp up its window.

duration). On the other hand, BIC and HSTCP are RTT-unfair and their window increase functions depend inversely on the RTT. Under short RTTs, BIC and HSTCP can become aggressive, increase their windows and fill up the buffer quickly, and subsequently invoke frequent router actions to drop the packets. The inter-loss intervals are short and CUBIC cannot freely increase its window size. Under long RTT situations, however, BIC and HSTCP window increase functions slow down, packet losses become infrequent, and the interval between loss events become longer on the average (see Fig. 7.7). With its window function being independent of RTT, CUBIC senses unused bandwidth and easily reaches the current W_{max} following Eq. 7.2. If there is no packet loss event at W_{max}, CUBIC probes the network to find a new and higher *maximum window*, W_{max}. This probing phase is called max-probing in [45] and corresponds to strong linear increases just before $t = 130$ seconds in Fig. 7.6(b). We summarize this important finding as follows.

In a network where TCP flows have all long RTTs, BIC and HSTCP connections are relatively conservative in their use of the bandwidth due to the slow window increases. The durations between packet loss events are long, and this enables the CUBIC window to quickly reach the current W_{max}, invoke the max-probing phase and tune to a new higher W_{max}. In short RTT environments, however, the BIC and HSTCP flows are more aggressive, fill the queues up frequently, and cause frequent packet losses. The duration between loss events would become too short for CUBIC to reach its current maximum window and invoke the max-probing phase.

7.6. Concluding Remarks

The Internet is being transformed as new high-speed TCP algorithms are increasingly being deployed. This work makes important contributions to the compatibility study of deployed TCP algorithms. The performance of the common TCP deployments in the presence of various queue management schemes at bottleneck

links is considered. Despite claims of their TCP-friendliness, only CTCP seems to be truly compatible to Standard TCP, and also intra-protocol RTT-fair in the considered network environments. In environments with heterogenous algorithms involving flows of different RTTs, those BIC and HSTCP flows with the least RTTs seem to overly dominate the links. Unfairness is far worse in the current default Drop-Tail queues and as well as in CHOKe queues. If all flows have similar RTTs, the dominant flows will be mostly BIC and HSTCP (if the common RTTs are small), or CUBIC (if RTTs are large). At large common round-trip-times, losses seem synchronized and infrequent and this allows CUBIC to frequently invoke the max-probing phase, and turn increasingly aggressive and dominant. When and to what extent CUBIC becomes aggressive is an interesting future work.

Our results show that the TCP-friendly CTCP flow suffers in the presence of BIC and/or CUBIC flows. With lack of incentives that reward compliance to a standard algorithm, networks can become hotbeds for heterogeneous—yet incompatible—TCP algorithms. Internet users are naturally inclined to adopt aggressive algorithms. This may lead to "arms race" in the development of aggressive algorithms [70]. We show that both the incumbent (i.e., Drop-Tail) and the considered queue schemes cannot provide the required incentives and traffic control.

If future congestion control algorithms and / or queue mechanisms are required to be TCP-compliant or TCP-aware, what particular TCP protocol should be defined as a reference standard is still an open question. Our simulations show that fair queue mechanisms such as AFpFT can facilitate the coexistence of heterogeneous end-to-end algorithms without sacrificing the link utilization or declaring a new TCP reference protocol.

8. Conclusions and Future Work

In this chapter, we conclude the thesis by summarizing our major contributions and pointing out the limitations that can be addressed in future works.

8.1. Summary of Contributions

In Chapter 2, an overview on the two tradeoffs in building router flow fairness mechanisms is given. They are the simplicity in per packet operations and the quality of the flow fairness. One or both of the two features are usually missing in the proposed architectures found in the literature. Existing approaches are either complex and stateful, or lack generality in ensuring flow fairness and protection. This thesis presents the designs and analysis of a host of relatively simpler and efficient router-based flow fairness and flow protection mechanisms. All the mechanisms proposed in the thesis are single aggregate queues shared by all flows.

