A Cross-Layer Design for Large Data Transfers

A Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science

Computer Engineering

by

Fatma Alali

May 2016

Approvals

This thesis is submitted in partial fulfillment of the requirements for the degree of

Master of Science

Computer Science

Fatma Alali

Approved:

Malathi Veeraraghavan (Advisor)

Joanne Dugan (Chair)

Kong-Cheng Wong

Accepted by the School of Engineering and Applied Science:

Craig H. Benson (Dean)

May 2016

Abstract

Software Defined Network (SDN) technologies have enabled the introduction of new services such as dynamic Layer-1 (L1) circuits and Layer-2 (L2) virtual circuits (VCs). Our objective is to fully realize high-rate circuits/VCs with almost zero packet loss for large dataset transfers. A multi-domain SDN dynamic Layer-2 (L2) VC was used to conduct experiments to achieve this objective. The results showed that a combination of (i) Circuit TCP (CTCP), in which the sending rate is held fixed, and (ii) Token Bucket Filter (TBF) rate shaper at the sending host, is best to achieve, high-throughput transfers. However, packet losses were still observed on these L2 VCs. Therefore, a new study was presented using more controlled environments.

A new experimental study was undertaken in a single-rack testbed in which packet losses and delays could be deliberately controlled. Three cases were emulated: (i) single circuit/rate-guaranteed VC for a single large transfer from a server, (ii) multiple simultaneous large transfers from a server, and (iii) semi-rate-guaranteed VC. CTCP and the TBF queueing discipline of the Linux traffic control (`tc`) utility are recommended for the first case, and parameter selection methods are provided. For the second case, the `tc` Hierarchical Token Bucket (HTB) discipline is required to support multiple transfers, each on a distinct VC with its own rate and Round-Trip Time (RTT). Our experiments showed that dynamic additions and deletions of classes are possible without impact on ongoing large-transfer flows. For the third case, CTCP is recommended if the throughput of the large transfer is of primary concern, while HTCP is recommended if higher consideration should be given to the other flows.

The thesis also provides insight and lessons learned about the to Linux TCP/IP stack. Three layers were considered: (i) Application layer, (ii) Transport layer, and (iii) Data-link layer. New findings, which are beneficial to networking researchers are presented. We also provide insights into how to monitor flows using tools such as `tcpdump`, `tshark`, and `tcptrace`.

Acknowledgments

I would like to take this opportunity to thank many people without whom this thesis would have not been possible. I would like to thank my advisor Professor Malathi Veeraraghavan for her support and guidance. She taught me not only the knowledge of this research area, but also the way to conduct research and tackle problems which I believe would be beneficial throughout my career. I would like to thank my committee members, Professor Joanne Dugan and Professor Kong-Cheng Wong for taking the time to review my thesis and provide great suggestions, and my colleagues for their support and kind words.

I would also like to thank my parents, my siblings and my friends whom encouraged and supported me throughout my graduate years. I am most thankful to my mother who never stopped praying for me to overcome all my problems and to have a successful life.

This work was supported by NSF CNS-1116081, OCI-1127340, ACI-1340910, CNS-1405171, and CNS-1531065, and U.S. DOE grant DE-SC0011358.

Acronyms

ACK Acknowledgment

AL2S Advanced Layer 2 Service

ATLAS A Toroidal LHC ApparatuS

BDP Bandwidth-Delay Product

BNL Brookhaven National Lab

CAC Connection Admission Control

CAPEX Capital expenses

CRC Cyclic Redundancy Check

CTCP Circuit TCP

cwnd Congestion window

CWR Congestion Window Reduced

DTN Data Transfer Node

DYNES Dynamic Network System

fcwnd CTCP fixed congestion window

FDM Frequency-Division Multiplexing

FDT Fast Data Transfer

GENI Global Environment for Network Innovations

GRO Generic-Receive Offload

GSO Generic- Segmentation Offload

HRT High-Resolution Timer

HTB Hierarchal-Token Bucket

IDC Inter-Domain Controller

L2 Layer-2

LHC Large Hadron Collider

LRO Large-Receive Offload

MARIA Mid-Atlantic Research Infrastructure Alliance

MPLS MultiProtocol Label Switching

MPTCP MultiPath TCP

NIC Network Interface Card

OCS Optical Circuit Switch

OESS Open Exchange Software Suite

OFDP OpenFlow Discovery Protocol

OPEX Operating expenses

OSCARS On-Demand Secure Circuits and Advance Reservation System

PQ Priority Queueing

pS perfSONAR

QoS Quality of Service

RDMA Remote Direct Memory Access

REN Research and Education Network

RoCE RDMA over Converged Ethernet

RTT Round Tripe Time

rwnd Receive Window

SDN Software Defined Network

SNMP Simple Network Management Protocol

TBF Token-Bucket Filter

tc Traffic control

TDM Time-Division Multiplexing

TSO TCP-Segmentation Offload

VC Virtual Circuit

VLAN Virtual LAN

WDM Wavelength-Division Multiplexing

WFQ Weighted Fair Queueing

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Research and Education Network (REN) providers have recently started offering high-speed dynamic Layer-2 (L2) networking services, in addition to basic IP-routed service, to universities and research laboratories. To support dynamic L2 path services, switches should be configurable. Control-plane protocols are required to create new path-based entries in the switch forwarding tables. OpenFlow/Software Defined Network (SDN) technologies have simplified the support of such dynamic rate-guaranteed L2-path services, also known as virtual circuit (VC) services, as the control-plane software can be implemented in SDN controllers external to the switches. Internet2 and ESnet, two US backbone REN providers, have deployed SDN controllers [2, 3] and Layer-2 switches to support dynamic L2 path service.

