Sender-driven Bandwidth Differentiation for Transmitting Multimedia Flows over TCP

LAU KWOK HUNG

A Thesis Submitted in Partial Fulfillment

of the Requirements for the Degree of

Master of Philosophy

in

Information Engineering

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor Professor Jack Y. B. Lee, who whole-heartedly provided guidance and invaluable advice throughout the whole duration of my research.

I would also like to thank my colleagues Felix, Ryan, Gary and Rudolf. They provided me with the earnest help and friendship during the past two years.

ABSTRACT

Over the years the Internet has shown extraordinary scalability and robustness in spite of the explosive growth in geographical reach, user population size, as well as network traffic volume. This scalability and robustness is, in no small part, supported by the Internet's transport protocols, and the Transmission Control Protocol (TCP) in particular. Nevertheless, with the rapid growth of multimedia-rich contents in the Internet, such as audio and video, the many strengths of TCP in data delivery are slowly imposing bottlenecks in multimedia data delivery where different media data flows often have different needs for bandwidth. As TCP's congestion control algorithm enforces fair bandwidth sharing among traffic flows sharing the same network bottleneck, different media data flows will receive the same bandwidth irrespective of the actual needs of the multimedia data being delivered. This work addresses this limitation by proposing a new algorithm to achieve non-uniform bandwidth allocation among TCP flows originating from the same sender passing through the same network bottleneck to multiple receivers. The proposed algorithm, called Virtual Packet Substitution (VPS), has four desirable features: (a) it allows the allocation of bottleneck bandwidth between a group of TCP flows; (b) the resultant traffic flows as a whole, maintain the same fair bandwidth sharing property with other competing TCP flows; (c) it can be implemented entirely in the sender's TCP protocol stack; and (d) it is compatible with and does not require modification to existing TCP protocol stack at the clients. Simulation results show that the proposed VPS algorithm can achieve accurate bandwidth allocation while still maintaining fair bandwidth sharing with competing TCP flows.

摘要

多年來儘管地理範圍，用戶人口數量，以及網絡流量的爆炸性增長，互聯網仍然持續擴充和非常穩健。這種持續擴充和穩健的性質很大程度上是受惠於互聯網的傳輸控制協議 (Transmission Control Protocol, TCP)。儘管如此，快速增長的網上多媒體內容，如視聽內容，令原本傳輸控制協議的優勢慢慢限制著多媒體數據放送，而不同的媒體數據往往有著不同的頻寬需求。因爲傳輸控制協議的擁塞控制算法 (Congestion control) 公平地分配瓶頸 (bottleneck) 頻寬給流經同一瓶頸的數據流，不同的數據流不論實際的需要亦會獲得同一頻寬。針對這個限制，本論文提出一個新的算法，去實現非平均頻寬分配給源自同一傳送端，經過同一瓶頸，而到達不同接收端的數據流。這個建議算法稱爲虛擬包替代 (Virtual Packet Substitution, VPS)，並擁有以下四個特點：(一) 允許頻寬分配予經過同一瓶頸的數據流；(二) 綜合流量與其他數據流的流量保持公平競爭；(三) 完全實施在傳送端的傳輸控制協議堆疊；和 (四) 兼容及無須修改接收端的傳輸控制協議堆疊。模擬結果表明虛擬包替代能夠精確地分配頻寬，同時維持和其他數據流公平地分享頻寬。

CONTENTS

Acknowledgement 1
Abstract 2
摘要 3
Chapter 1 INTRODUCTION 6
Chapter 2 BACKGROUND AND RELATED WORK 9
2.1 Application-Layer Bandwidth Differentiation 9
2.2 Related Work 14
2.2.1 Bandwidth Differentation 14
2.2.2 Shared Congestion Management 15
2.2.3 Flow Partition 16
Chapter 3 VPS PROTOCOL ARCHITECTURE 17
3.1 Virtual and Actual Flows 18
3.2 VPS Controller 21
Chapter 4 ACK Translation 25
4.1 Fast Retransmit and Fast Recovery 27
4.2 Timeout 30
4.3 Packet and ACK Reordering 33
4.4 False Duplicate ACK Suppression 35
4.5 Maxburst 37
4.6 Memory Overhead and Computation Complexity 38
Chapter 5 BANDWIDTH DIFFERENTIATION 41
5.1 Distribution of Virtual Packets 41
5.2 Temporary Suspension of Actual Flows 43
5.3 Receive Window Limit 44
5.4 Limited Data Transmission 44
Chapter 6 PERFORMANCE EVALUATOIN 45
6.1 Performance Metric 45
6.2 Simulation Setup 46
6.3 Performance over Different Time Scales 47
6.4 Performance over Different Bottleneck Bandwidth 53
6.5 Performance over Different Application-specified Ratios 54
6.6 Performance over Different Number of Flows 57

6.7 Heterogeneous Receivers 60
Chapter 7 CONCLUSIONS AND FUTURE WORK 65
Bibliography 66

Chapter 1

INTRODUCTION

Over the years the Internet has shown extraordinary scalability and robustness in spite of the explosive growth in geographical reach, user population size, as well as network traffic volume. This scalability and robustness is, in no small part, supported by the Internet's transport protocols, and the Transmission Control Protocol (TCP) [4] in particular. The flow and congestion control algorithms in TCP ensure that network bandwidth is shared among competing traffic flows in a fair manner [5], and network congestions are automatically alleviated by throttling the sending rate at the source.

Nevertheless, with the rapid growth of multimedia-rich contents in the Internet, such as audio and video, the many strengths of TCP in data delivery are slowly imposing bottlenecks in multimedia data delivery [1-3]. Specifically, TCP's congestion control algorithm enforces fair bandwidth sharing among traffic flows sharing the same network bottleneck. Thus two multimedia flows going through the same network bottleneck will receive the same bandwidth irrespective of the actual needs of the multimedia data being delivered.

For example, suppose a multimedia server is sending two streams of video data S_1 and S_2, of encoded video bit-rates 0.3 Mbps and 0.7Mbps respectively, through the same network bottleneck [17, 18] to two different clients. Now if the network bottleneck has 1Mbps available bandwidth, then in principle there is sufficient bandwidth to transport both video streams. However, if the server simply sends both video streams using TCP or a TCP-friendly streaming protocol that emulates TCP's fair bandwidth sharing property, then each of the video streams will get half the available bandwidth, i.e., 0.5Mbps. Obviously, this is too much for S_1 at 0.3Mbps and insufficient for S_2 at 0.7Mbps.

Apparently, if the server knows the video streams' bit-rates and can control the transmission rates at the application layer, then it seems the problem can be solved by sending the video at their playback rates instead of the rates allowed by TCP. This approach, however, suffers from two limitations. First, while it is possible for the server application to send data at a rate lower than the rate allowed by TCP, the opposite is simply impossible as the application will soon be blocked from sending more data by the network programming API (e.g., sockets [6]) once the sender's transport buffer is full. Second, even if the server frees up bandwidth from a TCP flow by sending at a lower rate, the saved bandwidth may not be fully transferred to another specific TCP flow.

To see why, suppose there are two other competing TCP flows S_3 and S_4 sharing the same bottleneck as the two video streams S_1 and S_2, then any bandwidth, say C bps, freed up by the server (through sending S_1 at a lower rate) will be up for grab by the remaining three competing flows (S_2, S_3, and S_4). Given TCP's fair bandwidth sharing property this means that each of the competing flows (S_2, S_3, and S_4) will receive one-third of the freed bandwidth, i.e., $C/3$ bps. This dilution effect increases with more competing TCP flows sharing the same network bottleneck and thus differentiating bandwidth at the application layer is not effective in the Internet where it is not uncommon to have tens or hundreds of flows going through a network link. Chapter 2.1 will further illustrate this limitation using simulations.

We tackle this problem in this work by proposing a new algorithm to achieve bandwidth allocation among TCP flows originating from the same sender passing through the same network bottleneck to multiple receivers. The proposed algorithm, called Virtual Packet Substitution (VPS), has four desirable features: (a) it allows the allocation of bottleneck bandwidth between a group of TCP flows; (b) the resultant traffic flows as a whole, maintain the same fair bandwidth sharing property with other competing TCP flows; (c) it can be implemented entirely in the sender's TCP protocol stack; and (d) it is compatible with and does not require modification to existing TCP protocol stack at the clients.

The rest of the thesis is structured as follows. Chapter 2 investigates in detail why differentiating bandwidth at the application layer is ineffective and then reviews some previous related work in the literature. Chapter 3 presents the operating principles of the proposed VPS algorithm, and Chapter 4 and 5 discuss in detail the algorithms for ACK translation and bandwidth differentiation. Chapter 6 evaluates the proposed VPS using extensive simulations and Chapter 7 summarizes the work and discusses some future work.

Chapter 2

BACKGROUND AND RELATED WORK

This chapter presents results from simulated experiments to show that performing bandwidth differentiation at the application layer is ineffective, and then review some previous related work in the literature.

2.1 Application-Layer Bandwidth Differentiation

A network application can control its data transmission rate to a receiver by explicitly controlling the timings of data transmissions, subject to the flow and congestion controls of the transport protocol. Therefore to achieve bandwidth differentiation among multiple receivers connected to the same sender, it appears that one can simply control the relative data transmission timings for the respective receivers without modifying the transport protocol. While this may be true for transport protocols such as UDP, which has no flow or congestion control, the following experiment will show that this is ineffective for transport protocols that have built-in congestion control, such as TCP or TCP-friendly protocols (e.g., TFRC [26]).

Let there be n TCP flows which share the same network bottleneck and is served by the same sender. The goal is to allocate the bottleneck bandwidth to the TCP flows according to application-specified ratios $\{w_1, w_2, \ldots, w_n\}$, i.e., flow i will be allocated a proportion of $w_i / \sum_{\forall j} w_j$ of the bottleneck bandwidth. Note that here the term *bottleneck bandwidth* refers to the total amount of bandwidth that would have been received by n ordinary TCP flows passing through the same network bottleneck, and so it will be smaller than the bottleneck link capacity in the presence of other competing TCP flows.

Specifically, a network application typically sends data using a network API such as sockets [6]. By calling an API function such as socket's send() the application can submit data for transmission over the transport protocol. In case the application sends data faster than the rate the transport protocol can handle, data will accumulate in the socket buffer until it is full – at which point the send() function will either block until socket buffer becomes available again, or will return failure. The former case is commonly referred as blocking I/O and the latter case is commonly referred as non-blocking I/O, and both I/O models are widely supported in modern operating systems.

If blocking I/O is employed, then the sender can simply use weighted round-robin to send data to the n receivers, i.e., sending w_i units of data for receiver i in each round. In case a send() function call blocks the sender simply waits until it unblocks before continuing with the next send() call. In contrast, if non-blocking I/O is employed then whenever the send() call fails with a buffer full condition, indicating that the transport protocol cannot keep up with the data rate, the sender will have the option to skip the blocked flow and proceed to the next one.

We implemented the above application-layer bandwidth differentiation algorithm using both blocking and non-blocking I/O models in the simulator NS2 [10]. The simulated network topology is depicted in Fig. 1, with 3 TCP flows controlled by the sender and 3 other competing TCP flows serving as cross traffic. In the dump-bell topology the bottleneck link is configured with 10Mbps bandwidth. The bandwidth ratio for flow i is set to $w_i=1+(i-1)d$, where $i=\{1,2,3\}$ and the weight differential $d=\{1,2,\ldots,10\}$. Results are averaged over 30 simulation runs.

Fig. 2 shows the throughput of the three controlled TCP flows versus the bandwidth weight differential d. For example, with $d=1$ the three TCP flows should have throughput ratios of 1:2:3. When implemented using blocking mode I/O the three TCP flows can indeed achieve throughput ratios close to the target ratios. This is not surprising as the application sends data strictly according to the bandwidth ratios. By contrast, the three controlled TCP flows in the non-blocking I/O implementation do *not* exhibit any bandwidth differentiation at all. In fact their throughputs are the same regardless of the bandwidth ratio settings.

To see why, consider the aggregate throughput of the three controlled TCP flows plotted in Fig. 3. Here we observe that the aggregate throughout in blocking I/O mode is substantially lower than those in non-blocking I/O mode. This is due to interaction between the application's bandwidth-differentiation algorithm and TCP's congestion control algorithm. Specifically, there are altogether 6 TCP flows in the network, of which 3 of them are under control by the application. Due to TCP's fair bandwidth sharing property we would expect the 6 flows to share the bottleneck bandwidth equally, say at C_f bps. In other words while the application can send data slower than C_f, it cannot send data faster than C_f bps as the send() function call will either block under blocking mode or fail under non-blocking mode.

In blocking mode this means that the flow with the highest bandwidth ratio can only send data at the fair share bandwidth C_f, while the other flows of lower bandwidth ratio simply sends at rate *below* the fair share bandwidth due to the bandwidth-differentiation algorithm. Thus the aggregate throughout of the three controlled TCP flows will become lower than $3C_f$, which is undesirable for the application.

In contrast, under non-blocking I/O mode whenever the send() call fails (due to reaching the fair share bandwidth limit) the algorithm simply skips the current flow and proceeds to send data for the next flow in the round. This results in deviation from the target bandwidth ratios (due to skipped send() calls) but enables the other flows to send data even if one or more of the flows are already saturated. As a result the fair share bandwidth of all three flows can still be utilized in non-blocking I/O mode. However, as all TCP flows share the same long-term bandwidth the application ends up not being able to achieve bandwidth differentiation at all as evident in Fig. 2. Therefore irrespective of the I/O modes, performing bandwidth differentiation at the application layer is simply ineffective. One can either achieve bandwidth differentiation but not fair bandwidth sharing (via blocking I/O) or fair bandwidth sharing but not bandwidth differentiation (via non-blocking I/O), but not both simultaneously.

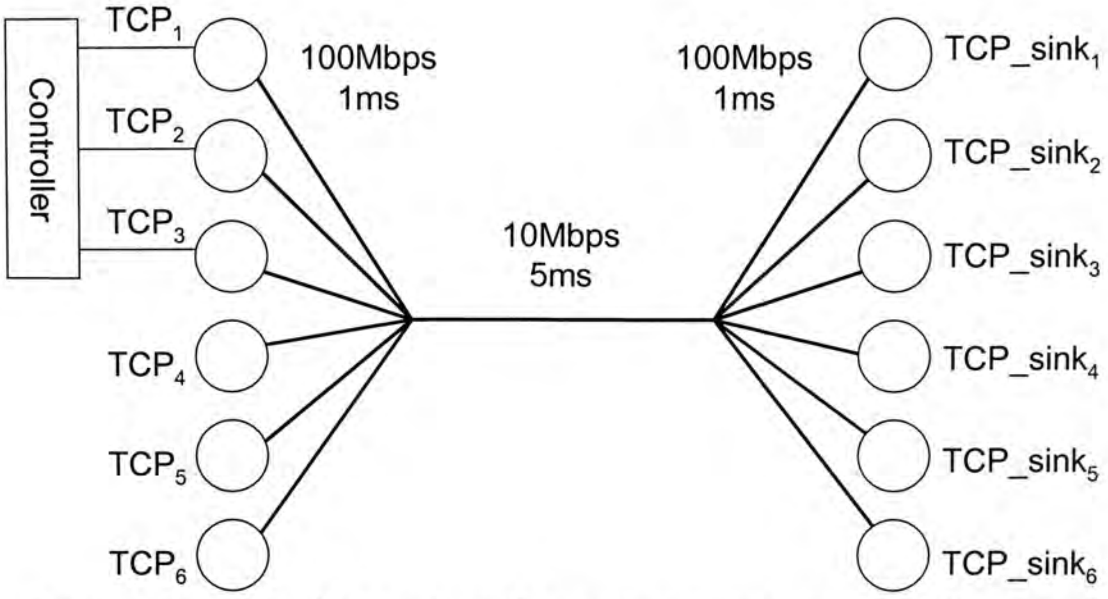


Fig. 1. Topology of the simulation for application-layer bandwidth differentiation.

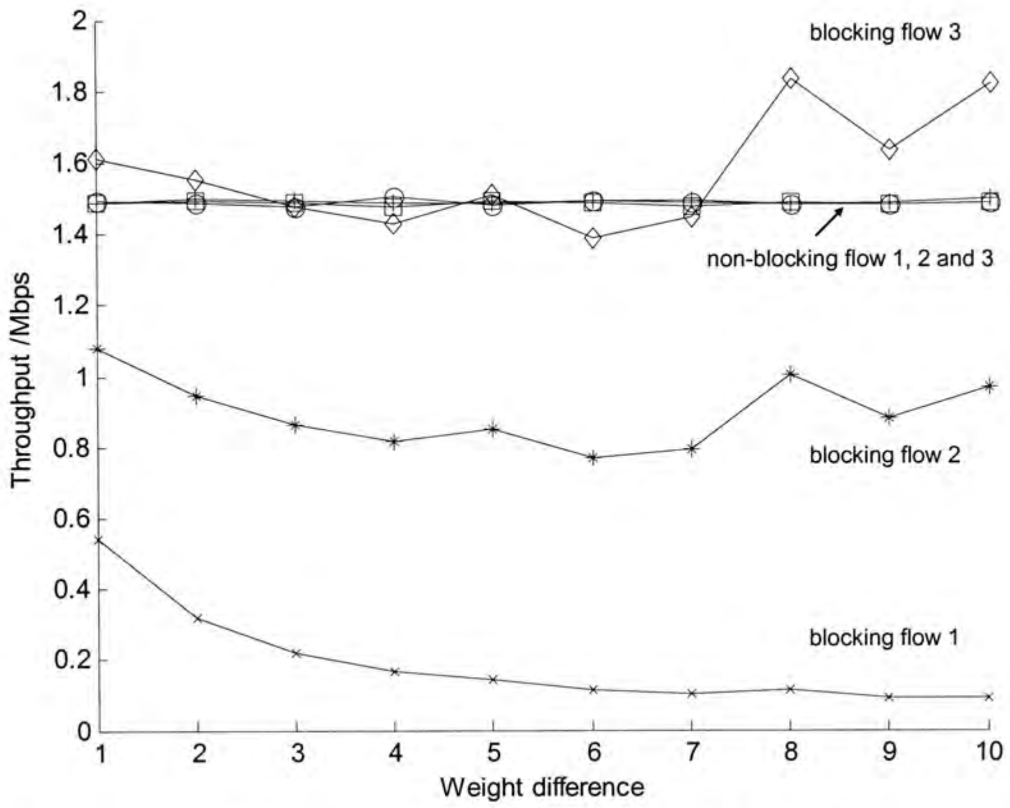


Fig. 2. Throughput of controlled TCP flows under blocking and non-blocking I/O..

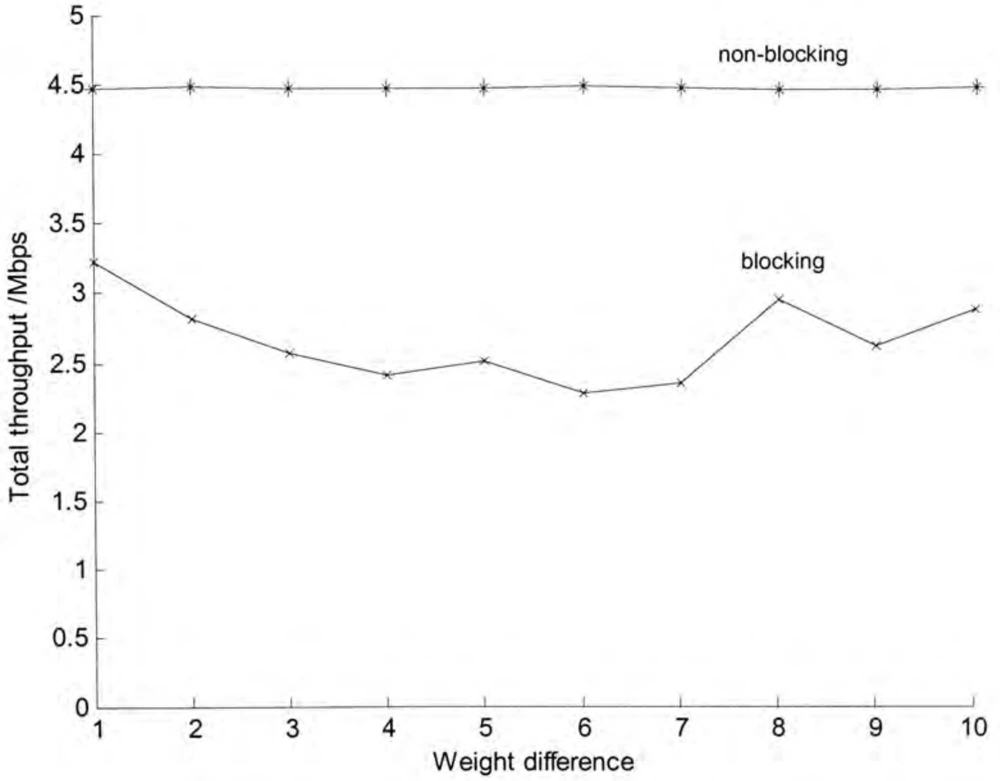


Fig. 3. Aggregate throughput of the 3 TCP flows versus bandwidth weight differentials.