Chapter 3 introduces an intuitive design and implementation of perflow fair queueing. The scheme is significantly enhanced and generalized as a statelet flow protection framework in Chapter 6. Since this statelet scheme normally maintains state for those flows having packets currently in the queue, the amount of flow state is bounded by the buffer size. The flow state is leveraged in two important functions. First, it is used for service tag computation to arriving packets. Secondly, when the queue overflows and packet(s) are dropped, it allows for the correction of the corresponding flow parameters and the riddance of a potential lockout behavior. The packet transmission is as a simple operation as the FIFO queueing discipline; that is, at each transmission epoch, the packet at the head of the queue is dequeued. A flow fairness comparison of the scheme against popular single-queue schemes in the literature is conducted in the presence of both Standard TCP and unresponsive flows in Chapter 6. Further performance evaluations are presented in Chapter 7 which demonstrate that the scheme is highly fair and efficient in resource utilization despite the heterogeneity in TCP congestion control algorithms at the sources. None of the compared schemes in the literature performs as well.

In discussing flow protection in this thesis, we use the upper bounds in link utilizations and buffer space shares that can be taken up by the aggressive (unresponsive) flows. Chapter 4 presents a suite of active queue management schemes for flow protection, namely geometric CHOKe (gCHOKe), that are completely stateless and simple. gCHOKe schemes are indexed by a configurable integral parameter

`maxcomp` m, as in gCHOKe(m), which caps the number of Bernoulli flow matching trials that can be tried per arriving packet. Flow protection improves with m. The CHOKe scheme turns out to be the simplest of the schemes with $m = 1$. An attractive property is that even for gCHOKe(∞), the additional per packet processing over that of CHOKe is very small whereas its flow protection is significantly better than that of CHOKe. An extremely aggressive flow asymptotically obtains *zero* service under gCHOKe.

Chapter 5 presents the first study that characterizes the "perplexing" transient behaviors of the CHOKe queue following changes in the exogenous rate of the unresponsive flow. When the UDP source rate increases, the transient UDP utilization decreases and vice versa. For a given UDP source rate, the highest transient utilization is achieved when the flow stops or finishes. These intriguing transient queue behaviors cannot readily be explained by existing literature results. By factoring the UDP rate change into the spatial distribution model of CHOKe, we track not only the evolution of the flow utilizations, but also derive the flow utilization (upper or lower) bounds during the transient regime. Extensive simulations verify the model. The model can easily be extended for other gCHOKe variants.

8.2. Future Work

In the rest of this chapter, we expose some of the limitations, focussing on the AFpFT scheme presented in Chapter 6.

Sec. 6.5 points out further optimizations for a more mature form of AFpFT. In general, for common buffer sizes and traffic scenarios under AFpFT, we observe no unfairness problems even under congestion. In the presence of a smaller buffer size, or equivalently a smaller flow list $\mathcal{FL}$ size, and a large number of aggressive high bandwidth flows, however, AFpFT's fairness performance may suffer. This is because some of the high bandwidth flows may escape being listed as their packets are dropped from the small buffer size and the flows' parameters are subsequently reset or removed. When new packets of the flows arrive to queue, they may invoke cheap tagging and get unfair advantage in service. One simple, yet coarse, remedy for this problem is the following. The moment when the flow's packets in the queue reduces to 0 (i.e., $count_f = 0$), we can start a flow timer. The flow parameters are then removed / reset when the timer expires. Therefore, the new arrivals of the flows obtain cheap tagging only if the corresponding flow parameter *count* is zero and the respective flow timer already expires. For new flows, their first packets are guaranteed to be prioritized as before. Yet, what value to set for the timer as default is left as a future work.

A related future work is the study of the spatial queue properties of AFpFT. We expect non-uniform distribution of flows in the queue, much like the CHOKe queue studied in Chapter 5. This is because packets are usually assigned cheap

tagging if they belong to small (often non-bottlenecked) flows, and expensive tagging otherwise. This study requires deriving an analytical model that captures the distributions of flows given, amongst others, the source traffic characteristics (e.g., source rates), the buffer size, and the bottleneck link capacity. Such a study even under static traffic conditions can be useful for several reasons. Firstly, it can be useful for dimensioning the buffer and link capacities so that any potential unfairness situations as described earlier in this section, if any, could be averted. Secondly, the study helps to understand the extent of service priority (scheduling and buffer space) provided to smaller flows.

Apart from Chapter 5, we focussed on the flow fairness performance of queue mechanisms in the steady state. A very interesting future work is the fairness performance of TCP flows during slow start and how the queue mechanisms impact the flow throughput in that regime.

A. Appendix

This appendix is devoted to proving the lemmas and theorems in Sec. 5.4.2. We need the following intermediate results.