1.1 Objective

The *objective* of this work is to develop solutions that enable the effective use of dynamic L2 path services for large dataset transfers. Specifically, these solutions address the questions of what transport layer protocol to use, and how to configure link-layer mechanisms within the end hosts that are engaged in the data transfer. These solutions should enable the data-transfer application to fully leverage a high-speed rate-guaranteed L2 path.

1.2 Motivation

Many scientific researchers rely on networks to access national super-computing facilities to run applications, or to download scientific datasets from external storage sites. For example, consider the scientific discipline of high-energy physicists. Experiments are conducted on the Large Hadron Collider (LHC), located near Geneva, Switzerland. The A Toroidal LHC ApparatuS (ATLAS) experiment is one of seven particle-detector experiments constructed on the LHC, and datasets created in this experiment are transferred to large storage systems located at Brookhaven National Lab (BNL) in New York. BNL acts as the Tier-1 site in the US for the ATLAS experiment, which means physicists located in the US will access the BNL high-performance computers to execute their applications on the ATLAS data, and/or download the original or analyzed datasets from the BNL storage systems over the WAN to local computing systems in their own institutions. A 100-MB dataset is generated each second, which adds up to about 1 PB each year.

There are *two* problems with moving large datasets over the current Internet. *First*, there are issues associated with throughput of the large data-transfers flows themselves, and *second*, the large data-transfer flows can have adverse effects on other flows.

We first describe the issues associated with throughput when large datasets are transferred over the Internet. Currently, the global service offered on the Internet is best-effort, which means that bandwidth resources are not reserved prior to data transfer. Without resource reservation, competing (bursty) flows can cause the occupancy levels of router buffers to increase suddenly, which could then cause packet drops. The IP layer does not offer reliable service; instead it is a protocol that simply delivers packets from source to destination. The protocol layer that makes the end-to-end packet delivery reliable is TCP. The TCP sender detects and recovers from packet losses using retransmissions. This basic error control scheme is augmented with a congestion control scheme in which TCP sender lowers its sending rate every time a packet loss is detected on the assumption that the packet loss occurred because of router buffer overflows. Prior to 1988 when a congestion-control was added to TCP [4], there were repeated congestion collapses of the NSFnet (which was the precursor to today's Internet).

The consequence of the TCP sender decreasing its rate when packet loss is detected is that the overall throughput drops especially on high Bandwidth-Delay Product (BDP) paths. The sending rate is increased as acknowledgments arrive at the sender indicating successful packet delivery, and therefore the higher the delay between the sender and the receiver, the slower the recovery. Dart et al. [5] demonstrated that a low packet loss rate as low as 0.005% can cause a significant drop in data-transfer throughput on high-BDP paths. The occurrence of packet losses on IP-routed paths, and consequent lowering and increasing of TCP sending rate, results in variable throughput, and correspondingly variable transfer completion times. Co-scheduling algorithms (e.g., joint scheduling of compute and network resources, or visualization displays and network resources), or workflow management systems [6], need predictable transfer times. Networking solutions that can guarantee a certain throughput for a transfer can offer predictable transfer times.

The *second* problem noted above was that large data-transfer flows can have adverse effects on other flows. As link rates increase, the range between the fastest and slowest flows increases. In other words, when link rates were 64 kbps, and each phone call needed 64 kbps, there was no difference in the rates required by different types of flows. Today, with 100 Gbps links, it is possible to engineer a single data transfer to occur at rates close to 100 Gbps, while there are still co-existing audio flows that require 64 kbps, or even lower rates with compression. Consequently, a high-rate data-transfer flow could have adverse effects on real-time audio/video flows that have stringent delay requirements. Prior work [7] reported experimental results that a single high-rate large-sized flow (referred to as α flow) can cause increased packet delays and/or packet losses in other flows, along with measurements obtained from ESnet that showed spikes in network traffic attributed to a single α flow on a 10 Gbps link.

These above-described problems with moving large datasets across IP-routed networks motivated us to look for alternative networking solutions. Dynamic L2 path service offered on SDNs is the alternative networking solution considered in this work. Unlike in IP-routed service where resources are not reserved for a flow, in dynamic L2 path service, packet losses are prevented by reserving bandwidth resources during circuit/virtual-circuit (VC) setup prior to data transfer. If there are no packet losses due to router/switch buffer overflows, high throughput, close to the rate of the provisioned circuit/VC rate, can be achieved. Furthermore, transfer time becomes more predictable, which resolves the co-scheduling and workflow management systems issue. With circuits/VCs, α flows can be isolated to prevent adverse effects on other flows. Flow isolation is done by mechanisms such as policing flows and scheduling packets, and directing packets of α flows to separate switch/router buffers.

While circuit/VC networks offer the above-stated advantages, there are associated disadvantages. The main disadvantage of circuit/VC service is that it not as scalable as connectionless IP-routed service because of the per-flow policing and scheduling (QoS-control) actions required in switches. A typical switch can perform these QoS-control actions on tens of flows, while a backbone-network link typically carries tens of thousands of flows simultaneously. A second disadvantage is the circuit/VC setup delay. On WANs, this delay could be on the order of 100ms, which means on a high-rate circuit, a file has to be large enough to make the circuit-setup delay overhead a small part of the total transfer time.