2.2 Related Work

Noting that differentiating bandwidth at the application-layer is ineffective, researchers have proposed various ways to achieve bandwidth differentiation at the transport layer. This section reviews these previous work and also related work in the areas of shared congestion management and flow partition.

2.2.1. Bandwidth Differentiation

Two previous studies [7-8] had investigated the problem of bandwidth differentiation. Mehra *et al.* [7] proposed a receiver-driven bandwidth allocation algorithm that can allocate bottleneck bandwidth among multiple TCP flows. The principle is to adjust the receiver's TCP receive windows and the delays in sending acknowledgements such that prioritized and weighted bandwidth sharing can be achieved. However, their algorithm can only allocate bandwidth among flows destined to the *same receiver*, whereas in this thesis the focus is to differentiate bandwidth for flows originating from the *same sender* to multiple receivers.

In another study, Crowcroft and Oechslin [8] proposed the MulTCP congestion control algorithm to achieve differentiated end-to-end service. The principle is to change the rate at which a TCP flow increases and decreases its congestion window to make it behaves like *N* concurrent TCP flows. Bandwidth allocation can then be achieved by setting the multiplier *N* for each flow. The protocol is simple to implement and only limited coordination among multiple TCP flows is required. However, as the objective of MulTCP is not to minimize the impact to other competing ordinary TCP flows, it will nevertheless cause the competing ordinary TCP flows to lose some of their fair share of bandwidth. This is further analyzed in Chapter 6 which extensive simulations are conducted to quantitatively compare MulTCP with the proposed VPS algorithm.

2.2.2. Shared Congestion Management

The underlying principle of bandwidth differentiation is to share congestion information of multiple data flows in regulating the transmission rate of individual flows. Similar principle has also been explored in the context of optimizing TCP performance. For example, J. Touch [30] proposed TCP control blocks (TCB) interdependence to cache and share TCP states, such as connection state, current RTT estimates, congestion control information, etc., among different TCP connections. The goal is to improve the transient performance of TCP, such as by using a cached congestion window size for a new TCP flow connecting to the same host in a previous connection to reduce the slow-start effect. These cached TCBs are consulted only during connection setup and teardown. In comparison, VPS takes the idea one step further by continuously sharing congestion information of multiple TCP flows *during* the whole connection period.

Balakrishnan *et al.* [1] proposed a Congestion Manager (CM) architecture where all traffic flows originating from the same host are aggregated and managed as a single flow. The CM collects congestion information and exposes an API to application to enable them to query the network's characteristics. The primary goal is to combat the trend where increasingly more applications attempt to work-around TCP's congestion control and fair bandwidth sharing property by establishing multiple concurrent TCP flows to the same host, or switching to UDP altogether to gain unfair share of the available bandwidth. When fully integrated with the network application the CM architecture can also assist network applications to adapt to congestion conditions without requiring them to perform congestion control or bandwidth probing on their own.

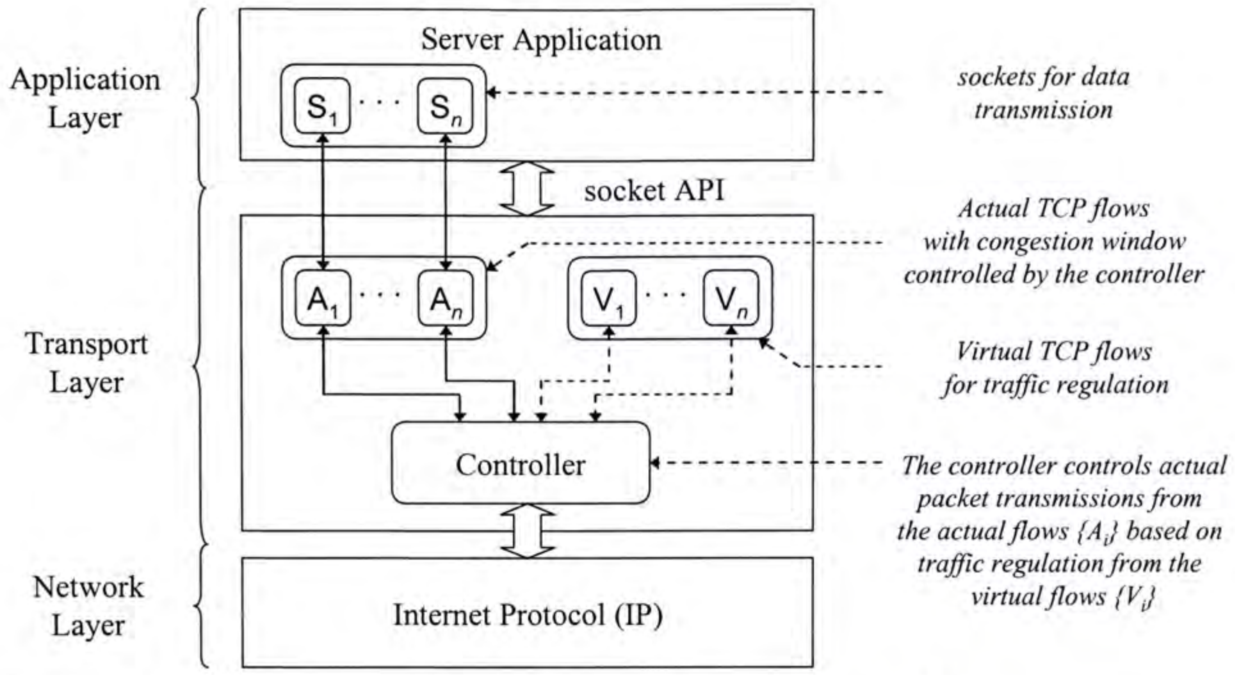
2.2.3. Flow Partition

The VPS algorithm performs bandwidth differentiation for flows sharing the same bottleneck. Thus VPS requires a method to identify and group such flows together – this is known as the flow partition problem. Since VPS is designed to be sender-driven, the flow partition mechanism should also be sender-driven such that no modification is required at the receiver. A number of such flow partition algorithms had been proposed in the literature, including the work by Lili Wang *et al.* [22] and Ningning Hu *et al.* [23]. Lili Wang *et al.* proposed to use measured TCP throughput to derive path correlations as flows sharing the same bottleneck are expected to have correlated throughput. This approach is entirely passive in the sense that no probing packets need to be injected into the network. In contrast, Ningning Hu *et al.* proposed to send a train of probing packets with increasing time-to-live (TTL) values to identify the location of the path bottleneck. The idea is that when the TTL of the probing packets is decremented to zero, the intermediate routers are expected to discard the probing packets and send ICMP error reports back to the sender. By measuring the time gap between ICMP packets from each router, the sender can then infer the location of the path bottleneck. These flow partition algorithms, being sender-based, can be adopted for use with the proposed VPS algorithm. Therefore the rest of the thesis will simply assume flow partition has been performed and focus on a group of flows already known to share the same network bottleneck.

Chapter 3

VPS PROTOCOL ARCHITECTURE

There are two problems to overcome to enable bandwidth differentiation in TCP. First, the congestion control algorithm needs to be modified such that credits received (via acknowledgement packets) for data transmission can be reallocated from one flow to another flow. The goal is to allocate the bottleneck bandwidth to the TCP flows according to application-specified ratios $\{w_1, w_2, \ldots, w_n\}$, i.e., flow i will be allocated a proportion of $w_i / \sum_{\forall j} w_j$ of the bottleneck bandwidth. Again the term *bottleneck bandwidth* refers to the total amount of bandwidth that would have been received by n ordinary TCP flows passing through the same network bottleneck, and so it will be smaller than the bottleneck link capacity in the presence of other competing TCP flows. Second, we need to regulate the aggregate traffic flows such that as a whole the group behaves in the same way as ordinary TCP flows so that other competing TCP flows (either from other hosts or from the same hosts but managed by ordinary TCP) will neither gain nor lose bandwidth.



Application Layer
Transport Layer
Network Layer
Server Application
S_1 · · · S_n
socket API
A_1 · · · A_n
V_1 · · · V_n
Controller
Internet Protocol (IP)
sockets for data transmission
Actual TCP flows with congestion window controlled by the controller
Virtual TCP flows for traffic regulation
The controller controls actual packet transmissions from the actual flows {A_i} based on traffic regulation from the virtual flows {V_i}

Fig. 4. Protocol architecture and interfaces to application and network layers.

Fig. 4 depicts the architecture of the proposed VPS algorithm. There are three core components, namely actual flows {A_i}, virtual flows {V_i}, and the VPS controller, all implemented inside the sender's transport layer. The application interfaces with the VPS algorithm through standard network programming APIs such as sockets [6]. When the server application creates a new TCP flow, e.g., by creating a new stream socket [6] or by accepting an incoming connection through a stream socket, VPS will create an actual flow and a virtual flow internally. Therefore for each group of TCP flows controlled by the VPS, the same number of actual flows and virtual flows will be created. The following sections explain details of the two types of flows and the VPS controller that interfaces between them.

3.1 Virtual and Actual Flows

In ordinary TCP when the application submits data for transmission, TCP will construct one or more TCP segments and then submit them, subject to the control of the congestion window and the receive window, to the IP layer for delivery. The congestion window, denoted by *cwnd* is computed by the sender using the well-known Additive Increase Multiplicative Decrease (AIMD) algorithm [9] and this window limits how much unacknowledged data the sender can send at a time. Upon receiving data correctly, the receiver will send an acknowledgement packet (ACK) back to the sender, informing the sender of the highest sequence number received, denoted by S_{max} as well as the size of the receive window, denoted by *rwnd*, for flow control purpose. After receiving the ACK the sender can then send more TCP segments up to the sequence number $S_{max}+\min\{cwnd, rwnd\}$.

TCP's congestion control algorithm is instrumental to TCP's fair bandwidth sharing property. The problem is that it also limits the amount of data a TCP flow can transmit, and thus makes it impossible to transmit data faster than the fair share bandwidth, as is evident in the experiment in Section 2.1. This congestion control algorithm contradicts the goal to differentiate bandwidth among a group of TCP flows. Therefore in VPS the virtual flows are only responsible for running the standard TCP congestion control algorithms to determine how many TCP segments can be transmitted based on information received from ACKs and timeout events, and generate virtual packets accordingly. The VPS controller then collects the virtual packets from all virtual flows and then distributes them to the actual flows according to the desired bandwidth ratios, thus allowing some actual flows to send data *faster* than the fair share bandwidth. Note that there is no fixed mapping between virtual flows and actual flows. Virtual packets generated by a virtual flow can be redistributed to any of the actual flows, and an actual flow can receive virtual packets from more than one virtual flows.

Now as the virtual flows generate virtual packets based on the same congestion control algorithm as that in ordinary TCP, the total rate at which virtual packets are generated by all *n* virtual flows will mimic those of *n* ordinary TCP flows. Thus although the actual flows may send data at different rates, the aggregate throughput of all *n* actual flows will still mimic the aggregate throughput of *n* ordinary TCP flows and so VPS flows *as a group* can still maintain TCP's fair bandwidth sharing property.

On the other hand, each actual flow interfaces with a socket to receive data from the application for transmission. The actual flow implements the rest of the TCP functions, including buffering, data transmission, RTT estimation, timeout and retransmission. Note that when an actual flow receives an actual ACK from the receiver it will pass it to the VPS controller, which then performs ACK translation to generate virtual ACK(s) for processing by the virtual flows. This will be explained in detail in Chapter 4.

To ease discussion we adopt the terminologies {[actual|virtual] [new|old|duplicate] [packet|ACK]} to represent the packets and ACKs. Packets and ACKs from actual flows and virtual flows are labeled with actual and virtual respectively. Packet is equivalent to data packet or TCP segment. New packets have sequence number higher than the maximum sequence number sent so far; otherwise the packets are called old packets. On the other hand, new ACKs have ACK number higher than the sender's highest ACK number, duplicate ACKs have ACK number equal to the sender's highest ACK number, and old ACKs have ACK number smaller than the sender's highest ACK number. Different kinds of packets and ACKs are processed in different ways and the details will be explained in Chapter 4 and 5.

3.2 VPS Controller

The VPS controller is the bridge between virtual flows and actual flows. It has two functions. First, it collects virtual packets from the virtual flows and then redistributes them to the actual flows according to the desired bandwidth ratios. For example, consider the case of two flows with bandwidth ratio of w_1:w_2=1:2, i.e., flow 1 and flow 2 are to receive 1/3 and 2/3 of the bottleneck bandwidth respectively. Then for every three virtual packets generated by the two virtual TCP flows $\{V_1,V_2\}$, the VPS controller will redistribute one of them to actual flow A_1, and two of them to actual flow A_2. In this way the two actual TCP flows will receive bandwidth according to their respective ratios. However there are considerable complications to this process and the details will be explained in Chapter 5.

Second, the VPS controller keeps track of the mappings between virtual TCP segments generated by virtual flows and the resultant actual TCP segments transmitted by actual flows so that incoming actual ACKs can be translated back into the corresponding virtual ACKs for congestion control processing by the virtual flows. Specifically, every time the VPS controller distributes a virtual packet to an actual flow by substituting a virtual TCP segment with an actual TCP segment, it creates a packet substitution record (PSR) with fields

$$\text{PSR} \triangleq \{A_{id}, A_{seq}, V_{id}, V_{seq}, S\}$$

where V_{id} and A_{id} are the virtual flow ID and actual flow ID respectively; V_{seq} and A_{seq} are the sequence numbers of the virtual TCP segment generated and the actual TCP segment transmitted respectively; and S is the state of the record, which can be in one of four states: unacknowledged (U), received (R), lost (L), or buffered (B). A PSR's state is initialized to the unacknowledged state upon creation. If the actual packet is determined to be lost (e.g., by receiving duplicate ACKs or 'holes' in SACK blocks [11]), the state will be marked as lost. When the ACK for the actual packet arrives, the state will be changed to received after the actual ACK is translated into an virtual ACK. However in some cases the ACK translation must be delayed to prevent triggering false fast retransmission at the virtual flows (see Section 4.4) and in such cases the state of the PSR will be changed to buffered.

To ease discussion we use the notation $V_i[j]$ to represent a virtual TCP segment generated by virtual flow i with virtual sequence number j; and $A_i[j]$ to represent an actual TCP segment transmitted by actual flow i with actual sequence number j. An ACK of $V_i[j]$ (or $A_i[j]$) means all packets up to $V_i[j]$ (or $A_i[j]$) inclusively are received successfully. For simplicity, TCP segments are considered to be of unit size so the sequence numbers for consecutive TCP segments are consecutive integers.

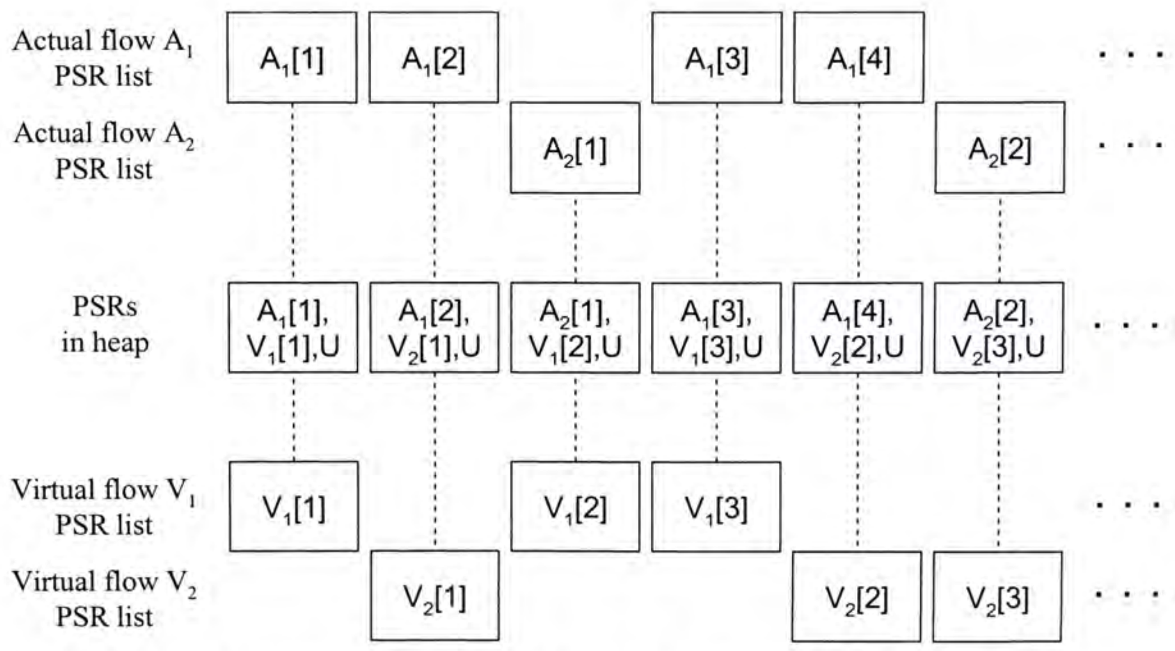


Fig. 5. Packet Substitution Records (PSRs) and the links to the PSR lists.

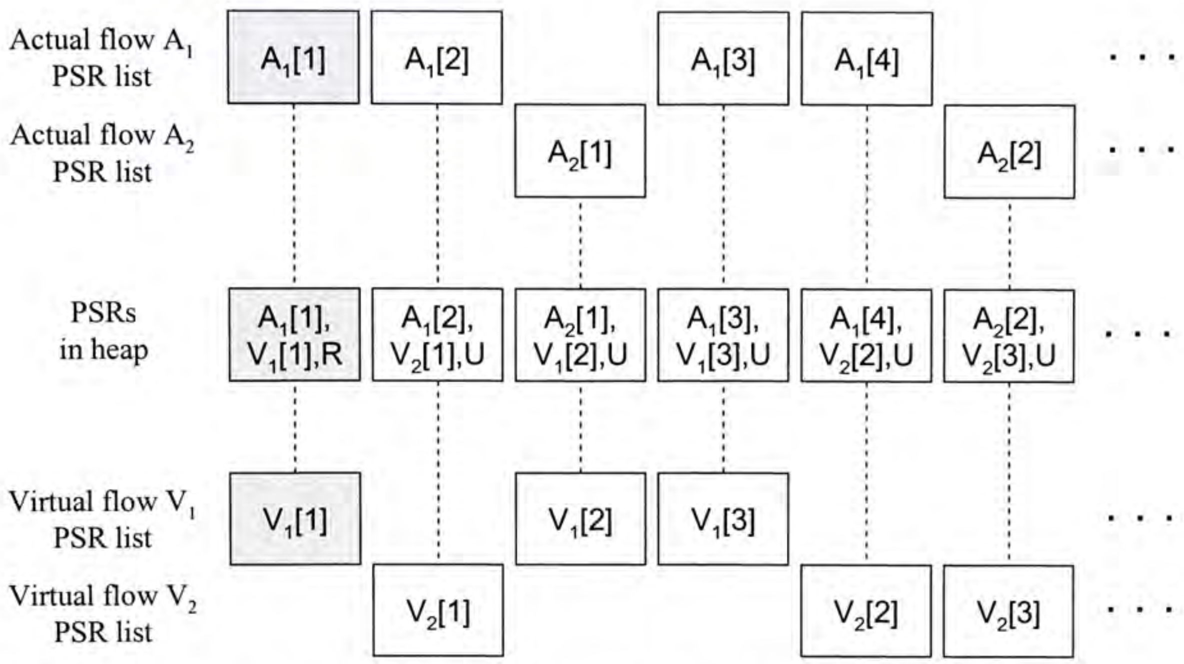


Fig. 6. A scenario where $A_1[1]$ is acknowledged (acknowledged PSRs are shaded in color).