From (5.12) and (5.21), respectively, we get

$$\tau^{'}(y) = 1/v(y) \tag{A.1}$$

$$v^{'}(y)/v(y) = \rho_0^{'}(y)/(1-\rho_0(y)). \tag{A.2}$$

Using $\rho_0^{'}(y)$ from (A.2) into (5.22) and solving for $v^{'}(y)$,

$$v^{'}(y) = x_0(1-r)\beta\rho_0(y). \tag{A.3}$$

Proof of Lemma 5.4.1

Proof The following set of equations are trivial and follow from (5.23).

$$x_0(1-r)\beta \int_0^y \frac{1}{v(s)}ds = \int_0^y \frac{\rho_0^{'}(s)}{\rho_0(s)}ds + \int_0^y \frac{\rho_0^{'}(s)}{1-\rho_0(s)}ds$$

$$\begin{aligned}
x_0(1-r)\beta\tau(y) &= \ln(\rho_0(s))|_0^y - \ln(1-\rho_0(s))|_0^y \\
x_0(1-r)\beta\tau(y) &= \ln\left[\frac{\rho_0(y)}{\rho_0(0)}\frac{1-\rho_0(0)}{1-\rho_0(y)}\right] \\
\tau(y) &= \frac{1}{x_0(1-r)\beta}\ln\left[\frac{\rho_0(y)}{1-\rho_0(y)}\frac{1-\rho_0(0)}{\rho_0(0)}\right] \\
\tau(y) &= \frac{1}{x_0(1-r)\beta}\ln\left[a\frac{\rho_0(y)}{1-\rho_0(y)}\right]
\end{aligned} \tag{A.4}$$

From (A.4), $\rho_0(y)$ can be expressed as (5.24). ∎

Proof of Lemma 5.4.3

Proof From (5.10) or (5.2), practical values of h_0 must satisfy $h_0 < 0.5$. For $h_0 = 0$ and $\mu_0 = 0$, the proof is trivial. We only need to prove the bounds of $\frac{1-\mu_0}{1-h_0}$ for $h_0 \in (0, 0.5)$.

We need (5.1) and the well known property of natural logarithms shown next.

$$\ln x \leq x - 1 \qquad \text{for } x > 0 \tag{A.5}$$

The proof is by contradiction and has two parts.

(i) First, we establish that $(1 - \mu_0)/(1 - h_0) \geq 1$. Let us assume:

$$\frac{1-\mu_0}{1-h_0} < 1 \qquad \Rightarrow \qquad h_0 < \mu_0. \tag{A.6}$$

From (A.6) and (5.1), we get

$$h_0 < \frac{\ln[\frac{1-h_0}{1-2h_0}]}{[\frac{1-h_0}{1-2h_0}] + \ln[\frac{1-h_0}{1-2h_0}]} \tag{A.7}$$

Since $0 < h_0 < 0.5$, $(1 - h_0)/(1 - 2h_0) > 1$ and $\ln[(1 - h_0)/(1 - 2h_0)] > 0$. Multiplying both sides of (A.7) by the denominator term found on the r.h.s. and simplifying, we obtain:

$$\frac{h_0}{1-2h_0} < \ln \frac{1-h_0}{1-2h_0}. \tag{A.8}$$

On the other hand, since $\frac{1-h_0}{1-2h_0} > 0$, we can apply property (A.5) on $\frac{1-h_0}{1-2h_0}$ to obtain:

$$\ln \frac{1-h_0}{1-2h_0} \leq \frac{1-h_0}{1-2h_0} - 1 = \frac{h_0}{1-2h_0} \tag{A.9}$$

which is a contradiction to (A.8) . Hence, we prove that $\frac{1-\mu_0}{1-h_0} \geq 1$. $\frac{1-\mu_0}{1-h_0} = 1$ when both $\mu_0,\ h_0 = 0$.

(ii) Here we establish that $(1 - \mu_0)/(1 - h_0) \leq 2$. As before, let us contradict by assuming that,

$$\frac{1-\mu_0}{1-h_0} > 2 \qquad \Rightarrow \qquad \mu_0 < 2h_0 - 1. \tag{A.10}$$

Since $h_0 < 0.5$, (A.10) says that $\mu_0 < 0$, which is not possible. This means the assumption in (A.10) must be wrong, or that $(1 - \mu_0)/(1 - h_0) \leq 2$.