To address these disadvantages, we propose the use of rate-guaranteed dynamic L2 path service only to move datasets that are large enough that the transmission delay (size divide by circuit/VC rate) is much larger than the propagation delay. Setting a high threshold for the dataset size also ensures that relatively few flows require circuit/VC service, which addresses the scalability issue.

1.3 Contributions

The main *contribution* of this thesis is a novel cross-layer design that leverages transport- and link-layer protocols to enable the full use of high-rate circuits/VCs to achieve high-throughput transfers. In evaluating this design, we answer several questions: (i) how to set values of parameters of the transport- and link-layer protocols, (ii) how to support the dynamic addition/deletion of circuits/VCs for servers that support multiple simultaneous α flows, and (iii) how to choose a transport-layer protocol for semi-rate guaranteed VCs. Also, the thesis provides insights and lessons learned about the Linux networking stack and presents new findings that could be beneficial to networking researchers. This work was reported in two publication [8, 9].

1.4 Thesis organization

Chapter 2 provides background information, and reviews related work. Chapter 3 describes our proposed cross-layer design and experimental evaluation. Chapter 4 describes a series of experiments designed to obtain an in-depth understanding of the Linux network stack, offers insights and describes lessons learned. The thesis is concluded in Chapter 5.

Chapter 2

Background and Related Work

Section 2.1 provides relevant background material, and Section 2.2 reviews related work.

2.1 Background

Background information on path-based networking, Software-Defined Networking (SDN), Research and Education Networks (RENs), and a specific multi-domain SDN that was used in our work, are described in Sections 2.1.1, 2.1.2, 2.1.3 and 2.1.4, respectively.

2.1.1 Path-based Networking

Before describing path-based networking, we provide a overview of different types of switches, and corresponding networks.

Table 2.1: Types of switches

Admission control(control-plane) \ Multiplexing (data-plane)	Circuit-multiplexing	Packet-multiplexing
Connectionless	N/A	Packet switch, e.g., IP routers, Ethernet switches
Connection-oriented	Circuit switch, e.g., SONET, OCS switches	Virtual-circuit (VC) switch, e.g., MultiProtocol Label Switching (MPLS), VLAN

Table 2.1 classifies switches into three types based on the multiplexing technique used on the data-plane, and the presence or absence of admission control on the control-plane. There are two types of multiplexing: (i) circuit multiplexing (position-based, where "position" means time-slot, frequency, and space), and (ii) packet multiplexing (header-based). *Circuit-multiplexing* techniques include: Time-Division Multiplexing (TDM), Frequency-Division Multiplexing (FDM), and Wavelength-Division Multiplexing (WDM). In TDM, multiple signals share a communication medium by sending data in their assigned timeslots, while in FDM/WDM (the term WDM is used in optical networks), multiple signals share a communication medium by sending data on different frequencies/wavelengths. In *packet-multiplexing* mode, a link is shared between packets of multiple flows, with the packet header carrying information that identifies the flow to which the packet belongs.

If the switch controller runs an admission-control procedure, the switch is *connection-oriented*, while there is no admission-control software in a *connectionless* switch. Since a position (timeslot, frequency and space unit) needs to be assigned to a flow on both the input link and output link of a circuit switch before the switch receives user data, circuit switches are necessarily connection-oriented. As Table 2.1 shows SONET and Optical Circuit Switches (OCS) are examples of circuit switches. If a packet switch controller includes admission-control software, it is a connection-oriented packet switch, also called a virtual-circuit (VC) switch. Examples of VC switches shown in Table 2.1 include MultiProtocol Label Switching (MPLS) and Virtual LAN (VLAN) switches. A connectionless packet switch controller does not run admission-control software. Packet switches operated in this mode are also referred to as datagram routers. Examples of connectionless packet switches shown in Table 2.1 are IP routers and Ethernet switches.

All switch controllers require a routing table that maps destination addresses to output ports. These tables can be computed on servers external to the switches and downloaded to the switch controllers, or these tables can be computed by routing software running within the switch controllers. The latter solution usually coexists with a distributed routing protocol for message exchanges between switch controllers to learn topology, reachability and optionally loading conditions.

In connectionless packet-switched networks, the routing table is sufficient for packet forwarding. In high-speed routers, the routing table (also called Routing Information Base, or RIB) stored in the switch controller (which is a processor with memory) is copied to a forwarding table (also called Forwarding Information Base or FIB) in each line (interface) card. When packets arrive on interfaces, the FIB is consulted to determine the outgoing interface on which to forward the packet based on the destination address carried in the packet header.

Connection-oriented switches also have a controller that store a routing table similar to the one stored in connectionless switch controllers. The difference is that this routing table is used only in the circuit/VC setup phase. The controller also runs signaling protocol software to handle requests for circuit/VC setup and release. Steps to set up a circuit/VC from node A to node B in a distributed solution are summarized as follows: (i) node A sends a request message to the first switch on the path. For circuits and VCs, the setup message contains the destination node B. In addition, a circuit-setup message specifies the required circuit rate, while a VC-setup message specifies a traffic descriptor (e.g., peak rate, average rate, and mean burst size) and QoS information (e.g, packet loss rate, delay, delay variance). (ii) The switch controller parses the setup request, and looks up the routing table for the next hop to reach the destination. (iii) The controller checks its Connection Admission Control (CAC) table to see if the requested bandwidth can be granted (or equivalently, the required QoS measures can be supported for the specific traffic description in VC setups). It selects timeslots and/or frequencies in circuit siwtches, and labels in VC switches. (iv) The switch updates the CAC table, and configures a timeslot/frequency/label mapping with the selected timeslots and/or frequencies in circuit switches, and labels in VC switches, (v) The switch then constructs a setup-request message and sends it to the next hop. The previous steps is repeated at every switch on the end-to-end path until the setup messages reaches the destination, node B. If the setup procedure was successful at each switch on the path, a confirmation message will sent in the reverse direction from node B to node A. If not, the setup request is rejected.