Each PSR is linked to its associated virtual flow and actual flow upon creation. Fig. 5 illustrates the relation between PSRs and the virtual/actual flows. In this example, the application opens two TCP flows with the bandwidth ratios of w_1:w_2=1:2. If we consider the actual TCP segments in groups of three, i.e., {$A_1[1]$,$A_1[2]$,$A_2[1]$}, {$A_1[3]$,$A_1[4]$,$A_2[2]$}, etc., we can see how the VPS control distributes the virtual packets according to the bandwidth ratios. In each case, two of every three virtual packets are distributed to flow 2, and one to flow 1.

Now when an actual ACK is received, the VPS controller will lookup the corresponding PSR entry and generate a virtual ACK for virtual flow V_{id} with virtual sequence number V_{seq}. In other words the VPS controller performs a reverse substitution to translate the received actual ACK back to a virtual ACK for processing by the corresponding virtual TCP flow's congestion control algorithm. Consider the example in Fig. 6, when the ACK[1] of $A_1[1]$ is received the corresponding PSR is marked as received. Next, the VPS controller will generate a virtual ACK of $V_1[1]$ for virtual flow V_1. Similarly, when the ACKs of $A_1[2]$ and $A_2[1]$ arrives one after another, the VPS will generate virtual ACKs of $V_2[1]$ and $V_1[2]$ respectively.

There is, however, one more complication - most of today's Internet hosts run a variant of TCP called Reno with the SACK option [11] enabled by default[2]. With the SACK option the ACK packet may also include additional acknowledgements on discontinuous ranges of sequence numbers. The VPS in this case will generate, as needed, separate virtual ACKs for the corresponding virtual flows. Note that SACK blocks are stored in an option field within the TCP header and due to the option field's size limit a TCP ACK can contain at most three SACK blocks [11]. However, virtual ACKs generated by the VPS controller are not limited by the option field size and thus the VPS controller will generate as many SACK blocks as needed during ACK translation, which is described in detail in the next chapter. In the rest of the thesis the SACK option is assumed to be always enabled for both VPS and ordinary TCP.

---

[1] The ACK number of the ACK of $A_1[1]$ is 2, which is one unit larger than the highest continuously received sequence number and represents the next pursuing sequence number.

[2] According to the statistics [13], about 90% of the host in the Internet use Windows 98, 2000, XP and Linux as the operating system and all these operating systems enable the SACK option by default [14-16].

Chapter 4

ACK TRANSLATION

The ACK translation process described in Section 3.2 is only the simplest scenario, in which all packets are received successfully and the ACKs arrive in the same order as the corresponding transmitted packets. However in case of fast retransmit and fast recovery (Section 4.1), timeout (Section 4.2), and out of sequence delivery (Section 4.3 and 4.4), VPS will need to perform additional processing to ensure proper ACK translation and to reduce the generation of unnecessary virtual duplicate ACKs. On the other hand, it is also possible for one actual ACK to result in the generation of multiple virtual ACKs. This may increase data transmission burstiness and Section 4.5 discusses the maxburst mechanism to address the issue. Finally, Section 4.6 analyzes the memory overhead consumed by the PSRs and the computation complexity in ACK translation.

4.1 Fast Retransmit and Fast Recovery

Fig. 7 shows the partial state diagram for fast retransmit and fast recovery in TCP Reno with SACK. For simplicity, states and events not related to fast retransmit and fast recovery are omitted. When a flow receives three duplicate ACKs, it will enter the fast retransmit and fast recovery state. First, it retransmits the packet indicated by the highest ACK sequence number S_{max}, irrespective of the congestion window and the receive window. Second, it sets the maximum sequence number sent, denoted by S_{sent}, as the exit condition of the fast recovery state, denoted by *recover*. Third, it halves the congestion window *cwnd*. Fourth, it computes a variable called *pipe* [29] which counts the amount of unacknowledged data from

$$pipe = S_{sent} - S_{max} - \text{amount of data acknowledged in the scoreboard} \quad (1)$$

After the initialization, *pipe* will be increased whenever packets are sent and decreased whenever ACKs are received. The variable *pipe* together with *cwnd* acts to constraint data transmission during the fast recovery state – new or old packets can only be sent when *pipe* is smaller than *cwnd*. During the fast recovery state, old packets will first be retransmitted before sending any new packets. When the ACK number of received ACKs exceeds *recover*, the flow leaves the fast recovery state to return to the congestion avoidance state.

Fig. 8 depicts the state diagram of VPS actual flows for fast retransmit and fast recovery. It is similar to TCP where it enters the fast retransmit state upon receiving three actual duplicate ACKs, sets the variable *recover* to mark the exit condition of the fast recovery state, retransmits the actual packet indicated by the highest ACK number of the actual flow, and exits the fast recovery state when the ACK number of the newly arrived actual ACK exceeds *recover*. However, since actual flows in VPS do not perform congestion control, they do not compute or maintain the variables *pipe* and *cwnd* as in ordinary TCP, except for triggering temporary suspension to be discussed in Section 5.2.

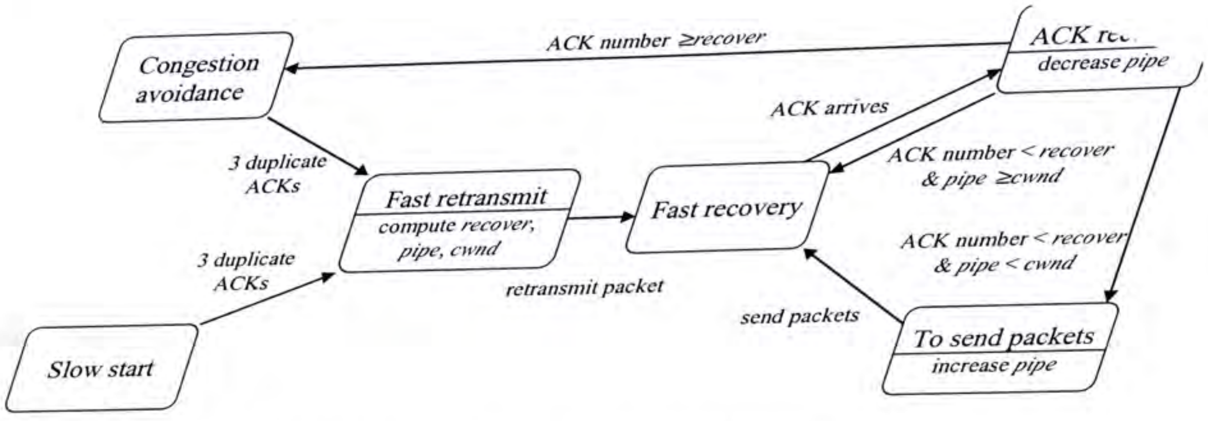
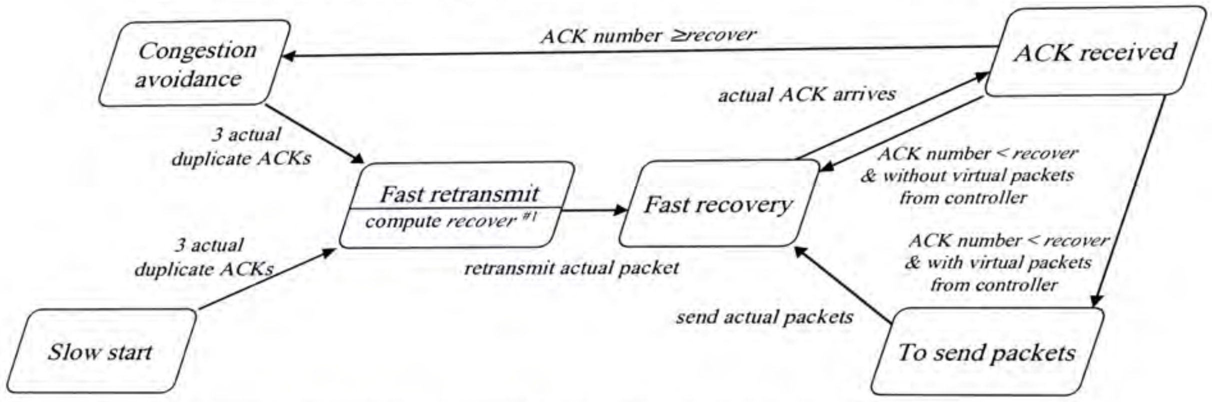


Fig. 7. Partial TCP state diagram for fast retransmit and fast recovery.



#1 *pipe* and *cwnd* are utilized for the temporary suspension of actual flows, see Chapter 5 for details.

Fig. 8. State diagram of VPS actual flows for fast retransmit and fast recovery.

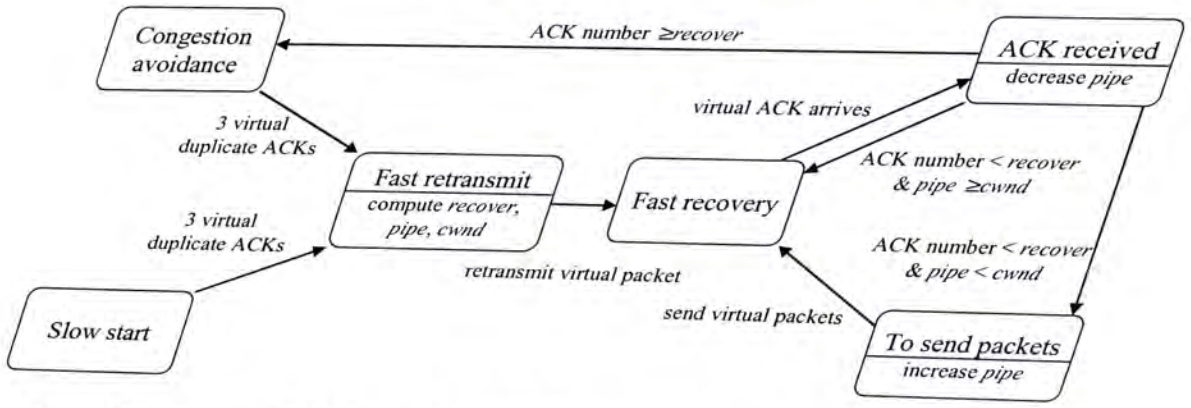


Fig. 9. State diagram of VPS virtual flows for fast retransmit and fast recovery.

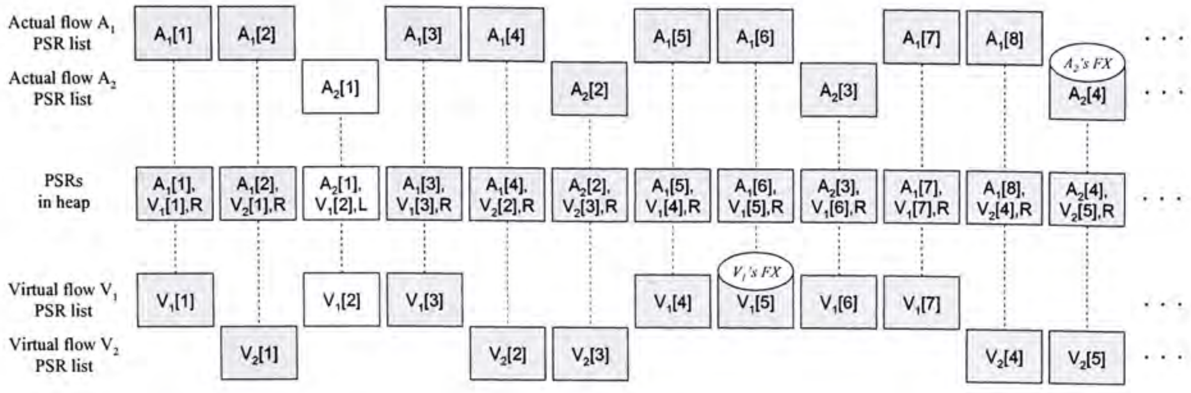


Fig. 10. All packets are acknowledged except $A_2[1]$ which is lost in the network. Acknowledged PSRs are shaded. The instants at which actual flow A_2 and virtual flow V_1 enter the fast retransmit and fast recovery state are labeled with circles.

Fig. 9 depicts the state diagram of VPS virtual flows for fast retransmit and fast recovery. It behaves in exactly the same way as TCP, except that it processes virtual ACKs generated by the VPS controller rather than actual ACKs sent by the receivers, and it generates virtual packets to the VPS controller rather than sending actual packets to the network.

Fig. 10 illustrates the operation of fast retransmit and fast recovery in VPS. The example assumes all packets are successfully received, except that $A_2[1]$ is lost. The receiver of $A_2[1]$ will send back duplicate ACKs upon receiving $A_2[2]$, $A_2[3]$ and so on until $A_2[1]$ is recovered. These duplicate ACKs will cause actual flow A_2 to enter the fast retransmit and fast recovery state.

Now consider actual flow A_1. After $A_1[1]$ and $A_1[3]$ are acknowledged, virtual flow V_1 will have its virtual packets $V_1[1]$ and $V_1[3]$ acknowledged. However $V_1[2]$ is not acknowledged as it is associated with the lost actual packet $A_2[1]$. In this case the VPS controller will generate a virtual duplicate ACK of $V_1[1]$ with SACK $\{V_1[3]\}$[3]

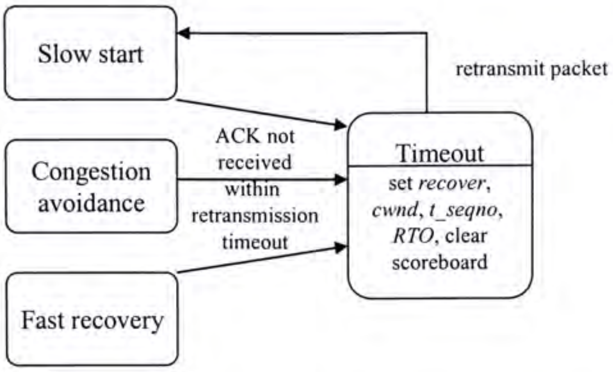
[3] The notation SACK $\{V_i[j]\}$ is used to acknowledge $V_i[j]$. The notation SACK $\{V_i[j]\text{-}V_x[y]\}$ is used to acknowledge all sequence numbers between $V_i[j]$ and $V_x[y]$ inclusively. The notation SACK

(except for cases to be discussed in Section 4.4). Similarly, upon receiving the ACK of $A_1[5]$ and the ACK of $A_1[6]$, the VPS controller will generate virtual duplicate ACKs of $V_1[1]$ with SACK $\{V_1[3]\text{-}V_1[4]\}$ and $V_1[1]$ with SACK $\{V_1[3]\text{-}V_1[5]\}$ respectively. As virtual flow V_1 has now received three duplicate ACKs of $V_1[1]$, it enters the fast retransmit and fast recovery state. As a result, the congestion window of virtual flow V_1 is halved and the virtual packet $V_1[2]$ is retransmitted.

4.2 Timeout

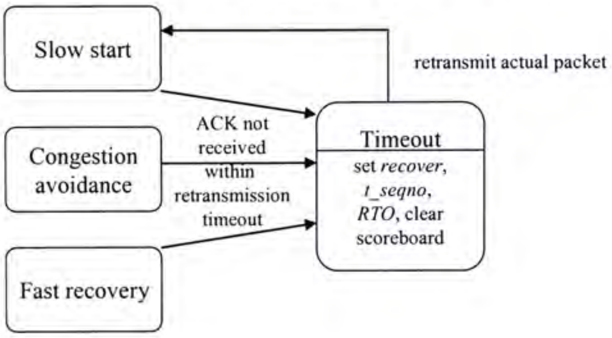
A timeout event occurs when a TCP sender does not receive the ACK of a transmitted packet within the retransmission timeout limit, denoted by *RTO* [21]. Fig. 11 depicts the partial state diagram for TCP timeout. After transiting to the timeout state, the flow sets the maximum sequence number sent S_{sent} as the exit condition of the slow start state, denoted by *recover*. Moreover, it resets the congestion window *cwnd* to 1, sets the variable *t_seqno* pointing to the next sequence number to send to the highest ACK number S_{max}, and doubles the current *RTO*. Furthermore, it clears the SACK scoreboard (i.e., zeroing all of the SACK bits) since the timeout might indicate that the receiver has reneged [11]. Finally it retransmits the packet indicated by the highest ACK number.

$\{V_i[j]\text{-}V_x[y], V_p[q]\text{-}V_r[s]\}$ is used to acknowledge discrete ranges of sequence numbers between $V_i[j]$ and $V_x[y]$, and between $V_p[q]$ and $V_r[s]$ inclusively. In real TCP SACK implementation, SACK information is stored in multiple SACK blocks if necessary. Each SACK block has a head and a tail sequence number. All sequence numbers between the head (inclusive) and the tail (exclusive) sequence numbers are acknowledged.



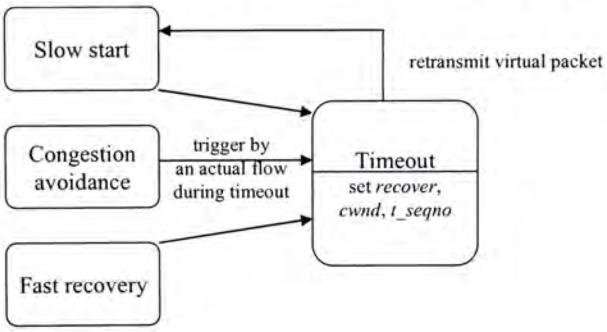
Slow start
retransmit packet
ACK not received within retransmission timeout
Congestion avoidance
Timeout
set *recover*, *cwnd*, *t_seqno*, *RTO*, clear scoreboard
Fast recovery

Fig. 11. Partial TCP state diagram for timeout.



Slow start
retransmit actual packet
ACK not received within retransmission timeout
Congestion avoidance
Timeout
set *recover*, *t_seqno*, *RTO*, clear scoreboard
Fast recovery

Fig. 12. Partial state diagram of VPS actual flows for timeout.



Slow start
retransmit virtual packet
trigger by an actual flow during timeout
Congestion avoidance
Timeout
set *recover*, *cwnd*, *t_seqno*
Fast recovery

Fig. 13. Partial state diagram of VPS virtual flows for timeout.