Combining (i) and (ii) completes the proof. ∎

Proof of Lemma 5.4.4

Proof Solving for $\rho_0^{'}(y)$ from (A.2) and (A.3),

$$\rho_0^{'}(y) = x_0(1-r)\beta\left[\frac{\rho_0(y)-\rho_0(y)^2}{v(y)}\right] \tag{A.11}$$

Since $\beta < 0$, and probabilities $\rho_0(y) < 1$, $\rho_0(y)$ is decreasing with y. Taking the differentiation further and using (A.3),(A.11) in place of $v^{'}(y)$ and $\rho_0^{'}(y)$ and simplifying, we obtain

$$\begin{aligned}\rho_0^{''}(y) &= x_0(1-r)\beta\frac{\rho_0^{'}(y)v(y)(1-2\rho_0(y)) - v^{'}(y)\rho_0(y)(1-\rho_0(y))}{v^2(y)} \\ &= x_0^2(1-r)^2\beta^2\frac{\rho_0(y)(1-\rho_0(y))(1-3\rho_0(y))}{v^2(y)}\end{aligned} \tag{A.12}$$

The critical point $(y^*, \rho_0(y^*))$ where $\rho_0^{''}(y) = 0$ is given as,

$$\rho_0^* = \rho_0(y^*) = \frac{1}{3} \tag{A.13}$$

$$y^* = \frac{1}{K}\ln\left(\frac{a}{2}\right) + \frac{1}{K}\frac{1-3\rho_0(0)}{2(1-\rho_0(0))}. \tag{A.14}$$

(A.14) is obtained upon substituting ρ_0^* for $\rho_0(y)$ in (5.26). It is easy to see that $\rho_0(y)$ decreases in concave fashion on $y \in [0, y^*]$ and in convex fashion on $y \in [y^*, b]$. See Fig. 5.7(a) for an example.

Note that the critical point (y^*, ρ_0^*) exists when the UDP arrival rate x_0 exceeds a certain value x_0^*, calculated using (5.16) as,

$$x_0^* = \min\{x_0 \mid \rho_0(0) \geq \rho_0^*\} = \min\left[\frac{C}{2}\frac{1-\mu_0}{1-h_0}\right] = \frac{C}{2}. \tag{A.15}$$

Above, we use Lemma 5.4.3 to state that $\frac{1-\mu_0}{1-h_0} \geq 1$. From (A.15), it follows that when $x_0 \leq C/2$, $\rho_0(y)$ is strictly convex decreasing.

For arrival rate $x_0 \in [C/2, \infty)$, since $\rho_0(0) \geq \rho_o^*$ by (A.15) and $\rho_0(b) := \mu_0 \leq 26.9\% \leq \rho_0^*$ by the Limit property (see Sec. 5.2.2), the critical point exists somewhere $y^* \in [0, b]$. ∎

Proof of Lemma 5.4.5

Proof From (A.3), it is trivial to see that $v(y)$ is decreasing with y since $v^{'}(y) < 0$. Differentiating (A.3) and using (A.11),

$$v^{''}(y) = x_0^2(1-r)^2\beta^2\rho_0(y)\frac{1-\rho_0(y)}{v(y)} \tag{A.16}$$

Since $v''(y) > 0$, $v(y)$ is convex decreasing throughout the queue. See Fig. 5.7(b). ∎

Proof of Lemma 5.4.6

Proof The queueing delay is a strict convex increasing function, since from (A.1) and (A.3), respectively,

$$\tau^{'}(y) = 1/v(y) > 0 \tag{A.17}$$

$$\tau^{''}(y) = -\frac{1}{v^2(y)}v^{'}(y) = -\frac{x_0(1-r)\beta\rho_0(y)}{v^2(y)} > 0 \tag{A.18}$$

∎

Proof of Lemma 5.4.9

Proof Set $\Delta T = \tau(b)$ and $\alpha = 1$, i.e., $x_{02} = x_0$ in Lemma 5.4.7.

$$\mu_0(\tau(b))|_{\alpha=1} = \frac{1}{1+ae^{-x_0\tau(b)\beta}} = \frac{1}{1+ae^{-x_0\tau(b)\ln(1-1/b)}}$$

$$= \frac{1}{1+a(1-1/b)^{-x_0\tau(b)}} \tag{A.19a}$$

$$= \frac{\rho_0(0)}{\rho_0(0)+(1-\rho_0(0))(1-1/b)^{-x_0\tau(b)}} \tag{A.19b}$$

$$= \frac{\rho_0(0)v(0)(1-1/b)^{x_0\tau(b)}}{\rho_0(0)v(0)(1-1/b)^{x_0\tau(b)}+N\rho_1(0)v(0)} \tag{A.19c}$$

$$= \frac{\rho_0(b)v(b)}{\rho_0(b)v(b)+N\rho_1(0)v(0)} = \mu_0 \tag{A.19d}$$

In the above step, we use $\rho_0(b) = \mu_0$, $v(b) = C$ from (5.18), and $N\rho_1(0)v(0) = (1-\mu_0)C$ from (5.20).