If a circuit was established successfully, user-data arriving in an timeslot/frequency on an input interface to a switch will be forwarded by the switch to the output interface indicated in the timeslot/frequency mapping table on the specified output-side timeslot/frequency. A VC switch performs a similar action after first demultiplexing each packet, extracting the label carried in the packet, and then consulting the label mapping table to determine the output interface and corresponding label to use in the header of the forwarded packet.

Policing and scheduling mechanisms are implemented at switches to support Quality of Service (QoS). Scheduling is performed at egress interfaces to decide which packet to transmit next. Weighted Fair Queueing (WFQ) and Priority Queueing (PQ) are examples of schedulers. In WFQ, multiple virtual queues are created on egress interfaces. Each queue is assigned bandwidth (either strictly restricted for just this queue or shared among all virtual queues) and buffer space. PQ can be combined with WFQ.

Policing is performed at ingress interfaces to ensure that a flow does not exceed its assigned rate (circuit/VC rate). If a new packet arrival causes the flow-rate to exceed the circuit/VC assigned rate, the policier marks the packet as being out-of-profile. Different actions can be configured to handle out-of-profile packets. For example, out-of-profile packets could be dropped, directed to a low-priority queue, or packets could be probabilistically droped.

2.1.2 Software-Defined Networking (SDN)

The circuit setup procedure described in Section 2.1.1 assumed that signaling protocols were implemented in a switch. For decades, the networking community designed distributed protocols for scalability. The SDN concept is changing this paradigm to a centralized approach.

The main idea of SDN is to move most of the software that runs on a switch controller to an external server. This includes routing-protocol software, routing-table precomputation software (e.g., software that runs Dijkstra's algorithm based on received routing-protocol messages), and signaling software (for handling signaling protocol messages for circuit/VC setup and release) in circuit/VC switches. What is left behind on the switch controller is simple software that can receive, parse and act on messages for configuring the forwarding table. In connectionless packet-switches, this forwarding table is as described in Section 2.1.1, mapping destination addresses to output ports. In circuit/VC switches, this forwarding table corresponds to the timeslot/frequency/label mapping table of our description in the Section 2.1.1. One example protocol used for this purpose is OpenFlow, and the forwarding/label mapping table is referred to as an OpenFlow table. The

OpenFlow protocol has been extended to include wavelengths for optical circuit switches (v. 1.4.0).

As a concept, the idea of OpenFlow/SDN is not new. But the reason for the successful deployment of SDN, which has primarily occurred in datacenters, is a lowering of costs, both capital expenses (CAPEX) and operating expensse (OPEX). For example, if a datacenter has 1 million servers (e.g., Google datacenters), then assuming 48-port switches, a total of 20,834 switches is required to connect the servers. The cost of a 48-port switch is high (on the order of $10K$) because of software-engineer salaries. Software run on the switches can typically only be modified by engineers employed by the switch vendor. Datacenter providers, especially Google, recognized that if most of the software could be removed from the switch controller and run externally on the datacenter provider's own servers, this software could be modified and upgraded using external software engineers. Furthermore, the software could be open-source, lowering costs even further.

In addition, such an external deployment of the controller software allows service providers (e.g., datacenter providers or ISPs) to add new features to the software to introduce new services. In the old model, feature requests from service providers are prioritized by the switch vendor, which could add significant delays to the deployment of new services, especially for smaller service providers.

2.1.3 Research and Education Networks (RENs)

University campuses and research laboratories are interconnected via regional RENs and backbone (core) RENs. Internet2 [1] is a US-wide backbone REN, which each region (typically US state) has its own regional REN. For example, the universities in Virginia, including University of Virginia (UVA), connect their campus networks to a network operated by Mid-Atlantic Research Infrastructure Alliance (MARIA). MARIA's network is connected to the Internet2's network via a 100 Gb/s Ethernet link.

Internet2 operates three networks, an IP-routed Layer-3 (L3) network, an Ethernet-switched Layer-2 (L2) network, and optical circuit-switched WDM Layer-1 (L1) network. Networking services are offered by each of these layers. Relevant to our work is the L2 network and the corresponding service, which Internet2 calls Advanced Layer 2 Service (AL2S), and hence this L2 network and the AL2S service is explained in further detail.

Fig. 2.1 shows the Internet2's AL2S network [1], which consists of 39 OpenFlow-enabled Ethernet switches. These switches are operated in VC mode, which means an SDN controller needs to explicitly configure all the AL2S switches on the path before data can be transferred. The specific SDN controller used in the AL2S network is called Open Exchange Software Suite (OESS) [2]. Below, we briefly describe how OESS is used to set up L2-paths (VCs) across a single domain (the word "domain" is used here to describe a network that is owned and operated by a single organization).