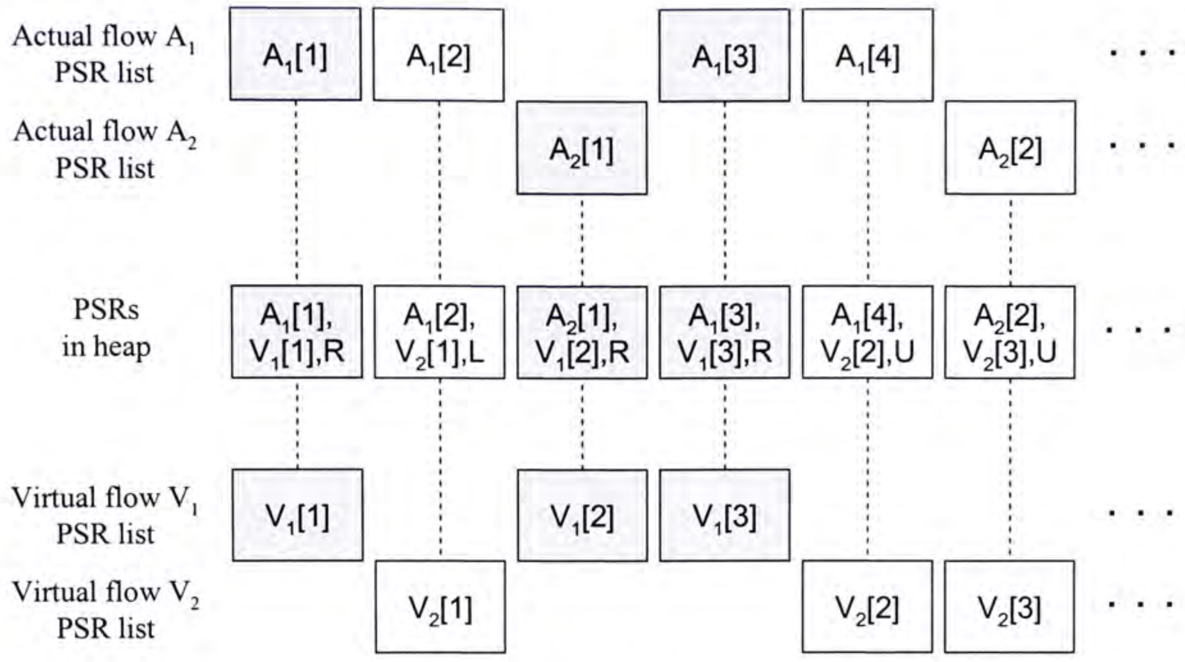


Fig. 14. A scenario where $A_1[1]$, $A_1[3]$ and $A_2[1]$ are acknowledged but $A_1[2]$ is lost (acknowledged PSRs are marked by shaded color).

Fig. 12 and 13 depict the state diagrams for timeout in VPS actual flows and virtual flows respectively. Actual flow operates similarly to TCP except that they do not set *cwnd*, as congestion control is performed by the virtual flows. A virtual flow sets *recover*, *cwnd* and *t_seqno* after transiting to the timeout state, and then retransmits the virtual packet indicated by the highest ACK number of the virtual flow. Note that virtual flow does not need to clear its scoreboard as that is done by the actual flows when resetting the corresponding PSRs to the unacknowledged state. Virtual flow also does not maintain the *RTO* (it is maintained by the actual flows) as RTT measurements, and thus timeouts, are meaningful only for actual packets. When an actual flow triggers timeout, the corresponding virtual flow in the PSR of the retransmitted packet will then be triggered into the timeout state as well.

Fig. 14 illustrates the timeout event in VPS. Packets $A_1[1]$ and $A_2[1]$ are acknowledged directly and $A_1[3]$ indirectly via the SACK{$A_1[3]$} block of duplicate ACK of $A_1[1]$. After ACK translation virtual packets $V_1[1]$, $V_1[2]$ and $V_1[3]$ will be acknowledged. Assume that $A_1[2]$ is lost and there is no further ACK received. After the retransmission timeout of actual flow A_1 expires, actual flow A_1 will trigger timeout, clearing $A_1[3]$ from the scoreboard, i.e., resetting its state to unacknowledged. Also, the packet indicated by the highest ACK number (i.e., $A_1[2]$) will be retransmitted. The virtual packet indicated in the PSR of $A_1[2]$ is $V_2[1]$, so virtual flow V_2 will be triggered to timeout, which resets the congestion window to 1 and retransmits virtual packet $V_2[1]$.

4.3 Packet and ACK Reordering

Due to multi-path routing [27] and local parallelism [28], packets and ACKs are sometimes reordered in the network, which means the order of arrival is different from that of the original packet transmission. The following explains and compares the handling of packet and ACK reordering in TCP and VPS.

First consider the reordering of ACKs as illustrated in Fig. 15 and 16. When packets $A_1[3]$ and $A_1[4]$ arrive at the receiver, the receiver sends back the ACK of $A_1[3]$ and $A_1[4]$ to the sender. However suppose the two ACKs are reordered inside the network such that the sender now receives the ACK of $A_1[4]$ *before* the ACK of $A_1[3]$. In TCP, when it receives the ACK of $A_1[4]$[4] it increases the congestion window by one unit[5] and the highest ACK number by two units[6]. When the ACK of $A_1[3]$ arrives at a later time, it is regarded as an old ACK (i.e., the ACK number of the incoming ACK is lower than the flow's highest ACK number) and is simply discarded.

[4] The ACK of $A_1[4]$ covers the ACK of $A_1[3]$.

[5] The congestion window (*cwnd*) is increased by *k* in slow start and *k*/*cwnd* in congestion avoidance. If ACK counting [31] is used, *k* is equal to 1. If Appropriate Byte Counting (ABC) [19] is used, *k* is adjusted by the increase on the number of previously unacknowledged bytes with a upper bound.

[6] The highest ACK number increases by the amount of acknowledged data, which is 2 in this case.

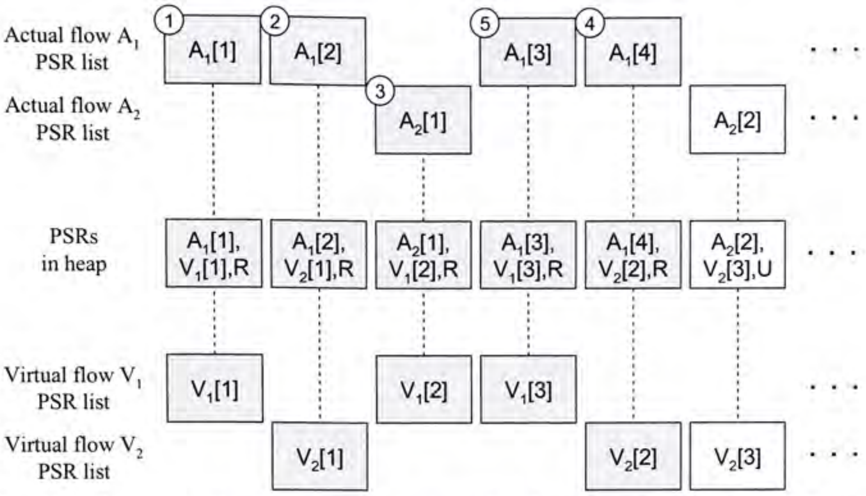


Fig. 15. A scenario where all packets are received but the order of ACK arrival is different from that of packet transmissions. The order of ACK arrival is indicated in small circles and acknowledged PSRs are marked by shaded color.

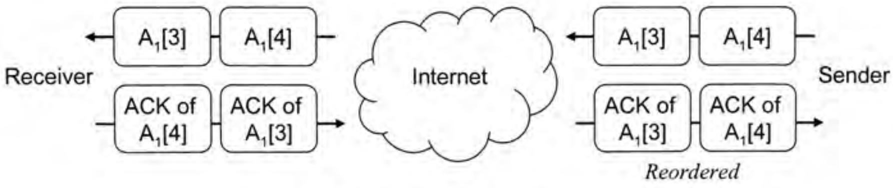


Fig. 16. ACK reordering.

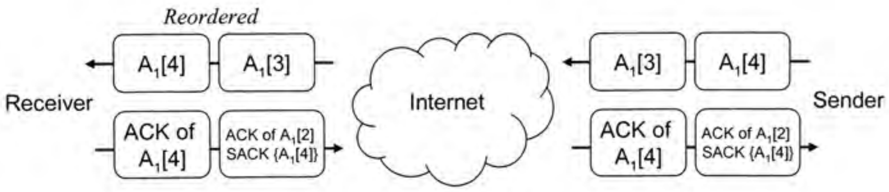


Fig. 17. Packet reordering.

In VPS, when it receives the ACK of $A_1[4]$, it generates virtual ACK of $V_1[3]$ and virtual ACK of $V_2[2]$. Both ACKs increase the congestion window and the highest ACK number of their respective virtual flow by one unit. When the ACK of $A_1[3]$ arrives at a later time, it is also regarded as an old ACK and is discarded. In this example, the aggregate increment of the congestion window upon the two concerned ACKs is two in VPS but one in TCP. However, if $A_1[3]$ and $A_1[4]$ are mapped to the same virtual flow, then only one virtual ACK will be generated and the behavior of VPS will be the same as TCP. Also if Appropriate Byte Counting [19] is used instead of ACK counting [31], the discrepancies will also be eliminated.

Next consider the reordering of packets as shown in Fig. 17. When $A_1[3]$ and $A_1[4]$ are reordered inside the network such that $A_1[4]$ arrives at the receiver first, then the receiver sends back an ACK of $A_1[2]$ with SACK $\{A_1[4]\}$ to the sender. Afterwards, when $A_1[3]$ arrives, the receiver sends back an ACK of $A_1[4]$. In TCP, when it receives the ACK of $A_1[2]$ with SACK $\{A_1[4]\}$, it will increase the duplicate ACK counter by one and update the scoreboard to record that $A_1[4]$ has been received. Both the congestion window and the highest ACK number remain unchanged. When the ACK of $A_1[4]$ arrives at a later time, the flow increases the congestion window by one unit and the highest ACK number by two units. In addition, the duplicate ACK counter is reset and the "hole" (i.e., $A_1[3]$) in the scoreboard is filled.

In VPS, when it receives the ACK of $A_1[2]$ with SACK $\{A_1[4]\}$, it changes the PSR state of $A_1[3]$ from unacknowledged to lost and generates a virtual ACK of $V_2[2]$. This ACK increases the congestion window and the highest ACK of virtual flow V_2 by one unit. When the ACK of $A_1[4]$ arrives later, it changes the PSR state of $A_1[3]$ from lost to received and generates a virtual ACK of $V_1[3]$. This ACK also increases the congestion window and the highest ACK of virtual flow V_1 by one unit. Thus the total increment of the congestion window caused by the two ACKs is two in VPS but one in TCP. However, if $A_1[3]$ and $A_1[4]$ are mapped to the same virtual flow, the first ACK generated will be a duplicate ACK which does not change the congestion window and the highest ACK number, while the second new ACK will update the congestion window and the highest ACK number in exactly the same way as TCP.

4.4 False Duplicate ACK Suppression

In case multiple receivers have different RTTs to the sender the ACK translation process may generate many false duplicate ACKs, resulting in unnecessary fast retransmit and fast recovery in the virtual flows. Fig. 18 illustrates this scenario with two receivers, of which receiver 2 has shorter propagation delay (and thus RTT) than receiver 1. Assume the sender transmits actual packets in the order shown in Fig. 18. Now as receiver 2 has shorter RTT, actual packet $A_2[2]$ arrives at receiver 2 earlier than $A_1[3]$ and $A_1[4]$, which are both transmitted before $A_2[2]$. Consequently, the ACK of $A_2[2]$ arrives at the sender earlier than the ACKs of $A_1[3]$ and $A_1[4]$ as shown in Fig. 19.

While this RTT heterogeneity does not affect TCP, it may result in the generation of false duplicate ACKs in VPS. For example, although there is neither actual packet lost nor actual packet reordering, the ACK of $A_2[2]$ will nonetheless generates a virtual duplicate ACK of $V_2[1]$ with SACK $\{V_2[3]\}$, i.e., a false duplicate ACK. As the RTT difference between actual flow A_1 and actual flow A_2 is likely to persist, similar false virtual duplicate ACKs will be continuously generated and thus incorrectly triggers fast retransmit and fast recovery in the virtual flows.

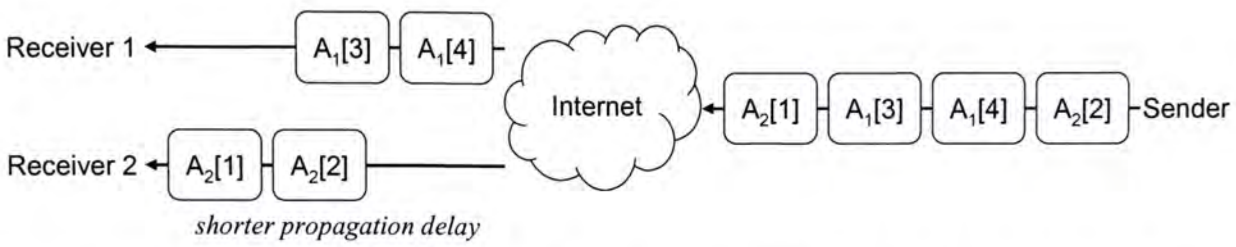


Fig. 18. The order of packet arrival at the receivers is different from that of packet transmission due to different propagation delays of the two receivers.

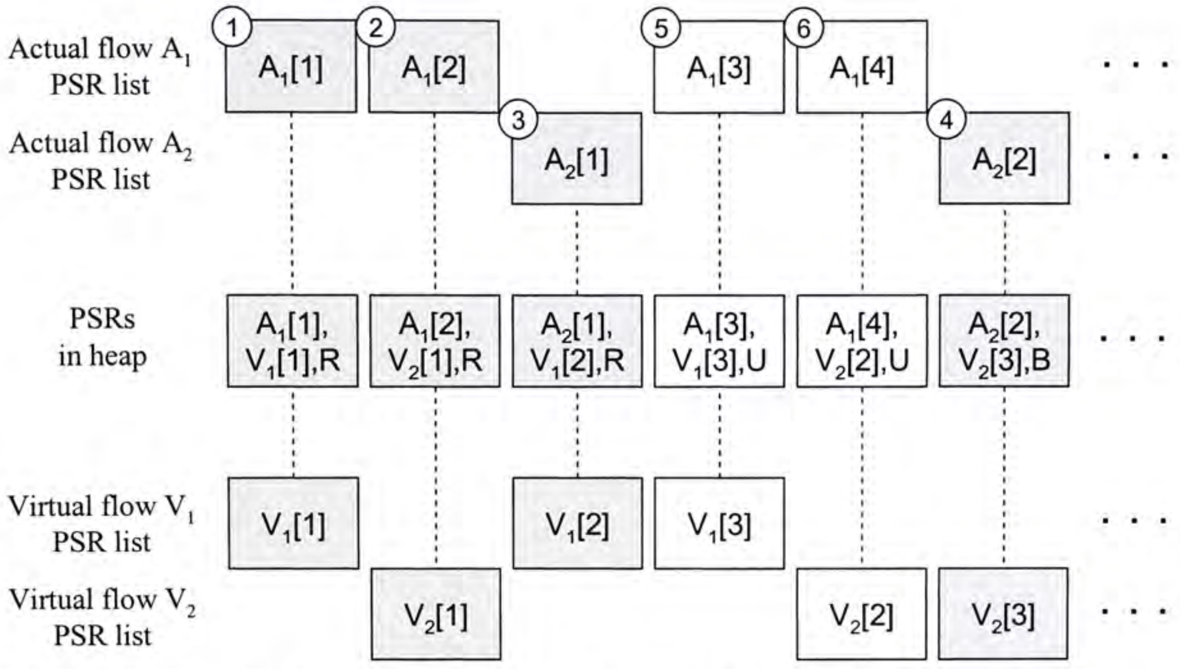


Fig. 19. A scenario where all packets are received but the order of ACK arrival is different from that of packet transmission. The order of ACK arrival is indicated in small circles and acknowledged PSRs are marked by shaded color.

To suppress these unnecessary duplicate ACKs the VPS controller will distinguish between lost and unacknowledged actual packets when performing ACK translation. Virtual duplicate ACK will only be generated for lost actual packets but not unacknowledged actual packets. Consider the example in Fig. 19, when the ACK of $A_2[2]$ is received the VPS controller will not generate an ACK for $V_2[3]$ as $V_2[2]$ (or $A_1[4]$) is still in the unacknowledged state. Instead, the PSR state of $A_2[2]$ is set to *buffered*. Subsequently when the ACK of $A_1[4]$ arrives, the VPS controller will generate a virtual ACK of $V_2[2]$, follow by the buffered virtual ACK of $V_2[3]$. The PSR state of $A_1[4]$ and $A_2[2]$ are then both set to *received*.

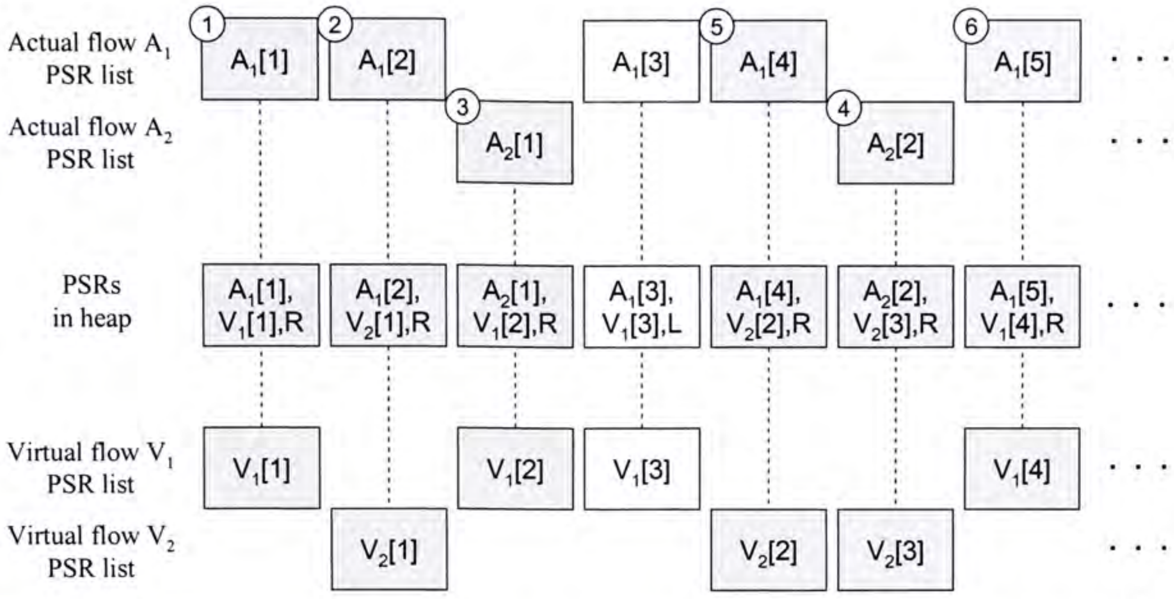


Fig. 20. A scenario where all packets are received except $A_1[3]$ but the order of ACK arrival is different from that of packet transmission. The order of ACK arrival is indicated in small circles and acknowledged PSRs are marked by shaded color.

Note that the above suppression algorithm does not affect true duplicate ACKs. Consider the example in Fig. 20, all packets are received except $A_1[3]$ which is lost in the network. When the ACK of $A_1[2]$ with SACK $\{A_1[4]\}$ arrives, the VPS controller sets the PSR state of $A_1[3]$ to *lost*. Subsequently when the ACK of $A_1[2]$ with SACK $\{A_1[4]\text{-}A_1[5]\}$ arrives, the VPS controller will not suppress the duplicate virtual ACK for $V_1[3]$ as the corresponding actual packet $A_1[3]$ has already been declared lost. Therefore duplicate virtual ACKs triggered by lost actual packets are not affected by the suppression algorithm.

4.5 Maxburst

In both ordinary TCP and VPS an ACK may trigger the transmission of multiple packets and in extreme cases, may increase the burstiness of the outgoing traffic. To limit the burstiness Floyd [24] proposed the Maxburst mechanism to limit the maximum number of data segments that can be transmitted in response to any given

ACK. This Maxburst mechanism is adopted in VPS, with a maximum burst size of 4 as used in the study by Venkata N. Padmanabhan *et al.* [25].