Proof of Theorem 5.4.10

Similar to proof of Lemma 5.4.9, we proceed for the general α as,

$$\mu_0(\tau(b)) = \frac{1}{1 + a(1 - 1/b)^{-\alpha x_0 \tau(b)}} \tag{A.20}$$

But from proof of Lemma 5.4.9 above, we find for $\alpha = 1$ that $\mu_0 = 1/(1 + a(1 - 1/b)^{-x_0\tau(b)})$. After rearranging, we obtain $(1 - 1/b)^{-x_0\tau(b)} = (1 - \mu_0)/a\mu_0$. Substituting this into (A.20) for general α completes the proof. ∎

Bibliography

[1] Minnesota Internet Traffic Studies (MINTS). http://www.dtc.umn.edu/mints/, Nov 2009.

[2] BIC and CUBIC project webpage. http://research.csc.ncsu.edu/netsrv/, June 2012.

[3] The ISC Domain Survey. http://www.isc.org/solutions/survey, Jan 2012.

[4] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. http://tools.ietf.org/html/rfc5681, Sep 2009. RFC 5681.

[5] M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. http://tools.ietf.org/html/rfc2581, Apr 1999. RFC 2581.

[6] N. Beheshti, Y. Ganjali, M. Ghobadi, N. McKeown, and G. Salmon. Experimental Study of Router Buffer Sizing. In *Proc. of Internet Measurement Conference*, 2008.

[7] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *Proc. of IEEE INFOCOM*, 1996.

[8] D. P. Bertsekas and R. G. Gallager. *Data Networks*. Prentice-Hall Inc., 1991.

[9] T. Bonald and L. Massoulie. Impact of Fairness on Internet Performance. In *Proc. of ACM SIGMETRICS*, 2001.

[10] B. Braden and et. al. Recommendations on queue management and congestion avoidance in the internet. http://tools.ietf.org/html/rfc2309, April 1998. RFC 2309.

[11] R. Braden, D. Clark, and S. Shenker. Integrated services in the internet architecture: an overview. http://tools.ietf.org/html/rfc1633, 1994. RFC 1633.

[12] L. S. Brakmo, S. W. OMalley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proc. of ACM SIGCOMM*, 1994.

[13] B. Briscoe. Flow rate fairness: dismantling a religion. *ACM SIGCOMM Computer Communication Review*, 37(2), Apr 2007.

[14] K. chan Lan and J. Heidemann. A measurement study of correlations of internet flow characteristics. *Computer Networks, Elsevier*, 50(1), Jan 2006.

[15] P. Chhabra, A. John, H. Saran, and R. Shorey. Controlling malicious sources at internet gateways. In *Proc. of IEEE ICC*, 2003.

[16] D.-M. Chiu and R. Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Journal of Computer Networks and ISDN*, 17(1), 1989.

[17] M. Christiansen, K. Jeffay, D. Ott, and F. D. Smith. Tuning RED for Web Traffic. *IEEE/ACM Trans on Networking*, 9(3), June 2001.

[18] D. Clark. The design philosophy of the darpa internet protocols. In *Proc. of ACM SIGCOMM*, Aug 1988.

[19] M. E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. In *Proc. of ACM SIGMETRICS*, 1996.

[20] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Proc. of ACM SIGCOMM*, 1989.

[21] D. M. Divakaran, G. Carofiglio, E. Altman, and P. V.-B. Primet. A flow scheduler architecture. In *Proc. of IFIP Networking*, 2010.

[22] D. M. Divakaran, S. Soudan, P. V.-B. Primet, and E. Altman. A survey on core switch designs and algorithms. http://hal.inria.fr/inria-00388943/, 2009. Technical report.

[23] A. T. Eshete and Y. Jiang. Approximate Fairness Through Limited Flow List. In *Proc. of International Teletraffic Congress (ITC)*, 2011.