OESS learns the network topology by running a protocol called OpenFlow Discovery Protocol (OFDP) [10]. OESS offers a GUI through which administrators and other users can request an L2-path by specifying the endpoints, rate, start time, duration, and VLAN Identifiers (IDs). The OESS runs a path computation algorithm to find a path between the user-specified endpoints using its knowledge of the network topology. The OESS then sends an OpenFlow message to each switch on the path to set up an entry in the OpenFlow table, which is the same as the label mapping table described in Section 2.1.1. The label corresponds to the VLAN ID. A standard called IEEE 802.1Q [11] defines a header that carries a VLAN ID. One such header is added to each Ethernet frame that is sent on the L2 path. Each OpenFlow switch then matches the VLAN ID and incoming port to determine the outgoing port to which the packet should be directed. The switch modifies the VLAN ID in the outgoing packet based on the VLAN ID listed for the output interface in the OpenFlow table. This type of packet forwarding ensures that all packets of a flow take the same path, which explains the name path-based networking.

2.1.4 A Multi-Domain SDN

As part of a parallel effort in our research group, a multi-domain SDN was deployed and configured to offer users inter-domain dynamic L2 path service. This service was used in our work. Hence, the multi-domain SDN deployment is described here briefly to provide readers some background on the L2 paths used in our work.

The multi-domain SDN deployed by our group leveraged small OpenFlow networks called Dy-

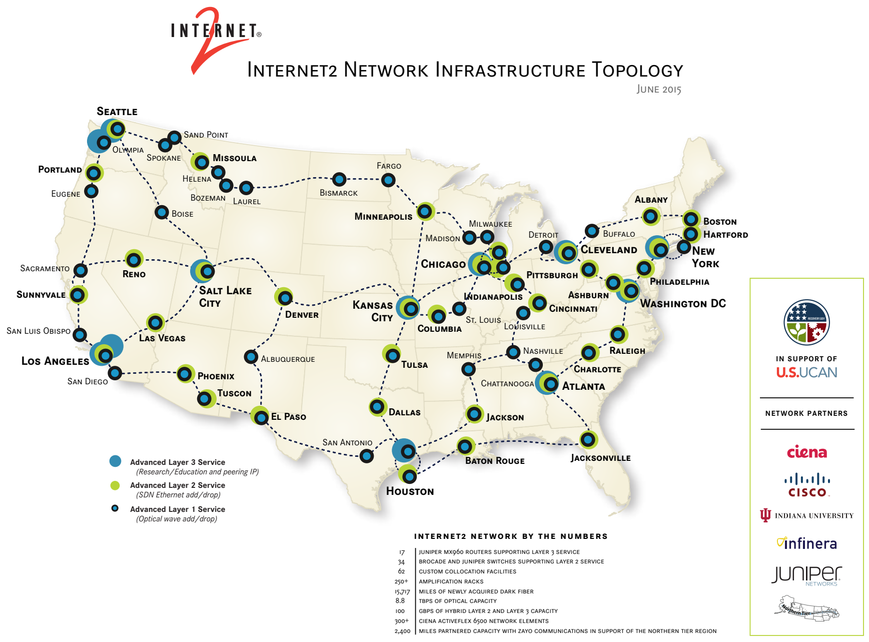


Figure 2.1: Internet AL2S network [1]

namic Network System (DYNES) [12] that were deployed as part of an earlier NSF funded project. The DYNES networks were deployed at 40 universities and 11 regional networks across the US. The DYNES instruments from eight universities were part of the multi-domain SDN deployment. These eight universities were as follows: (i) U. Virginia (UVA), (ii) MAX GigaPoP (MAX), (iii) U. Wisconsin, Madison (UWisc), (iv) University of New Hampshire (UNH), (v) Internet2 Lab (I2Lab), (vi) Rutgers University, (vii) Indiana University (IU), and (viii) Colorado University (CU).

Each university-campus DYNES equipment, as shown in Fig. 2.2, consists of three hosts: Fast Data Transfer (FDT) server, Inter-Domain Controller (IDC) host, perfSONAR (pS) [13] host, and one Ethernet switch (which is OpenFlow enabled in some sites). The FDT server runs data-transfer applications, the IDC host runs SDN controllers, and the pS host runs active-measurement tools for monitoring network performance. Two SDN controllers are executed in the IDC host: OESS (described in the Section 2.1.3), and On-Demand Secure Circuits and Advance Reservation System (OSCARS) [3]. OESS is an intra-domain SDN controller that controls switches via OpenFlow, while OSCARS supports inter-domain service.

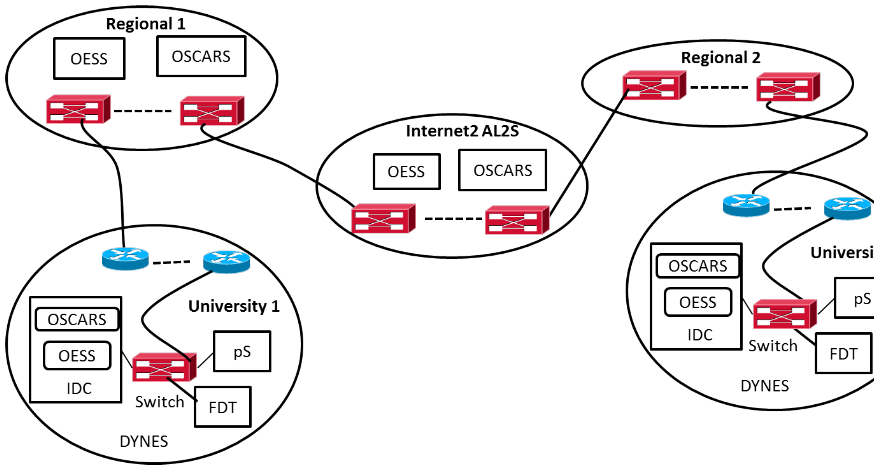


Figure 2.2: An illustration of our multi-domain dynamic L2 path service deployment

Some regional RENs such as Regional 1 in Fig. 2.2 run OSCARS and OESS controllers to offer dynamic L2 path service while others such as Regional 2 in Fig. 2.2 offer only static L2 path service and hence do not deploy OESS and OSCARS. As can be expected with the roll-out of a new networking service, organizations will slowly deploy the service one-at-a-time. Static L2 path service is available from most RENs and university campus networks, and can be used to bridge gaps in the dynamic L2 service offering.