4.6 Memory Overhead and Computation Complexity

Compared to ordinary TCP, VPS incurs additional memory overhead to maintain the PSRs and additional computations to perform ACK translation. Each PSR contains the actual flow number, the actual packet sequence number, the virtual flow number, and the virtual packet sequence number. Assuming four bytes for each of these four numbers and allocate one byte each for the state and for reference counting, then a PSR entry consumes 18 bytes memory storage. The PSRs can be implemented using data structures dynamically allocated from the heap. Each PSR entry is referenced by one actual flow and one virtual flow. A PSR entry can be deallocated when both the highest ACK sequence numbers of the actual flow and the virtual flow exceeds the PSR's actual and virtual sequence numbers respectively. Thus the number of active PSRs is related to the number of unacknowledged packets, which in turn is bounded from above by the product of the maximum window size multiplied by the number of actual flows in the system. As the maximum window size is a constant, the maximum memory overhead is proportional to the number of actual flows and thus of O(n), where n is the number of actual flows.

In addition to memory overhead, ACK translation needs to be performed for every actual ACK received, and so its computation complexity will be of significance in practice. When an actual ACK arrives, the VPS controller scans through the actual PSR list to look for the PSR entry corresponding to the newly arrived ACK, till the highest ACK number or the tail of the last SACK blocks. The typical size of a PSR list is the minimum of the flow's receive window and the congestion window, but the maximum size of a PSR list is bounded by the number of flows times the maximum size of the receive window. Since the maximum size of the receive window is a constant, the computation complexity of the scanning operation is of O(n), where n is the number of flows.

Therefore both worst-case memory overhead and computation complexity are of O(n), implying good scalability of the VPS algorithm. The typical overhead and complexity will be even smaller in practice. For example, in a simulation with 3 VPS flows and 3 TCP flows, and using the maximum receive window of 128 packets, the cumulative percentage of the number of iterations of scanning operation for each actual ACK received is plotted in Fig. 21. Note that in 40% of the cases only 2 iterations of scanning operation are needed to process an actual ACK. Moreover, 90% of the cases execute fewer than 20 iterations of scanning operation to process an actual ACK.

After scanning the PSR list the VPS controller generates one or more virtual ACKs. A non-delayed ACK typically acknowledges one packet and so the VPS controller generates one virtual ACK from one actual ACK. However, it is possible for an ACK to acknowledge multiple packets and virtual ACKs may also be buffered (c.f. Section 4.4). In these cases the VPS controller may generate multiple virtual ACKs from one actual ACK received. Fig. 22 plots the cumulative percentage of the number of virtual ACKs generated for every actual ACK received. The results show that in 99.7% of the cases one actual ACK translates into one virtual ACK, and in only 0.3% of the cases one actual ACK generates two virtual ACKs.

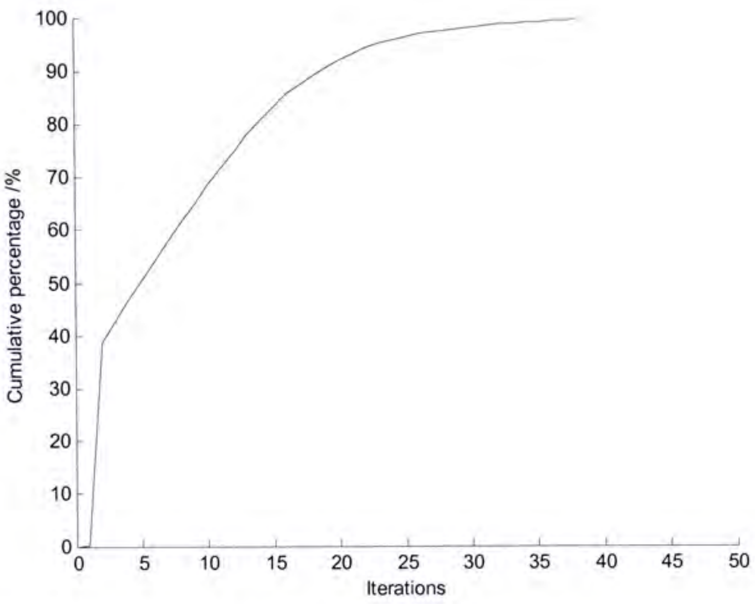


Fig. 21. Cumulative percentage of the number of iterations of scanning operation for every actual ACK.

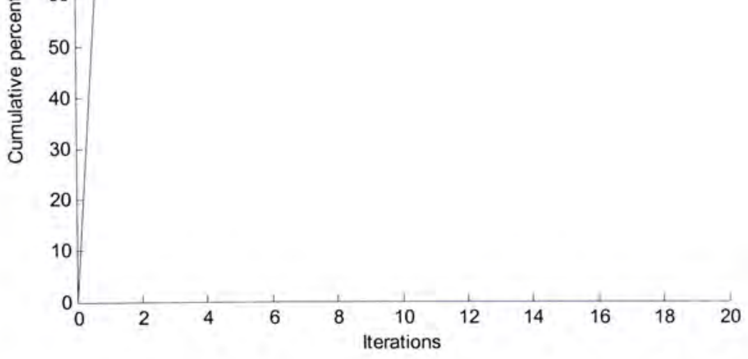


Fig. 22. Cumulative percentage of the number of iterations of ACK generation operation for every actual ACK.

Chapter 5

BANDWIDTH DIFFERENTIATION

VPS achieves bandwidth differentiation by a controlled substitution of virtual packets generated by the virtual flows to actual packets transmitted by actual flows. The following chapters present the algorithms employed by the VPS controller in distributing virtual packets to actual flows, discuss some subtle complexities in emulating the behavior of TCP flows, and describe the exceptional cases when data transmission is limited by constraints other than the congestion window.

5.1 Distribution of Virtual Packets

A virtual/actual flow may generate two types of virtual/actual packets, namely virtual/actual new packets and virtual/actual old packets. New packets refer to newly transmitted TCP segments using new sequence numbers while old packets refer to retransmitted TCP segments using old sequence numbers. Given a virtual old packet the VPS controller will only substitute an actual old packet for retransmission, i.e., retransmission generated by the virtual flows triggers only retransmission in the actual flows.

In contrast, a virtual new packet may trigger the transmission of either actual new packets or actual old packets. In the former case a new PSR entry will be created to record the mapping between the virtual new packet and the actual new packet for subsequent ACK translation. The latter case occurs when the selected actual flow has retransmission pending and in this case the actual flow will first retransmit an actual old packet without consuming the virtual new packet, which will be reused in the next round of virtual packet distribution.

To reduce the burstiness of outgoing traffic the VPS controller distributes virtual packets to the actual flows using a weighed random distribution algorithm. Specifically, virtual packets are distributed in rounds and given the application-specified bandwidth ratios w_i's, the VPS controller will select actual flow *i* in the *j*th-round with probability

$$P_{i,j} = \frac{(V_{i,j} - \Delta_i)}{\sum_{\forall k \in \mathrm{K}} (V_{k,j} - \Delta_k)} \tag{2}$$

where $V_{i,j} = w_i \cdot j$ is the cumulative weight of actual flow *i* at the *j*th round and Δ_i is the cumulative number of virtual new packets distributed to actual flow *i*. When distributing a virtual new packet, K is the set of flows that: (a) have new packets to send; (b) are not limited by the TCP receive window; and (c) are not in the temporary suspension state (Section 5.2). When distributing a virtual old packet, K is the set of flows that: (a) have old packets to retransmit, and (b) are not in the temporary suspension state. When $V_{k,j} = \Delta_k, \forall k$, i.e., all actual flows have been distributed their application-specified share of bandwidth, the VPS controller will begin a new round of virtual packet distribution.

5.2 Temporary Suspension of Actual Flows

In TCP, when the network is congested and a packet is dropped in the network, the sender will receive duplicate ACKs and eventually trigger fast retransmit and fast recovery. During fast retransmit and fast recovery, TCP will halve the congestion window and compute the amount of unacknowledged data, denoted by the variable *pipe*, from (1) described in Section 4.1. The flow will decrease *pipe* upon receiving ACKs. Actual packets, new or old, can only be sent if *pipe* is smaller than the congestion window. When the flow has just entered the fast retransmit state, *pipe* is usually about twice the size of the congestion window. Thus it takes time to receive sufficient ACKs to decrease *pipe* to within the congestion window size, and during that time the flow is effectively suspended from transmission. This behavior helps alleviate network congestion by deferring retransmission to allow the network congestion to ease.

In VPS however, an actual flow transmits packets via receiving virtual packets from the VPS controller and thus is not subject to the same suspension mechanism as described earlier. To simulate this behavior in VPS an actual flow upon entering the recovery state will use the congestion window and *pipe*[7] from the corresponding virtual flow recorded in the PSR of the retransmitted packet to simulate the suspension behavior. If the corresponding virtual flow has not entered the fast retransmit state, the actual flow will estimate the congestion window size in fast retransmit state (i.e., half

[7] The actual flow will get local copies of *cwnd* and *pipe* from the corresponding virtual flow rather than sharing the variables. Also the actual flow and the virtual flow modify the variables independently.

of the current congestion window) and the value of *pipe* (i.e., current maximum sequence number – current highest ACK number – current size of SACK scoreboard). Otherwise the actual congestion window size and *pipe* will be computed and used directly. Similar to the virtual flows, the actual flow will decrease *pipe* upon receiving valid ACKs and leave the suspension state to resume normal transmission once *pipe* is smaller than the congestion window.

5.3 Receive Window Limit

The VPS controller distributes virtual new packets among actual flows that are able to send new or old packets. However, if the amount of unacknowledged data exceeds the actual flow's receive window, then it cannot send any more new packets. When an actual flow has an application-specified ratio much larger than the rest of the flows or when the bandwidth capacity is very large, the actual flow may run into the receive window limit. In this case the affected flow will not consume any more virtual packets (until the receive window become available again) and the unused virtual packets will be distributed to the rest of the actual flows. As a result the realized bandwidth ratios may not conform to the specification {w_i's} exactly. A simple solution is to enable the TCP window scale option [20] to increase the size of the receive window.

5.4 Limited Data Transmission

On the other hand if the application does not have data to transmit for a particular actual flow, then the actual flow will not participate in virtual packet distribution by the VPS controller. Similar to the case of running into receive window limit, the unused virtual packets will then be distributed to the rest of the eligible actual flows.

Thus for flows that have data to sent their bandwidth ratios in a round will still be maintained by the VPS controller, and the actual flows together will still behave in a TCP-friendly manner.

Chapter 6

PERFORMANCE EVALUATION

This chapter evaluates the performance of VPS using simulations and compares it to the standard TCP Reno [5, 12] and the MulTCP [8], of which the latter also supports bandwidth differentiation. To ease description we will use the name VPS flows to refer to traffic flows using the proposed bandwidth differentiation protocol and TCP flows to refer to traffic flows using the standard TCP protocol.

6.1 Performance Metric

To facilitate comparison, we define a metric to quantify the protocols' accuracy in allocating bandwidth according to the specified ratios. Let there be N flows with application-specified ratios $\{w_1,w_2,\ldots,w_N\}$. Let $\{r_1,r_2,\ldots,r_N\}$ be the actual throughput measured in the simulation. Then for each flow we compute its allocation accuracy, denoted by A_i, from

$$A_i = \frac{r_i / \sum_{k=1}^{N} r_k}{w_i / \sum_{k=1}^{N} w_k} \tag{3}$$

where the numerator is the actual proportion of bandwidth received and the denominator is the proportion as specified by the bandwidth ratios. Therefore if the protocol performs perfectly the two will be the same and the allocation accuracy will be equal to 1. A value smaller/larger than 1 implies that the actual bandwidth received is less/more than that specified by the bandwidth ratios.

To evaluate the overall accuracy for all flows we compute the overall allocation accuracy, denoted by A, from

$$A = \frac{\left(\sum_{i=1}^{N} A_i\right)^2}{N\sum_{i=1}^{N} A_i^2} \tag{4}$$

which ranges from 0 to 1, with 1 indicating perfect allocation and lower values indicating larger deviations from the specified bandwidth ratios.

To evaluate the protocols' bandwidth sharing property, we define a fairness ratio, denoted by F, from

$$F = \frac{\sum_{\forall i} r_i}{\sum_{\forall j} s_j} \tag{5}$$

where the numerator is the aggregate bandwidth of all VPS flows and the denominator is the aggregate bandwidth in the same simulation setup but with the VPS flows replaced by ordinary TCP flows. In our simulation the number of VPS flows and standard TCP flows are the same so a value of F=1 implies perfect fair sharing of bandwidth with competing TCP flows, and higher/lower values of F indicates that the VPS flows receive more/less bandwidth than ordinary TCP flows.

6.2 Simulation Setup

The simulator is developed using the NS2 version 2.28 simulator [10] with the network topology shown in Fig. 22. There are three types of network traffic, including N VPS flows, N Reno TCP flows and a UDP flow generating exponential background traffic at rate equal to 10% of the core bottleneck bandwidth. All flows pass through the same bottleneck with *y* Mbps bandwidth, and delay of 5, 25, and 50ms (labeled as short, mid and long respectively in the figures). The bottleneck link adopts the droptail queueing discipline. All other links have 100Mbps bandwidth and delay of 1, 5, and 10ms (again labeled as short, mid and long respectively in the figures). Unless specified otherwise, we use *N*=3 VPS flows with application-specified bandwidth ratios of {1, 2, 3} and three competing TCP Reno flows. For both TCP Reno and VPS flows we assume the sender always has data to send, and all senders and receivers have the SACK option enabled. Each simulation run lasts for 500s of simulated time and each point of data is an average of 30 simulation runs.

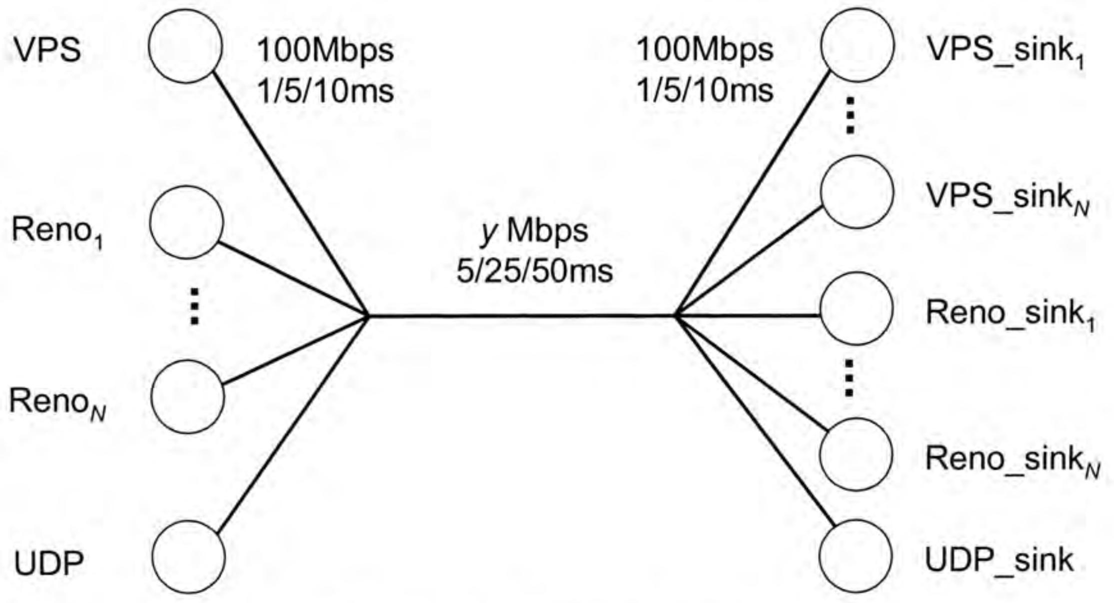


Fig. 23. The simulation topology.

6.3 Performance over Different Time Scales

In this simulation, we set the capacity of the core bottleneck to be 1Mbps and run the simulation for 1000 seconds. Fig. 24 and 25 show the throughput (averaged over

1-second intervals) against time for VPS and MulTCP flows. We observe that while there are short-term variations due to TCP's congestion control algorithm, all three VPS flows closely conform to the specified bandwidth ratio at all times, including periods of slow start and congestion avoidance. In comparison, the three MulTCP flows exhibit more throughput fluctuations, and in some cases a flow with higher bandwidth ratio (e.g., flow 3 at time 15s) may receive even less bandwidth than a flow with lower bandwidth ratio (e.g., flow 2).

This observation is due to the different ways VPS and MulTCP achieves bandwidth differentiation. In particular, MulTCP flows operate independently, each with a modified congestion control algorithm to achieve different sending rates. Thus packet loss events will affect only the corresponding MulTCP flow, which will then reduce its transmission rate. The other MulTCP flows, not being affected by the packet loss event, will continue to transmit at their current rate and so in the shorter time scale their bandwidth ratios may deviate from the specifications. By contrast, in VPS a packet loss event will affect the generation of virtual packets in a virtual flow but the effect will be reflected to all actual flows as the virtual packets are first pooled from all virtual flows before redistributing to the actual flows according to the application-specified bandwidth ratios.

Fig. 26 and 27 further compare the allocation accuracies for different *time scales*, i.e., the length of the averaging interval used in computing throughput. The results show that VPS flows achieve allocation accuracy close to 1 for all time scales, with coefficient of variation (CoV) close to zero. The minor deviations at short time scales (e.g., 1s) are due to variations during fast retransmit and fast recovery. By contrast, MulTCP exhibits significantly lower allocation accuracies at short time scales and cannot match VPS even at a time scale of 100 seconds. The CoV is also substantially higher than VPS in all time scales, suggesting larger deviations from the application-specified bandwidth ratios.

In the same experiment we also measured the fairness ratio of VPS, MulTCP and TCP Reno with SACK, and plotted the results in Fig. 28. In computing the fairness ratio the numerator of (5) uses the timescale under consideration (i.e., 1s, 2s, 5s, 10s, 20s, 50s or 100s) to calculate the throughputs, while the denominator of (5) uses the duration of the simulation (i.e., 1000s) to calculate the throughputs. We observe that VPS flows achieve fairness ratios close to 1, suggesting that the protocol can maintain fair bandwidth sharing with ordinary TCP flows. In comparison, fair bandwidth sharing is not part of the MulTCP protocol design goal and thus it is substantially more aggressive than ordinary TCP flows. This is undesirable as ordinary TCP flows sharing the same network bottleneck with MulTCP will suffer from lower bandwidth.

Fig. 29 plots the CoV of the fairness ratios. All three protocols show increased variation in fairness over shorter time scales. We conjecture that this is due to the short-term dynamics of TCP's congestion control algorithm in exploring the available network bandwidth and in reacting to short-term congestions triggered by packet loss. As all three protocols employ similar AIMD algorithm in congestion control they exhibit similar CoV in the fairness ratios.

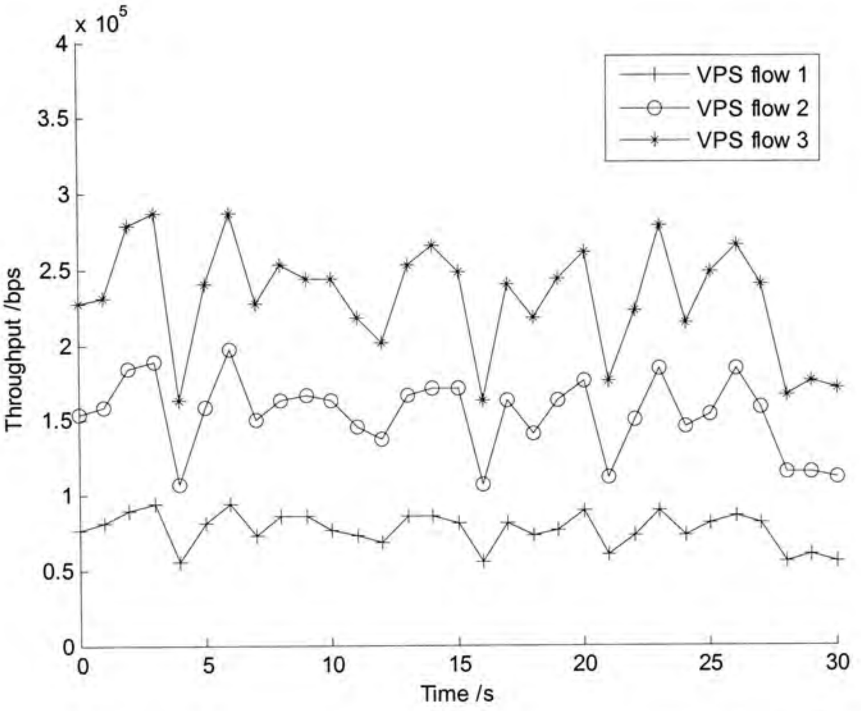


Fig. 24. Throughput averaged over 1-second interval against time for VPS flows.