[24] A. T. Eshete and Y. Jiang. On the Flow Fairness of Aggregate Queues. In *Proc. of BCFIC*, Feb 2011.

[25] A. T. Eshete and Y. Jiang. On the Transient Behavior of CHOKe. Nov 2011. Submitted to IEEE/ACM Trans. on Networking.

[26] A. T. Eshete and Y. Jiang. Generalizing the CHOKe Flow Protection. *Computer Networks, Elsevier*, Sep 2012. In Press.

[27] A. T. Eshete and Y. Jiang. Protection from Unresponsive Flows with Geometric CHOKe. In *Proc. of IEEE ISCC*, 2012.

[28] A. T. Eshete, Y. Jiang, and L. Landmark. Fairness among High Speed and Traditional TCP under different Queue Management Mechanisms. In *Proc. of AINTEC (to appear)*, Nov 2012.

[29] K. Fall and K. Varadhan. ns–network simulator (v2.34). http://www.isi.edu/nsnam/ns/, Nov 2011.

[30] S. Floyd. RED project webpage. http://www.icir.org/floyd/red.html.

[31] S. Floyd. Congestion Control Principles. http://www.ietf.org/rfc/rfc2914.txt, Sep 2000. RFC 2914.

[32] S. Floyd. HighSpeed TCP for Large Congestion Windows. http://www.ietf.org/rfc/rfc3649.txt, Dec 2003. RFC 3649.

[33] S. Floyd. Metrics for the Evaluation of Congestion Control Mechanisms. http://tools.ietf.org/html/rfc5166, Mar 2008. RFC 5166.

[34] S. Floyd and M. Allman. Comments on the Usefulness of Simple Best-Effort Traffic. http://www.ietf.org/rfc/rfc5290.txt, Jul 2008. RFC 5290.

[35] S. Floyd and K. Fall. Promoting the use of end-to-end congestion control in the internet. *IEEE/ACM Trans. on Networking*, 7(4), 1999.

[36] S. Floyd, R. Gummadi, and S. Shenker. Adaptive RED: An algorithm for increasing the robustness of RED's Active Queue Management. Tech. Report, 2001.

[37] S. Floyd, T. Henderson, and A. Gurtov. The NewReno modification to TCP's fast recovery algorithm. http://www.ietf.org/rfc/rfc3782.txt, Apr 2004. RFC 3782.

[38] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Trans. on Networking*, 1(4), 1993.

[39] S. Floyd and J. Kempf. IAB Concerns Regarding Congestion Control for Voice Traffic in the Internet. http://tools.ietf.org/html/rfc3714, Mar 2004. RFC 3714.

[40] S. Floyd and E. Kohler. Internet research needs better models. *ACM SIGCOMM Computer Communication Review*, 33(1), 2003.

[41] S. J. Golestani. A Self-Clocked Queueing Scheme for Broadband Applications. In *Proc. of IEEE INFOCOM*, June 1994.

[42] V. V. Govindaswamy, G. Záruba, and G.Balasekaran. RECHOKe: A Scheme for Detection, Control and Punishment of Malicious Flows in IP Networks. In *Proc. of IEEE GLOBECOM*, 2007.

[43] P. Goyal, S. S. Lam, and H. M. Determining end-to-end delay bounds in heterogeneous networks. *Multimedia System*, 5(3), May 1997.

[44] P. Goyal, H. M. Vin, and H. Cheng. Start-time Fair Queuing: A Scheduling Algorithm for Integrated Services Packet Switching Networks. *IEEE/ACM Trans. on Networking*, 5(5), 1997.

[45] S. Ha, I. Rhee, and L. Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 42, 2008.

[46] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. http://tools.ietf.org/html/rfc6582, 2012. RFC 6582.

[47] C. Hu, Y. Tang, X. Chen, and B. Liu;. Per-Flow Queueing by Dynamic Queue Sharing. *Proc. of IEEE INFOCOM*, 2007.

[48] V. Jacobson. Congestion avoidance and control. In *Proc. of ACM SIGCOMM*, 1988.

[49] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons, 1991.

[50] H. Jiang and C. Dovrolis. Why is the internet traffic bursty in short time scales? In *ACM SIGMETRICS Performance Evaluation Review*, 2005.

[51] C. Jin, D. X. Wei, and S. H. Low. FAST TCP: motivation, architecture, algorithms, performance. In *Proc. of IEEE INFOCOM*, 2004.