Background information on Internet2 AL2S was provided in Section 2.1.3. Besides OESS, Internet2's AL2S network runs OSCARS as illustrated in Fig. 2.2.

Next, we describe how an inter-domain VC is established dynamically. For the data-plane experiments conducted in our work, we used this procedure to set up inter-domain VCs.

When the OESS receives an inter-domain L2-path provisioning request, it submits the request to OSCARS on behalf of the user. In turn, OSCARS performs path selection with call admission control within its own domain to determine if the required bandwidth resources are available from the specified start-time to the specified end-time. If successful, OSCARS sends a `createReservation` message with endpoints, rate, start time and duration, to the OSCARS in the next domain, which is selected based on the computed path. The procedure is executed in a daisy-chain fashion until the OSCARS of the last domain on the end-to-end path is reached. If successful, `Confirmation` events are sent from one domain's OSCARS server to the next in the reverse direction. If resources are unavailable in the requested timeslots in any domain, path reservation fails. Reverse-direction messages are used to release resources held for the request in upstream domains, and the user request is rejected.

Path provisioning occurs either automatically or upon receiving a `createPath` message from the user just before the reservation start-time. This procedure also uses a daisy-chain of signaling messages between OSCARS servers. When it is time for OSCARS to provision the circuit in the local domain, it contacts the OESS for path provisioning. OESS takes the provisioning request from OSCARS, and provisions the path through its domain using the procedure described in Section 2.1.3.

2.2 Related Work

Solutions to enable fast transfers of large datasets fall into two categories: transport-layer solutions and application-layer solutions. Several enhancements to TCP, specifically its congestion-control algorithm, have been proposed, e.g., FAST-TCP [14], CUBIC [15], and HTCP [16]. Of these, HTCP has been recommended by ESnet [17]. HTCP increases its congestion window more aggressively than Standard TCP; the increase parameter is varied depending upon the elapsed time since the last congestion event. HTCP does not alter the multiplicative decrease algorithm used in Standard TCP. The reason we pursued a solution using rate-guaranteed VCs is that these TCP advances still suffer from the problem of packet losses as the underlying network is connectionless (IP), and also suffer from the problem of having adverse effects on other flows, as described in Chapter 1.

Other advances include new transport protocols that run over UDP have also been proposed for fast transfers: UDT [18], RAPID [19], and RBUDP [20]. UDT is a UDP-based user-space protocol developed for high speed wide-area networks, where reliability and congestion control are implemented in the user space. Another technology, called Remote Direct Memory Access (RDMA) over Converged Ethernet (RoCE), has been studied experimentally to support fast data transfers [21]. The UDP based transport-protocols are not as fast as TCP enhancements because these are typically implemented in user-space. The RoCE solution suffers from the handicap of requiring special Network Interface Cards (NICs). Furthermore, packet losses on the path can degrade performance [21].

Application-layer solutions include GridFTP [22] and Fast-Data Transfer (FDT) [23]. These solutions tune TCP and other networking-stack parameters at senders and receivers, use parallel TCP connections, stripe transfers using multiple servers at each end with parallel file systems [24, 25], and pipelining, which is especially useful if a dataset has many small files [26,27]. Other papers have proposed the use of multi-hop path splitting [28]. The idea of multi-hop path splitting is to split the path between the source and destination into a multiple-hop chain of intermediate nodes. Multiple TCP connections are chained together sequentially on the end-to-end path between source and destination. This solution improves TCP performance for two reasons: (i) the RTTs on each chained connection is shorter than on a direct TCP source-to-destination, which allows for a more rapid detection of, and recovery from, congestion, and (ii) Lost packets can be recovered faster from a closer TCP end-point than the source. Multi-pathing at the transport layer was proposed in MultiPath TCP (MPTCP) [29]. The key idea of multi-pathing is to stripe the data over multiple paths that are used to transfer different chunks of data. Unlike in the GridFTP solution, which can be used to run transfers over multiple parallel TCP connections with the segmentation and reassembly of payload being handled at the application layer, MPTCP performs these functions in the transport layer. Khanna et al. propose combining multi-hop path splitting with multi-pathing [30].

Finally, we describe work that is related to our work in its use of the Linux Traffic-Control `tc` utility. Hanford et al. [31] proposed the use of `tc` with HTCP over best-effort IP networks to control the sending rate. When sending from a Data Transfer Node (DTN) to another host, under-buffered switches, slow firewalls and other slow network devices on the path could cause packet losses, and corresponding drops in throughput. Hanford et al. proposed a tuning daemon that uses the `ss` Linux utility to monitor active sockets at the sender, and suggests `tc` parameter settings to optimize the large data transfer. Our work differs in its usage of `tc` as it targets transfers circuit/VCs, not best-effort IP networks.