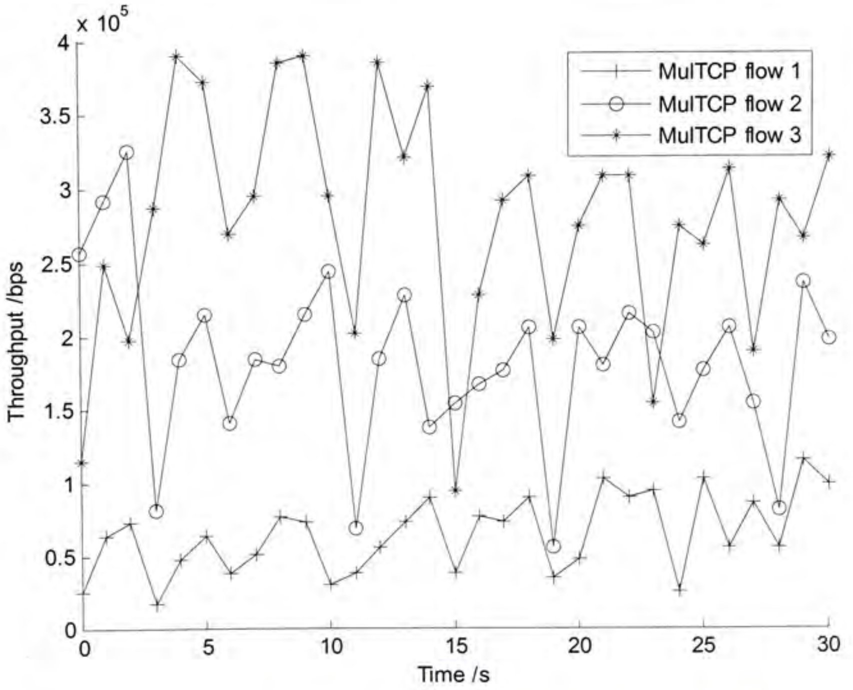


Fig. 25. Throughput averaged over 1-second interval against time for MulTCP flows.

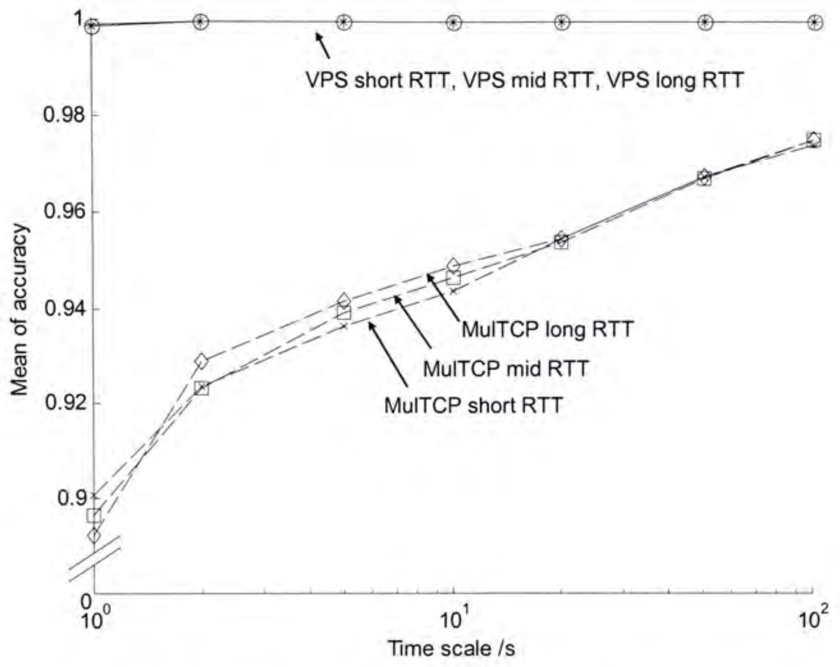


Fig. 26. Mean allocation accuracy for different time scales.

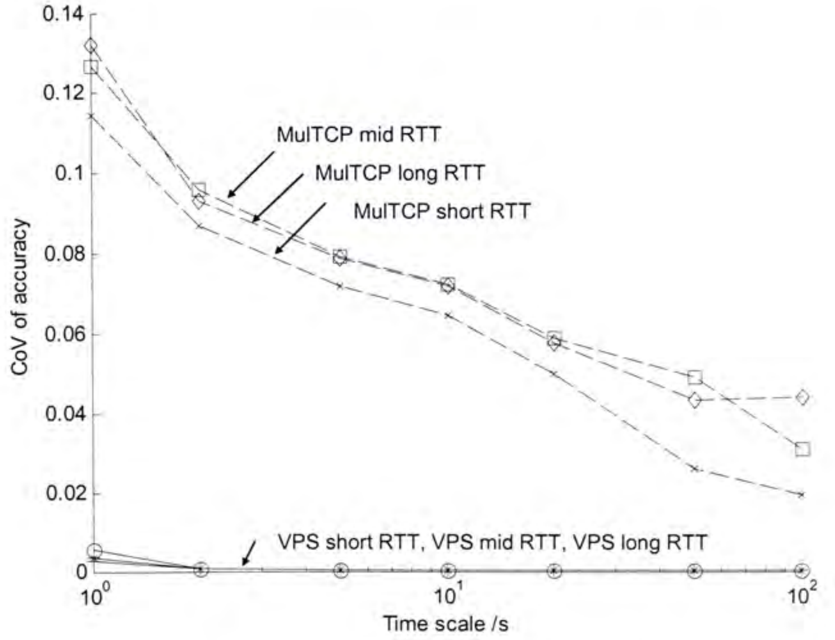


Fig. 27. Coefficient of variation of allocation accuracy for different time scales.

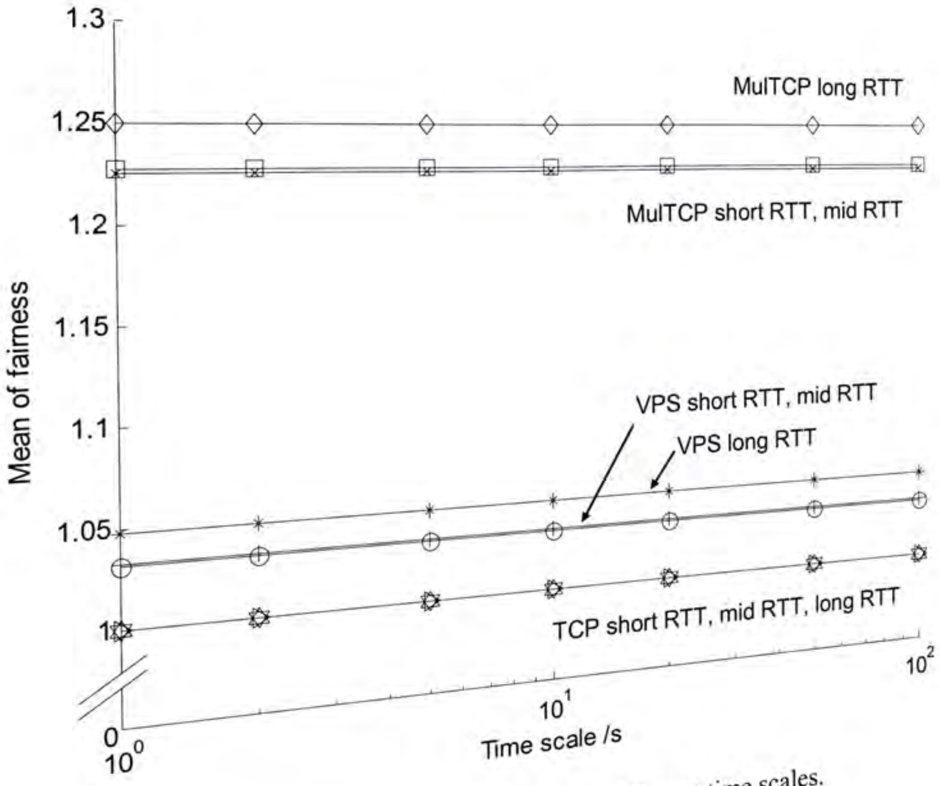


Fig. 28. Mean fairness ratio for different time scales.

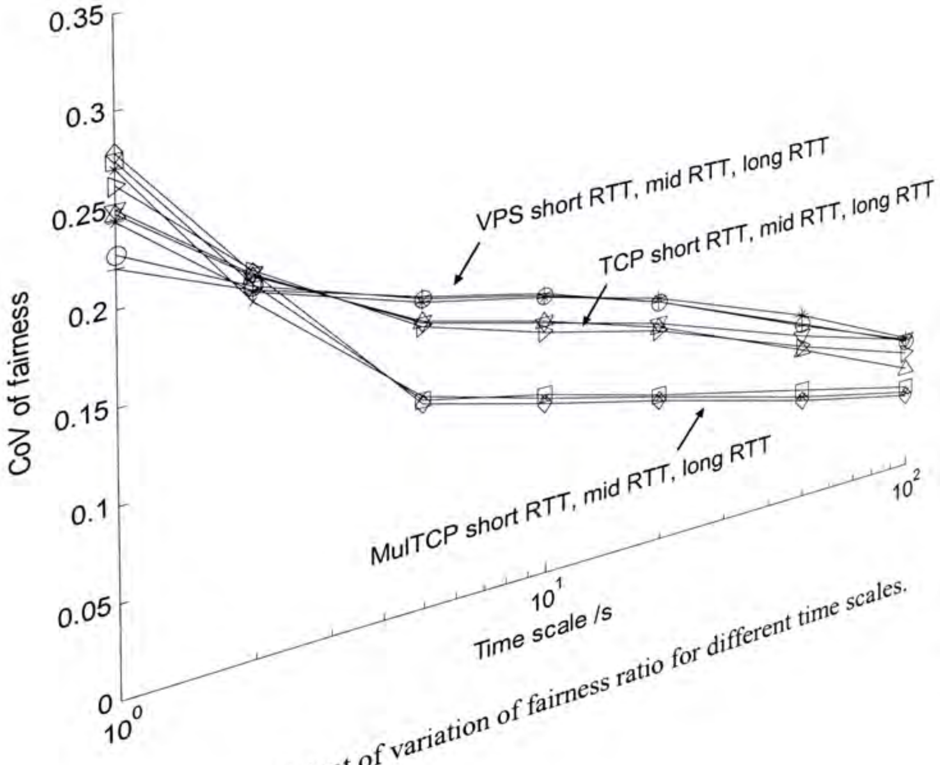


Fig. 29. Coefficient of variation of fairness ratio for different time scales.

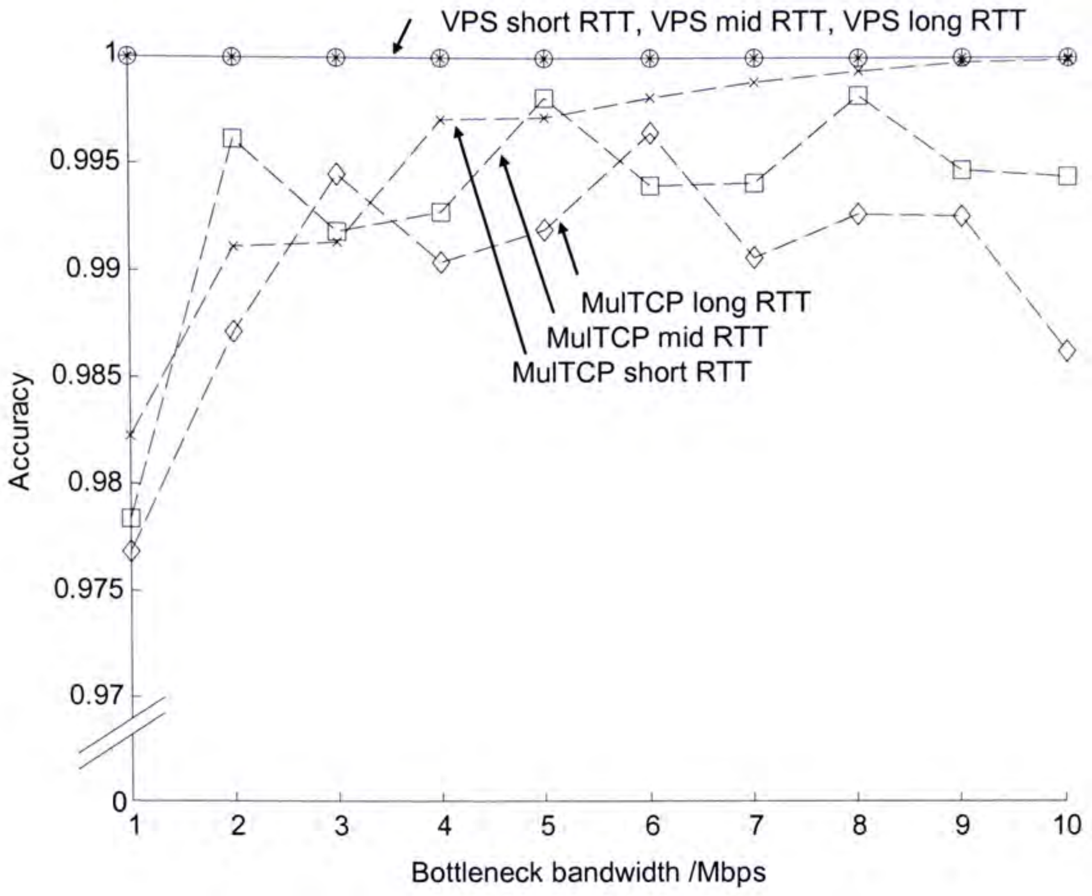


Fig. 30. Accuracy against bottleneck bandwidth.

6.4 Performance over Different Bottleneck Bandwidth

In this simulation, the capacity of the bottleneck link is varied from 1Mbps to 10Mbps to study the impact of the bottleneck bandwidth on the algorithms' performance. Fig. 30 and 31 plot the allocation accuracy and fairness respectively for different bottleneck bandwidth settings. VPS performs consistently in both metrics over bottleneck link capacity from 1Mbps to 10Mbps. By contrast, MulTCP not only has lower allocation accuracy, but the accuracy also deteriorates further at lower bottleneck bandwidth (e.g., at 1Mbps). The fairness performance of both VPS and MulTCP are consistent across the range of bottleneck bandwidth, with VPS similar to ordinary TCP and MulTCP considerably more aggressive than ordinary TCP.

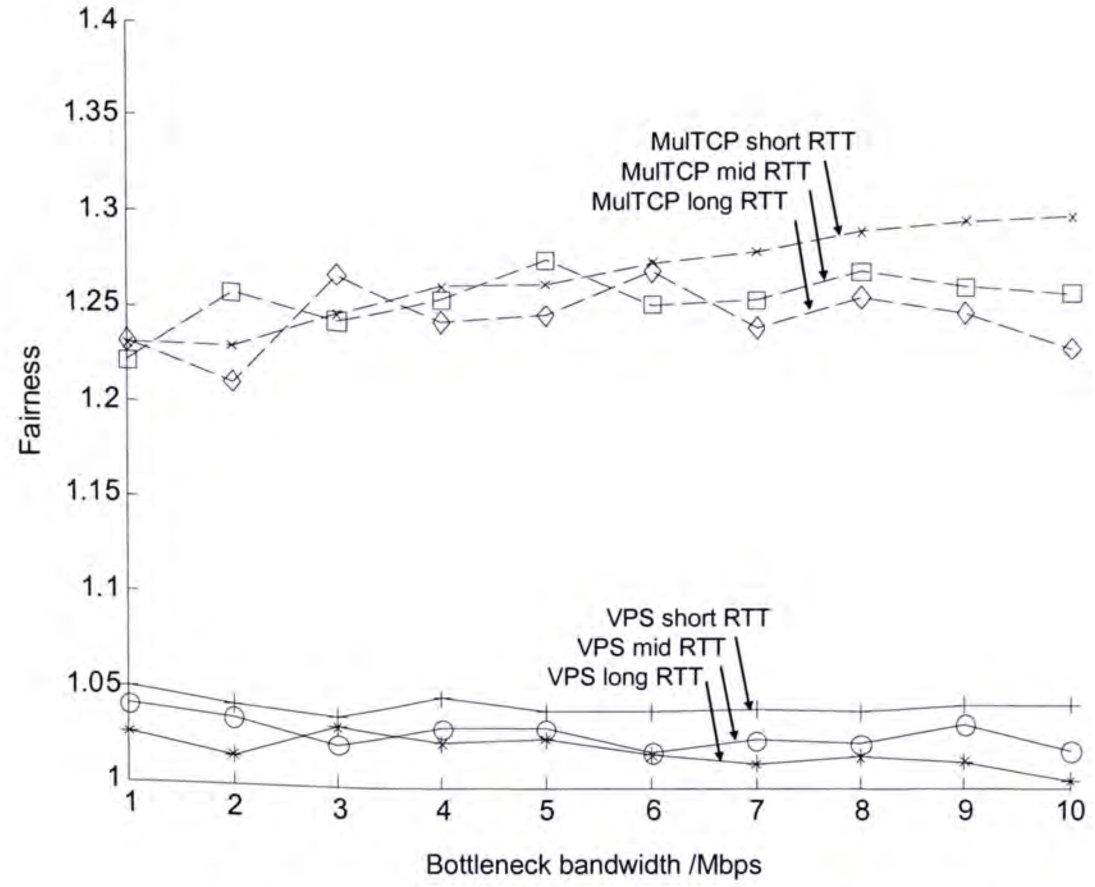


Fig. 31. Fairness against bottleneck bandwidth.

6.5 Performance over Different Application-specified Ratios

In another experiment, we vary the bandwidth ratios to investigate the protocols' performance when the bandwidth ratios are wider apart. We define a ratio differential, denoted by d, to set the bandwidth ratio for flow i to equal to w_i=1+d(i-1), where i=1, 2 or 3 and d varies from 1 to 10 in the simulation. Thus larger values of d will widen the difference of bandwidth ratios between successive flows.

Fig. 32 to 35 plot the allocation accuracy and fairness for VPS and MulTCP with bottleneck bandwidth of 1Mbps and 10Mbps. Again, the VPS flows perform consistently over the entire parameter range with negligible variations in both allocation accuracy and fairness. By contrast, the allocation accuracy of MulTCP decreases with wider bandwidth ratios as the higher-ratio MulTCP flows generate more bursty traffic, thus leading to more frequent packet loss. Also MulTCP becomes increasing more aggressive with wider bandwidth ratios as the higher-ratio MulTCP flows increase their congestion window at increasingly faster rates proportional to the bandwidth ratios.