[52] D. Katabi, M. Handley, and C. Rohrs. Internet congestion control for high bandwidth-delay product networks. In *Proc. of ACM SIGCOMM*, 2002.

[53] J. Kaur. *Scalable Network Architectures for Providing Per-flow Service Guarantees*. PhD thesis, The University of Texas at Austin, Aug 2002.

[54] J. Kaur and H. M. Vin. Core-stateless guaranteed rate scheduling algorithms. In *Proc. of IEEE INFOCOM*, 2001.

[55] F. Kelly, A. Maulloo, and D. Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49(3), 1998.

[56] T. Kelly. Scalable tcp: improving performance in highspeed wide area networks. *ACM SIGCOMM Computer Communication Review*, 33(2), 2003.

[57] S. Keshav. *An engineering approach to computer networking: ATM networks, the Internet, and the telephone network*. Addison-Wesley Longman Publishing Co, 1997.

[58] E. Kohler, M. Handley, and S. Floyd. Datagram congestion control protocol (dccp). http://www.ietf.org/rfc/rfc4340.txt, Mar 2006. RFC 4340.

[59] A. Kortebi, L. Muscariello, S. Oueslati, and J. Roberts. On the scalability of fair queueing. In *Proc. of ACM SIGCOMM HotNets*, 2004.

[60] A. Kortebi, L. Muscariello, S. Oueslati, and J. Roberts. Evaluating the number of active flows in a scheduler realizing fair statistical bandwidth sharing. In *Proc. of ACM SIGMETRICS*, 2005.

[61] A. Kortebi, S. Oueslati, and J. Roberts. Cross-protect: implicit service differentiation and admission control,. In *High Performance Switching and Rout-*

ing, 2004.

[62] J. F. Kurose and K. W. Ross. *Computer Networking: a top down approach (4th edition)*. Addison-Wesley, 2008.

[63] J.-Y. Le Boudec. Rate adaptation, Congestion Control and Fairness: A Tutorial, Dec. 2008.

[64] D. Lin and R. Morris. Dynamics of random early detection. *ACM SIGCOMM Computer Communication Review*, 27(4), 1997.

[65] S. Lohr. Video road hogs stir fear of internet traffic jam. http://www.nytimes.com/2008/03/13/technology/13net.html, Marc 2008.

[66] R. Mahajan, S. Floyd, and D. Wetherall. Controlling high-bandwidth flows at the congested router. In *Proc. of IEEE ICNP*, 2001.

[67] S. Mascolo. Reasons not to deploy TCP BIC/CUBIC. http://www.postel.org/pipermail/end2end-interest/, Nov 2011.

[68] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. http://tools.ietf.org/html/rfc2018, Oct 1996. RFC 2018.

[69] M. May, J. Bolot, C. Diot, and B. Lyles. Reasons not to deploy RED. In *Proc. of IEEE IWQoS*, May-June 1999.

[70] K. Munir, M. Welzl, and D. Damjanovic. Linux beats Windows! - or the Worrying Evolution of TCP in Common Operating Systems. In *Proc. of PFLDnet*, 2007.

[71] J. Nagle. Congestion Control in IP/TCP Internetworks. http://www.ietf.org/rfc/rfc896.txt, Jan 1984. RFC 896.

[72] J. Nagle. On packet switches with infinite storage. *IEEE Trans. on Communications*, 35(4):435—438, 1987.

[73] A. M. Odlyzk. Data networks are mostly empty and for good reason,. *IT Professional*, 1(2), Mar-Apr 1999.

[74] T. J. Ott, T. Lakshman, and L. H. Wong. Sred: Stabilized RED. In *Proc. of IEEE INFOCOM*, 1999.

[75] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: a simple model and its empirical validation. In *Proc. of ACM SIGCOMM*, 1998.

[76] R. Pan, L. Breslau, B. Prabhakar, and S. Shenker. Approximate fairness through differential dropping. *ACM SIGCOMM Computer Communication Review*, 33, 2003.

[77] R. Pan, C. Nair, B. Yang, and B. Prabhakar. Packet dropping schemes, some examples and analysis. In *Proc. of Allerton Conference on Communication, Control and Computing*, 2001.

[78] R. Pan, B. Prabhakar, and K. Psounis. CHOKe - a stateless active queue management scheme for approximating fair bandwidth allocation. In *Proc. of IEEE INFOCOM*, 2000.