Chapter 3

A cross-layer design for large transfers across L2 paths

3.1 Introduction

The problem statement of this work is to determine what protocols/mechanisms to use at the transport layer and link layer within end hosts in order to take full advantage of end-to-end rate-guaranteed L2 paths. Chapter 1 described our motivation for this study. If new OpenFlow/SDN technologies can be leveraged to dynamically setup rate-guaranteed end-to-end L2 paths, then packet losses due to switch/router buffer overflows can be prevented, which in turn allows the transfer to achieve high throughput. Further, by separating out the high-speed large-sized flows into their own rate-guaranteed L2 paths, adverse effects of such α flows (also called *elephant flows*) on other flows, especially delay-sensitive real-time flows, can be avoided.

We started by running experiments across inter-domain L2 paths that were created across the multi-domain SDN that our research group deployed as described in Section 2.1.4 of Chapter 2. The *first* part of this chapter describes how we leveraged a combination of a transport protocol designed for rate-guaranteed circuits/VCs, and a Linux utility that allowed for rate control at the sending host, to achieve high-throughput, low-loss transfers across inter-domain L2 paths. While these results were encouraging, we found this production environment' to have too many unconstrained variables. We use the term "production environment" because the L2 paths (used for our experimental traffic) were provisioned on links that also carried real-user traffic. For example, the real WAN traffic generated by UVA users shared the access links on which the VLANs required for our experimental L2 paths were provisioned.

To gain more control of our experimental environment, we switched to using the Global Environment for Network Innovations (GENI) [32] testbed, which is an NSF-funded federation of many racks that are owned-and-operated by various universities. The GENI rack providers developed and deployed scheduling and authentication software to allow thousands of networking researchers to login into any rack (there are over 100 racks between multiple GENI rack projects: ProtoGENI, InstaGENI, ExoGENI, OpenGENI, and CiscoGENI), and run experiments. The scheduling software has a GUI which allows a user to select multiple VMs or bare-metal (non-virtualized) hosts at one or more racks, and if the racks are located in different universities, then the scheduling software signals the Internet2 OESS and OSCARS to dynamically provision rate-guaranteed L2 paths and/or multipoint topologies to interconnect the user's selected VMs/hosts.

The *second* part of this chapter offers a more systematic study of the interactions between path characteristics, transport-layer and link-layer protocols/mechanisms. The experiments were carried out within single racks to avoid interference from other traffic, which would occur if the experiments involved multiple racks, because GENI uses multiple campus networks, regional networks and Internet2, as was the case with our multi-domain SDN. Linux tools such as `netem` were used to inject artificial delays or packet losses at receivers to emulate WAN paths. The result of this work is a novel *cross-layer design* for support high-speed large data transfers across rate-guaranteed circuits/VCs. The work presented in this chapter was published in an IEEE conference [9] and ACM conference [8].

Section 3.2 provides background information on transport-layer and link-layer protocols. Section 3.3 presents the results of our experiments on inter-domain L2 paths provisioned across our multi-domain SDN. Section 3.4 describes our proposed cross-layer design. Section 3.5 describes our methodologgy for selecting parameters in different implementations of our cross-layer design. The chapter is concluded in Section 4.6.

3.2 Background

3.2.1 Transport-layer protocols

In prior work [33], our research group developed a transport-layer protocol designed specifically for use on circuits/VCs. The transport-layer protocol is named *Circuit-TCP (CTCP)* because it is simply TCP without congestion control. The purpose of TCP's congestion control mechanism is to determine the available capacity on the path and to dynamically adjust the sending rate to that capacity. Since the circuit/VC is configured with a specific rate, TCP's congestion control functionality is no longer needed. In CTCP, the congestion window is held fixed at a value specified by a parameter `fcwnd`.

While packet losses will not occur due to switch buffer overflows in circuits/rate-guaranteed VCs, TCP's *flow control* is still needed as the receive buffer can overflow in multi-tasking receivers. The window-based flow control mechanism of TCP is hence retained in CTCP.

Finally, as bit errors can still occur, TCP's *error-control* mechanism is necessary. The CTCP sender maintains retransmission timers, and performs retransmissions as in standard TCP.

For semi-rate-guaranteed VCs, in addition to CTCP, other high-speed TCP implementations were considered. Recall our brief description of HTCP in Section 2.2. HTCP has been identified as a winner of sorts by ESnet for high-speed transfers on large bandwidth-product paths, and therefore in our experiments, besides CTCP, we also consider HTCP.

3.2.2 Link-layer mechanisms for sending rate control

Ethernet Network Interface Cards (NICs) are now available at 10 Gbps (10 GigE), and 40 GigE. Therefore the rate at which packets will be transmitted by a sending-host NIC could be as high as the access-link and backbone-link rates of service providers. With RAID disk-arrays, parallel file-systems, and high-speed transfer applications such as GridFTP, it is feasible for a host to send data at high rates.

However, in path-based networking, a user/application should request a circuit/VC of a certain rate, and then send data at that specified rate after the circuit/VC is set up. Since the circuit/VC rate could be lower than the sending-host NIC rate, some mechanism is required for rate control at the sending host.

There is an existing utility in Linux to perform exactly this function of rate control. It is is called traffic control (`tc`). This utility supports multiple queueing disciplines. Of these, our experiments involved three disciplines: Token-Bucket Filter (TBF) queuing discipline, Hierarchal-Token Bucket (HTB), and Byte limited First In, First Out queueing discipline (bfifo).