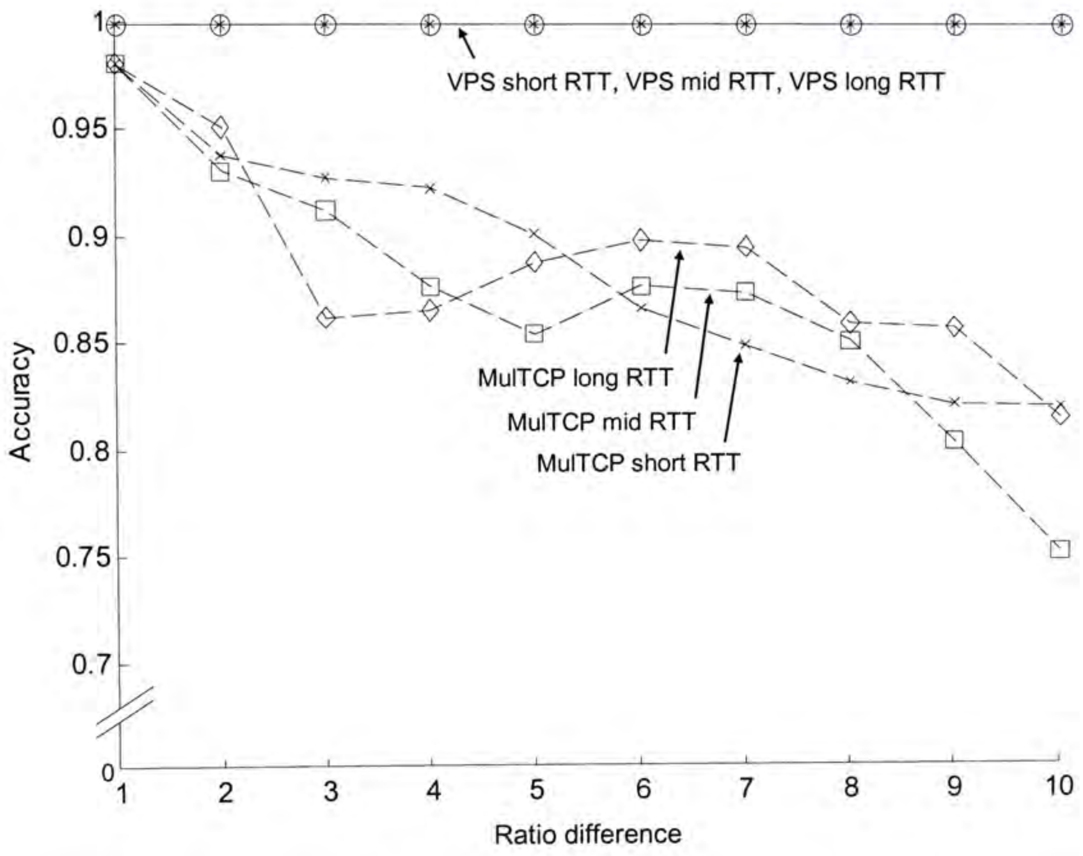


Fig. 32. Accuracy against ratio difference with bottleneck bandwidth of 1Mbps.

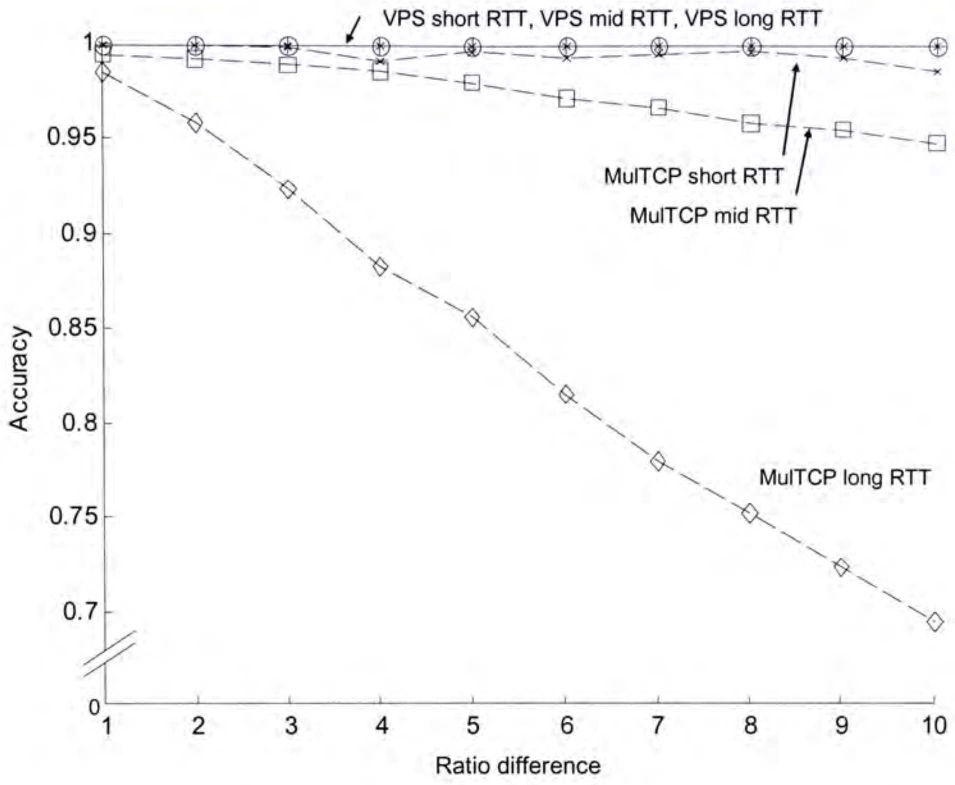


Fig. 33. Accuracy against ratio difference with bottleneck bandwidth of 10Mbps.

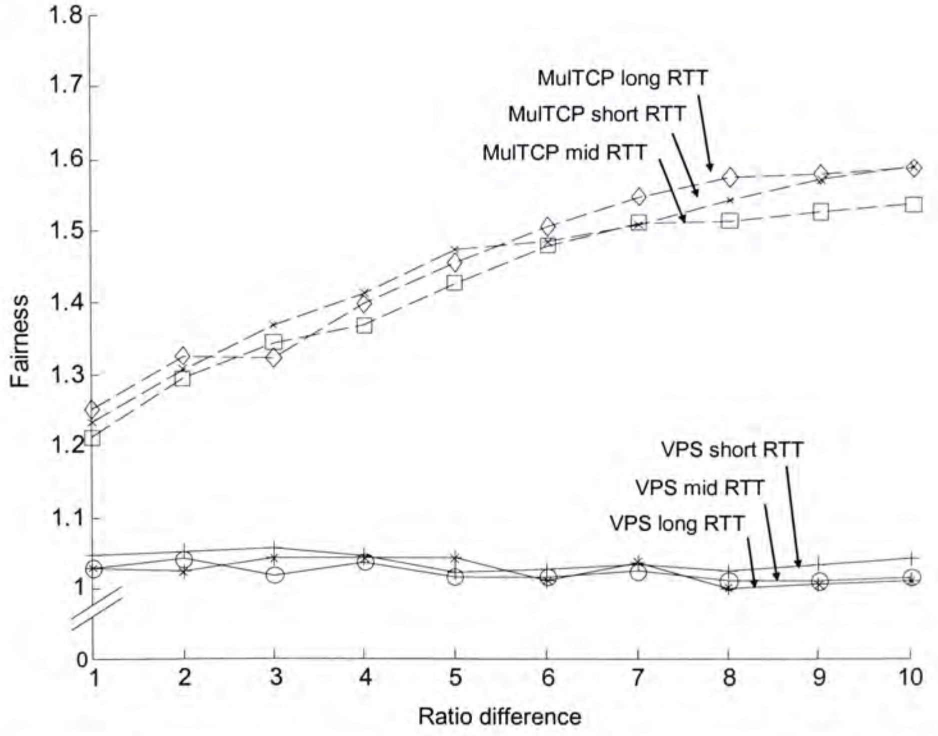


Fig. 34. Fairness against ratio difference with bottleneck bandwidth of 1Mbps.

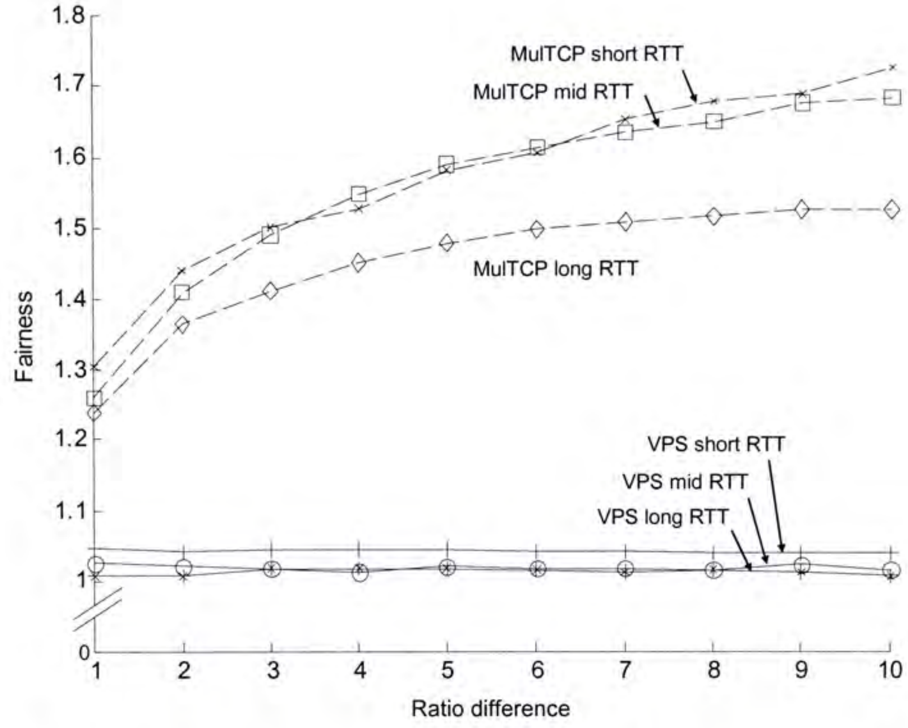


Fig. 35. Fairness against ratio difference with bottleneck bandwidth of 10Mbps.

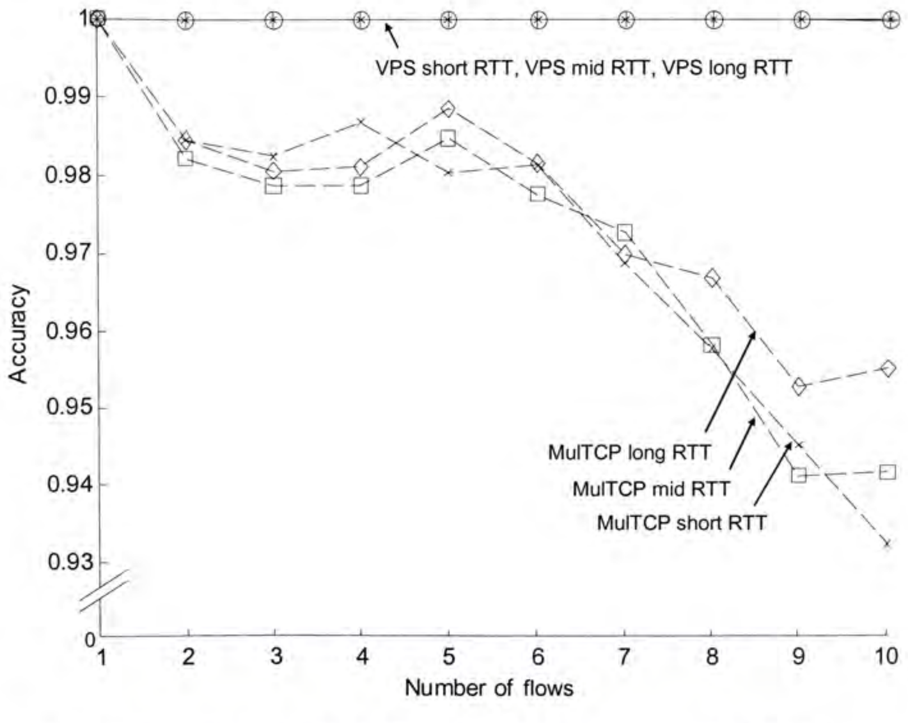


Fig. 36. Accuracy against number of flows with bottleneck bandwidth of 1Mbps.

6.6 Performance over Different Number of Flows

To investigate the protocols' scalability to more traffic flows, we re-run the experiments by varying the number of flows from 1 to 10 and plot the results in Fig. 36 to 39 showing the allocation accuracy and fairness of VPS and MulTCP with bottleneck bandwidth of 1Mbps and 10Mbps. In terms of allocation accuracy, VPS performs consistently in all cases while MulTCP's allocation accuracy decreases for more number of flows when the bottleneck bandwidth is low (1Mbps). For fairness ratio, VPS flows exhibit small increases in the fairness ratio at larger number of flows. In contrast, MulTCP flows exhibit significant increases in the fairness ratio when the number of flows increases from 1 to 3.

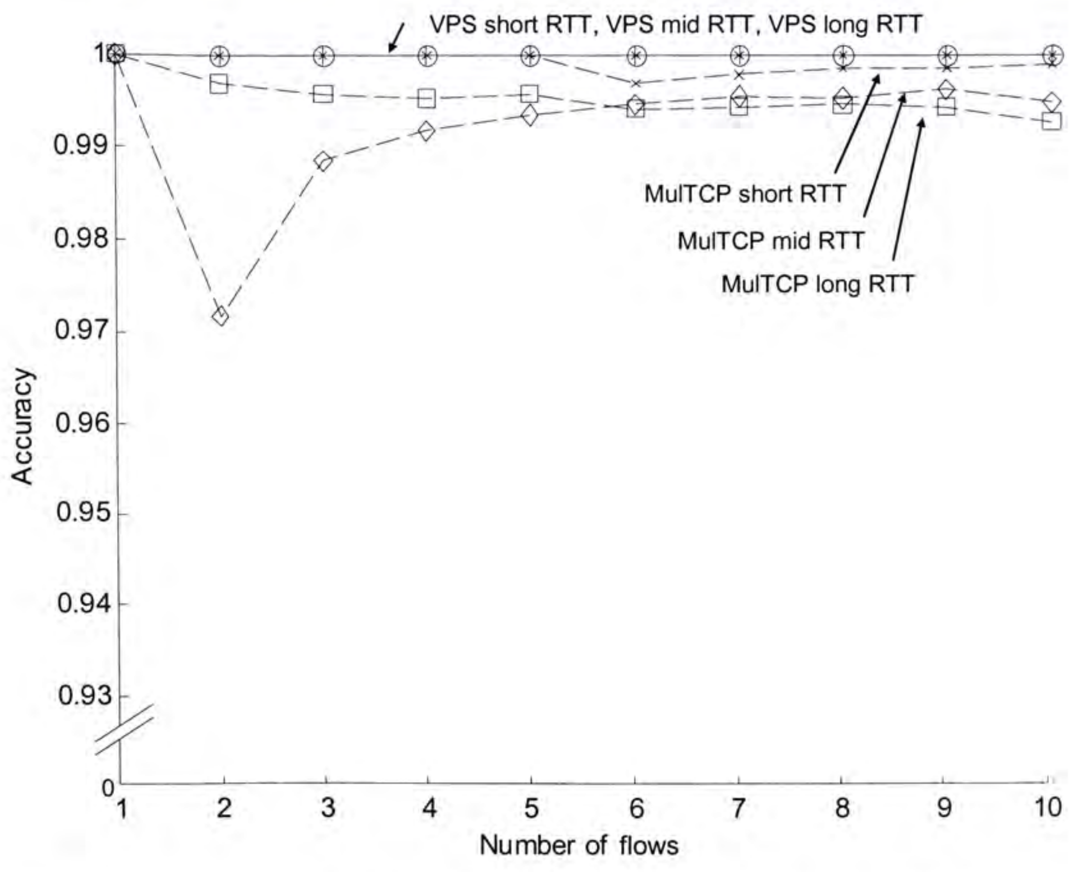


Fig. 37. Accuracy against number of flows with bottleneck bandwidth of 10Mbps.

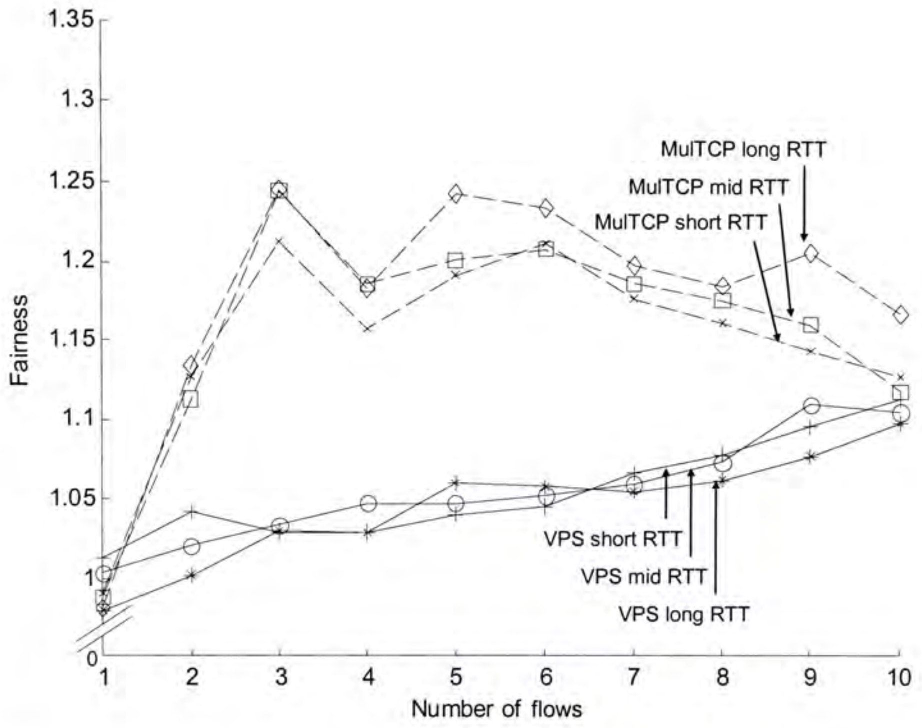


Fig. 38. Fairness against number of flows with bottleneck bandwidth of 1Mbps.

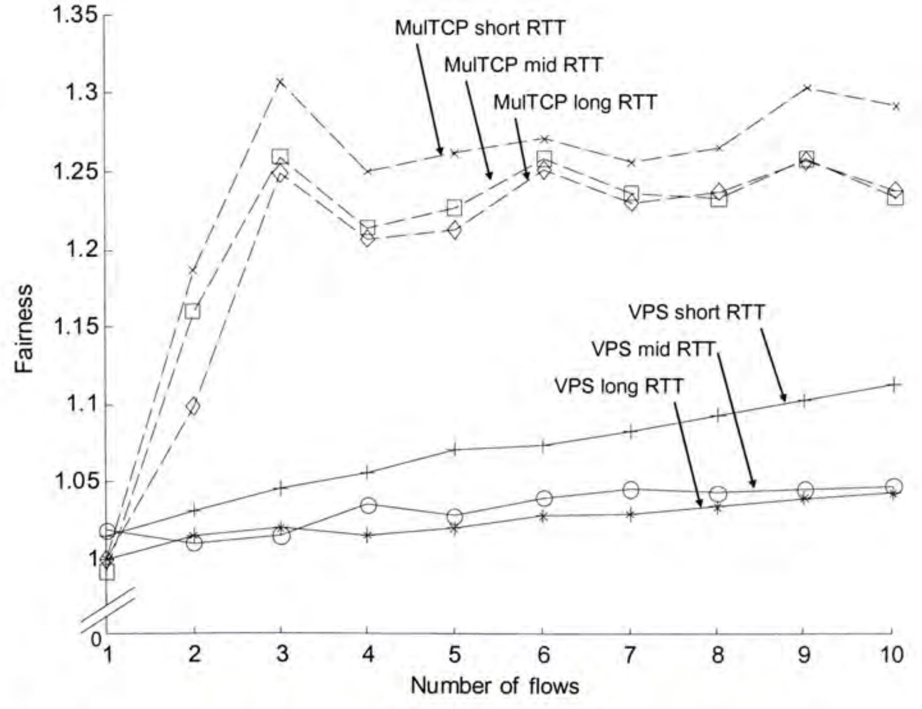


Fig. 39. Fairness against number of flows with bottleneck bandwidth of 10Mbps.

6.7 Heterogeneous Receivers

The previous simulations include setups with short, medium, as well as long RTTs. Nonetheless the RTT is the same for all receivers in the simulation. To investigate the effect of receivers with different RTTs, we conducted another set of simulations by setting the propagation delay of the last link to the receivers to $x+(i-1)y$, where $x=1$, 5 or 10 for short, intermediate and long RTT respectively; $i=1$, 2 or 3 for flow 1, 2 or 3; and y denotes the additional propagation delay factor and varies from 1 to 10 ms.