[79] D. Papadimitriou, M. Welzl, M. Scharf, and B. Briscoe. Open research issues in internet congestion control. http://tools.ietf.org/html/rfc6077, Feb 2011. RFC 6077.

[80] A. K. Parekh. *A Generalized Processor Sharing Approach to Flow Control in Integrated Services Networks.* PhD thesis, Massachusetts Institute of Technology, Feb 1992.

[81] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single node case. *IEEE/ACM Trans. Networking*, 1(3), June 1993.

[82] J. Postel. User datagram protocol. http://www.ietf.org/rfc/rfc768.txt, Aug 1980. RFC 768.

[83] F. Qiana, A. Gerber, Z. M. Mao, S. Sen, O. Spatscheck, and W. Willinger. TCP Revisited: A Fresh Look at TCP in the Wild. In *Proc. of Internet Measurement Conference*, Nov 2009.

[84] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. http://tools.ietf.org/html/rfc3168, Sep 2001. RFC 3168.

[85] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2, 1984.

[86] S. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. *IEEE/ACM Trans. Networking*, 4, June 1996.

[87] B. Sikdar, S. Kalyanaraman, and K. S. Vastola. An integrated model for the latency and steady-state throughput of tcp connections. *Performance Evaluation*, 46(2-3), 2001.

[88] R. Stanojević and R. Shorten. Beyond CHOKe: stateless fair queueing. In *Proc. of NET-COOP*, 2007.

[89] D. Stiliadis and A. Varma. Latency-rate servers: a general model for analysis of traffic scheduling algorithms. *IEEE/ACM Transactions on Networking*, 6(5), 1998.

[90] I. Stoica. *Stateless Core: A Scalable Approach for Quality of Service in the Internet.* PhD thesis, CMU, Pittsburgh,USA, Dec 2000.

[91] I. Stoica. CSFQ project webpage. http://www.cs.berkeley.edu/ istoica/csfq/, June 2012.

[92] I. Stoica, S. Shenker, and H. Zhang. Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks. In *Proc. of ACM SIGCOMM*, 1998.

[93] I. Stoica, H. Zhang, and S. Shenker. Self-Verifying CSFQ. In *Proc. of IEEE INFOCOM*, 2002.

[94] B. Suter, T. Lakshman, D. Stiliadis, and A. Choudhury. Buffer Management Schemes for Supporting TCP in Gigabit Routers with Per-flow Queueing. *IEEE Journal on Selected Areas in Communications*, 17(6), Jun 1999.

[95] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-Speed and Long Distance Networks. In *Proc. of IEEE INFOCOM*, 2006.

[96] A. Tang, J. Wang, and S. H. Low. Understanding CHOKe: Throughput and Spatial Characteristics. *IEEE/ACM Trans. on Networking*, 12(4), 2004.

[97] M. Waldvogel, G. Varghese, J. Turner, and B. Plattneri. Scalable High Speed IP Routing Lookups. In *Proc. of ACM SIGCOMM*, 1997.

[98] J. Wang, A. Tang, and S. H. Low. Maximum and asymptotic UDP throughput under CHOKe. In *Proc. of ACM SIGMETRICS*, 2003.

[99] D. X. Wei and P. Cao. NS2 TCPLinux: An NS2 TCP Implementation with Congestion Control Algorithms from Linux. In *Proc. of Valuetools—Workshop on NS2*, 2006.

[100] L. Xu, K. Harfoush, and I. Rhee. Binary Increase Congestion Control (BIC) for Fast Long-Distance Networks. *Proc. of IEEE INFOCOM*, 2004.

[101] L. Xue, C. Cui, S. Kumar, and S.-J. Park. Experimental evaluation of the effect of queue management schemes on the performance of high speed TCPs in 10Gbps network environment . In *Proc. of IEEE ICNC*, 2012.

[102] P. Yang, W. Luo, L. Xu, J. Deogun, and Y. Lu. TCP Congestion Avoidance Algorithm Identification. In *Proc. of IEEE ICDCS*, 2011.

[103] D. K. Y. Yau, J. C. S. Lui, F. Liang, and Y. Yam. Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. *IEEE/ACM Trans. on Networking*, 13(1), 2005.

[104] L. Zhang. Virtual clock: a new traffic control algorithm for packet switching networks. In *Proc. of ACM SIGCOMM*, 1990.

[105] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the characteristics and origins of internet flow rates. *Proc. of ACM SIGCOMM*, 2002.