Fig. 40 to 42 plot the allocation accuracy versus the extra delay factor y for bottleneck bandwidths of 1, 5, and 10Mbps respectively. As expected VPS can achieve an allocation accuracy of 1 in all three cases. Fig. 43 to 45 plots the fairness ratio versus the extra delay factor y for bottleneck bandwidths of 1, 5, and 10Mbps respectively. The results show that receivers of heterogeneous RTT do have a negative impact on the fairness ratio of VPS. In particular, VPS flows become less aggressive under heterogeneous RTTs. We conjecture that this is due increases in triggering false retransmit in the virtual flows as the RTT differentials continually result in virtual packets arriving out of order. Further investigation will be needed to pinpoint the cause and to develop algorithms to compensate for RTT heterogeneity.

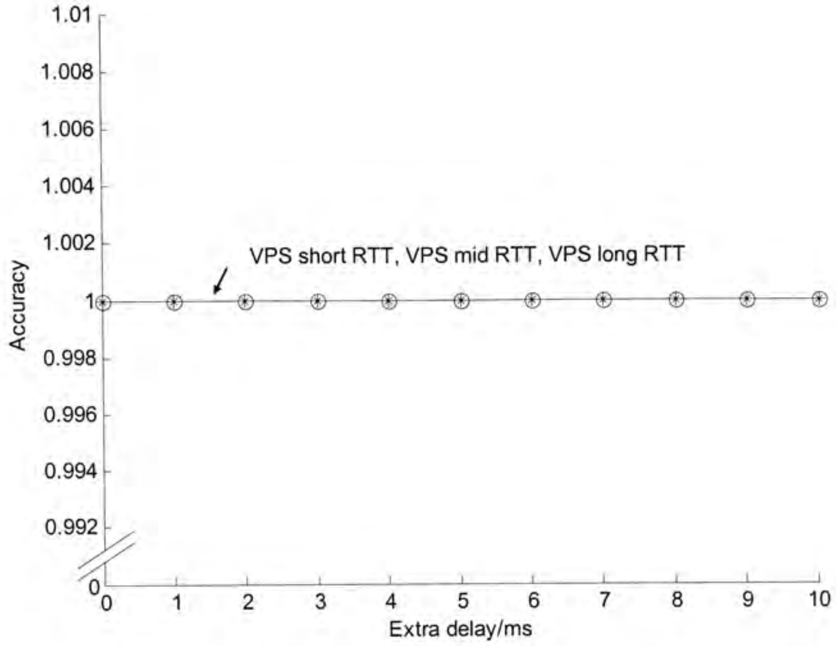


Fig. 40. Accuracy against extra delay factor with bottleneck bandwidth of 1Mbps.

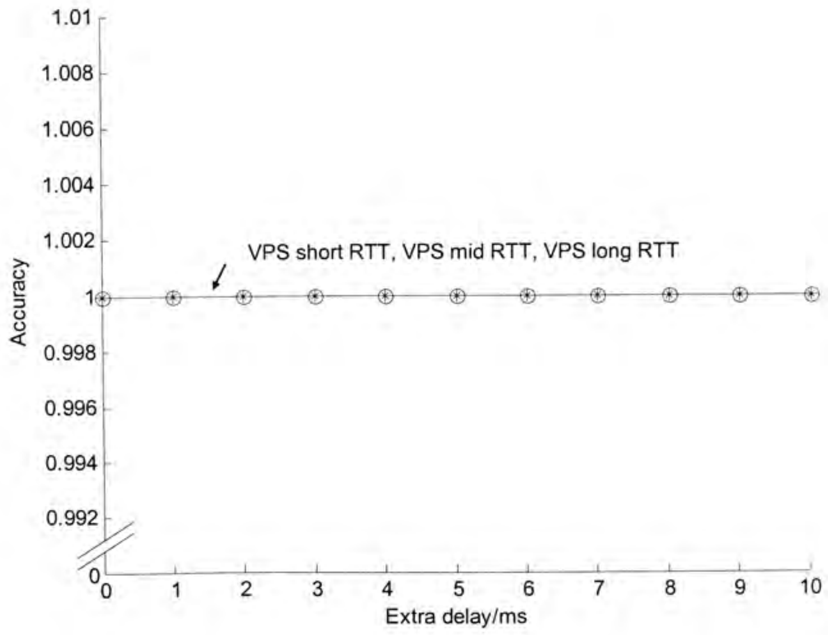


Fig. 41. Accuracy against extra delay factor with bottleneck bandwidth of 5Mbps.

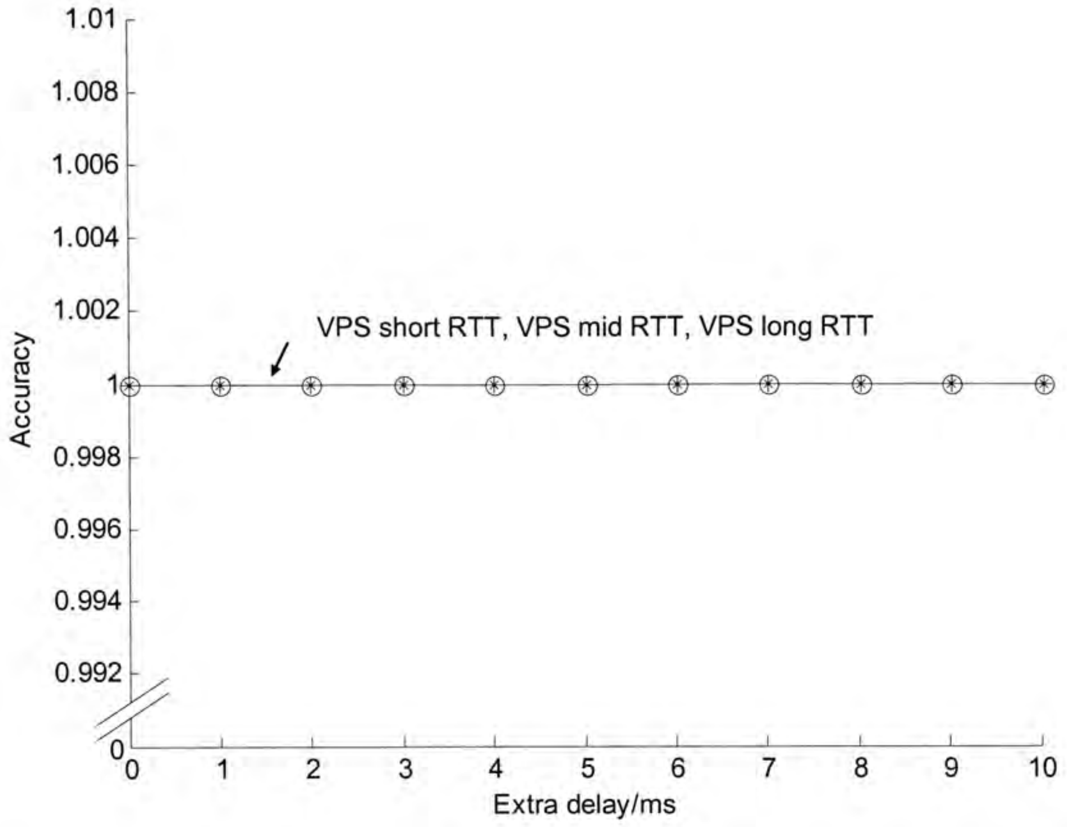


Fig. 42. Accuracy against extra delay factor with bottleneck bandwidth of 10Mbps.

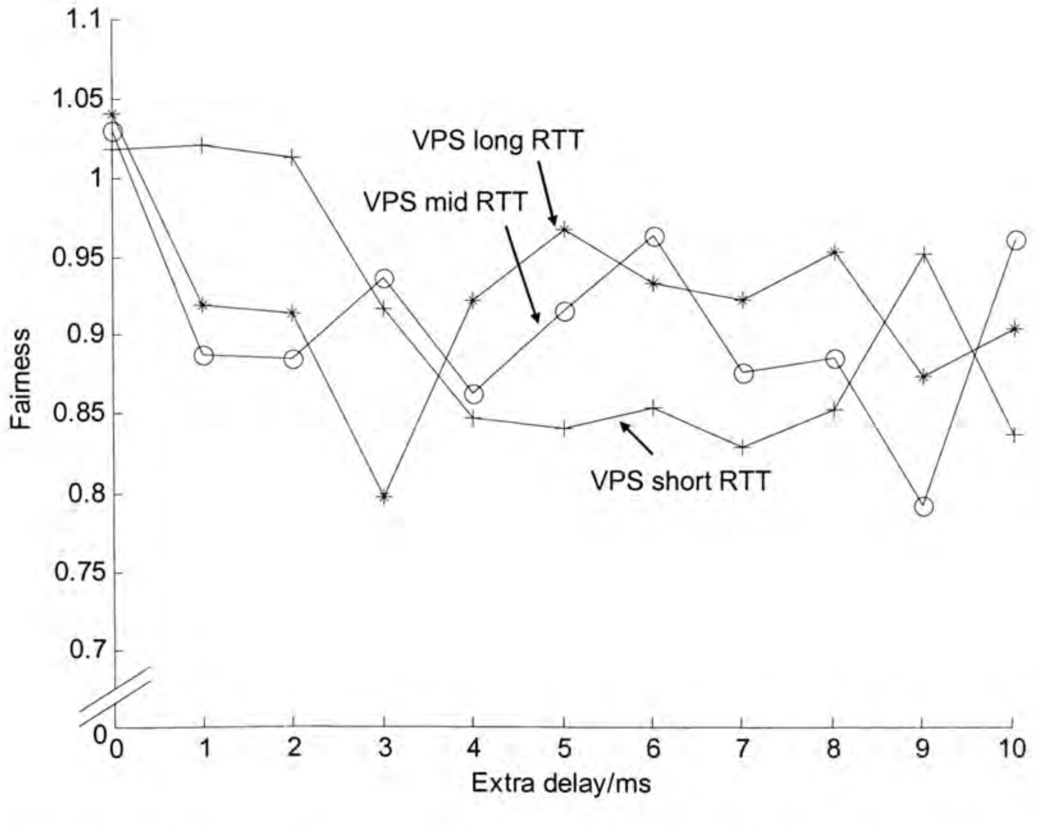


Fig. 43. Fairness against extra delay factor with bottleneck bandwidth of 1Mbps.

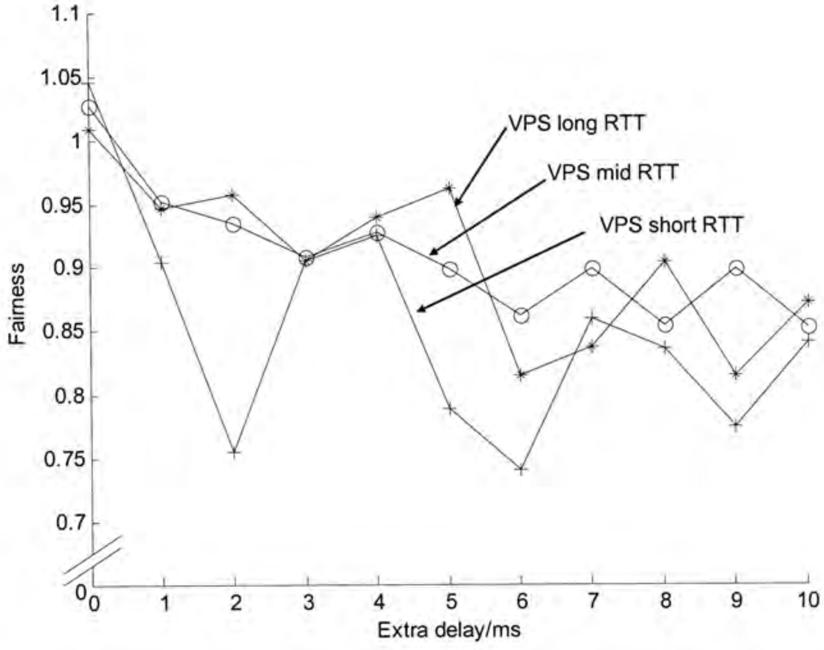


Fig. 44. Fairness against extra delay factor with bottleneck bandwidth of 5Mbps.

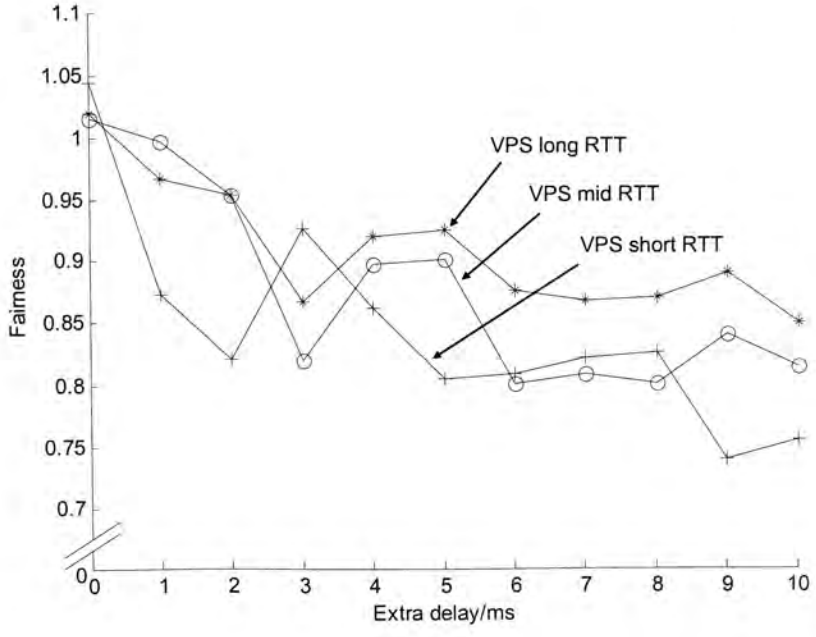


Fig. 45. Fairness against extra delay factor with bottleneck bandwidth of 10Mbps.

Chapter 7

CONCLUSIONS AND FUTURE WORK

This work presented a Virtual Packet Substitution (VPS) algorithm for the allocation of bandwidth among TCP flows originating from the same sender passing through the same network bottleneck to multiple receivers. As the VPS algorithm assigns transmission quota strictly according to the specified bandwidth ratios, it can achieve perfect bandwidth allocation accuracy over time scales as short as one second. Moreover, VPS can maintain excellent fairness with competing TCP flows by computing the transmission quota from virtual flows running the standard TCP Reno congestion control algorithm. The capability to allocate non-uniform bandwidth between TCP flows opens many new possibilities for network services. For example, a service operator may use bandwidth differentiation to provide different quality of service to users of different subscription levels (i.e. more bandwidth for premium subscribers). A media server may dynamically adjust the bandwidth ratios to react to quality feedbacks from clients, and so on. This work merely introduces a new tool and more works are needed to explore the applications and optimization of the newfound tool to various network applications and services.

BIBLIOGRAPHY

[1] H. Balakrishnan, H. S. Rahul and S. Seshan, "An Integrated Congestion Management Architecture for Internet Hosts," *ACM SIGCOMM Computer Communication Review*, Vol.29, Issue 4, pp.175-187, Oct. 1999.

[2] V. N. Padmanabhan, "Coordinating Congestion Management and Bandwidth Sharing for Heterogeneous Data Streams," *Proc. NOSSDAV 99.*

[3] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, M. Stemm and R.H. Katz, "TCP Behavior of a Busy Internet Server: Analysis and Improvements," *Proc. IEEE INFOCOM*, pp.252-262, Mar. 1998.

[4] W. R. Stevens, *TCP/IP Illustrated, Vol. 1*, Addison-Wesley, New York, 1994.

[5] V. Jacobson, "Congestion Avoidance and Control," *Proc. of ACM SIGCOMM*, 1988.

[6] W. R. Stevens, *UNIX Network Programming*, Prentice-Hall, Inc., Upper Saddle River, NJ, 1990.

[7] P. Mehra, A. Zakhor and C. Vleeschouwer, "Receiver-driven Bandwidth Sharing for TCP," *Proc. IEEE INFOCOM*, pp. 1145-1155, Mar. 2003.

[8] J. Crowcroft and P. Oechslin, "Differentiated End-to-End Internet Services Using a Weighted Proportional Fair Sharing TCP," *Computer Communication Review*, Vol.28(3), pp.53-67, Jul. 1998.

[9] D. M. Chiu and R. Jain, "Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks," *Computer Networks and ISDN Systems*, Vol.17, pp.1-14, 1989.

[10] S. McCanne, S. Floyd, "ns-2 Network Simulator" – http://www.isi.edu/nsnam/ns/.

[11] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, "TCP selective acknowledgement and options," *RFC2018*, IETF, Oct. 1996.

[12] V. Jacobson, "Modified TCP Congestion Avoidance Algorithm," Technical report, 30 Apr. 1990. ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt.

[13] OS Platform Statistics, http://www.w3schools.com/browsers/browsers_stats.asp.

[14] SACK Support for Various Operating Systems, http://www.psc.edu/networking/projects/tcptune/.

[15] SACK Support for Windows .NET, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcetcpip/html/

cmcontcpselectiveacknowledgment.asp.

[16] SACK Support for Linux 2.4 and after, http://www.linuxpakistan.net/man.php?query=tcp&apropos=0&section=0&type=2.

[17] D. Katabi, I. Bazzi and X. Yang, "A Passive Approach for Detecting Shared Bottlenecks," *Proc. IEEE Computer Communications and Networks*, pp.174-181, Oct. 2001.

[18] O. Younis and S. Fahmy, "On Efficient On-line Grouping of Flows with Shared Bottlenecks at Loaded Servers," *Proc. IEEE Int. Conf. Network Protocols*, pp.175-184, Nov. 2002

[19] M. Allman, "TCP Congestion Control with Appropriate Byte Counting (ABC)," *RFC3465*, IETF, Feb. 2003.

[20] V. Jacobson, R. Braden, D. Borman, "TCP Extensions for High Performance," *RFC1323*, IETF, May. 1992.

[21] Information Sciences Institute, University of Southern California, "Transmission Control Protocol," *RFC793*, IETF, Sep. 1981.

[22] Lili Wang, James N. Griffioen, Kenneth L. Calvert, Sherlia Shi, "Passive Inference of Path Correlation," *ACM NOSSDAV'04*, June. 2004.

[23] Ningning Hu, Li (Erran) Li, Zhuoqing Morley Mao, Peter Steenkiste, Jia Wang, "Locating Internet Bottlenecks: Algorithms, Measurements, and Implications," *ACM SIGCOMM'04*, Aug. 2004.

[24] S. Floyd, "HighSpeed TCP for Large Congestion Windows," *RFC3649*, Dec. 2003.

[25] Venkata N. Padmanabhan and Randy H. Katz, "TCP Fast Start: A Technique for Speeding Up Web Transfers," *Proc. IEEE Globecom '98 Internet Mini-Conference*, Nov. 1998.

[26] Handley, M., Floyd, S., Pahdye, J., and Widmer, J., "TCP Friendly Rate Control (TFRC): Protocol Specification," *RFC3448*, Jun. 2006.

[27] V. Paxson, "End-to-end routing behavior in the internet," in *Proc. ACM SIGCOMM'96*, pp. 25-39, Oct. 1996.

[28] Jon C. R. Bennett, Craig Partridge and Nicholas Shectman, "Packer Reordering is Not Pathological Network Behavior," *IEEE/ACM Transactions on Networking*, Vol. 7, No. 6, pp.789, Dec. 1999.

[29] E. Blanton, M. Allman, K. Fall and L. Wang, "A Conservative Selective Acknowledgement (SACK)-based Loss Recovery Algorithm for TCP," *RFC3517*, Apr. 2003.

[30] J. Touch, "TCP Control Block Interdependence," *RFC2140*, Apr. 1997.

[31] M. Allman, V. Paxson and W. Stevens, "TCP Congestion Control," *RFC2581*, Apr. 1999.