TBF is a simple queuing discipline that is used to shape traffic to a specific rate. TBF has three basic parameters: `rate`, `limit`, and `burst`. The `rate` parameter represents the rate at which tokens are generated and held in a token bucket. The `limit` parameter represents the size of a buffer that holds packets while waiting to be transmitted by the NIC. The `burst` parameter represents the token bucket size, which determines the maximum number of packets that can be sent out back-to-back at the NIC rate.

HTB is a classful queuing discipline in which multiple queues can be defined. For each class, two basic parameters, `rate` and `ceil`, can be specified, where `rate` represents the maximum rate that is guaranteed to the class and its children (the "hierarchical" part of the queueing discipline implies that each class can have sub-classes that are referred to as children), and `ceil` represents the maximum rate at which flows directed to the class can send data based on its own allocation and the allocation of its parent class, if present. There are other parameters related to the token bucket aspect of HTB. Filters can be set in the `tc` command to specify the class to which packets of specific flows should be directed. A default class for all unspecified flows can also be configured. Bandwidth borrowing across siblings of a parent class is allowed. To prevent bandwidth borrowing, all classes should be specified at the same level as root classes where the HTB qdisc is the parent.

We attached leaf classes to top-level HTB classes as there was no parameter equivalent to the TBF `limit` parameter in HTB. We used *bfifo* for these leaf classes. It is a simple first-in first-out queuing discipline, which has one main parameter, `limit`, which represents the size of a buffer where packets are held while awaiting transmission by the NIC.

We used the HTB discipline when the sending host supports multiple simultaneous α flows, each assigned to a different circuit/VC. Since the rate and RTT of different circuits could vary, the

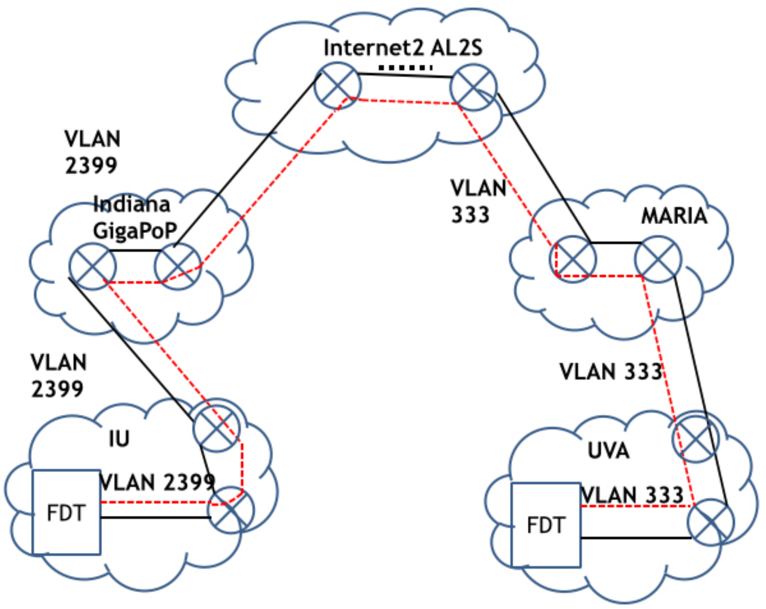


Figure 3.1: End-to-end L2 path between University of Virginia (UVA) and Indiana University (IU)

rate parameter of each class, and the buffer size associated with the lower-level `bfifo` disciple, were set accordingly.

3.3 High-speed transfers across L2 paths provisioned across multiple domains

3.3.1 Experiment setup and execution

The multi-domain SDN described in Section 2.1.4 was used for this set of experiments. Specifically, an L2 path was provisioned from the FDT host at the University of Virginia (UVA) to the FDT host at Indiana University (IU). Recall from Section 2.1.4 that both UVA and IU had existing DYNES networks. Each DYNES FDT host has two Intel Xeon E5620 four-core processors (for a total of eight cores) and 24 GiB RAM.

Fig. 3.1 shows that the L2 path passes through five domains: UVA, MARIA (which is UVA's regional REN), Internet2 AL2S, Indiana GigaPop (which is IU's regional REN), and IU. VLAN ID

333 was configured in the UVA FDT, across the UVA DYNES switch, other UVA campus switches, through the regional REN MARIA switches to the link connected to a port on the Internet2 AL2S switch at Ashburn, VA. The path continues through Internet2 AL2S network, and exits this network through a port to IU's regional REN, Indiana GigaPoP. VLAN ID 2399 was used for the segment of the L2 path between the Internet2 AL2S switch in Chicago and the IU FDT. Some switches on the end-to-end L2 path were configured dynamically through controllers, e.g., through UVA DYNES switch, Internet2 AL2S switches, and IU DYNES switch. The segments in between these switches were provisioned statically using other means.

The application `iperf3` was used to execute memory-to-memory transfers on the end-to-end L2 path. This application reports throughput, number of retransmissions, and total amount of data sent. Shell scripts were used to execute multiple `iperf3` runs, each of duration 20 sec, with gaps between runs (to ensure some change in the background network traffic), and to collect `tc` statistics. The collected `iperf3` logs were parsed to extract throughput and packet retransmission rates, and the `tc` logs were parsed to ensure that no packets were dropped at the sending host because of rate limiting.

We experimented with two versions of TCP: CTCP and HTCP. Many TCP parameters need to be set for high-speed transfers [17]. For both HTCP and CTCP, the numbers used were as follows: the minimum, default and maximum values for the parameter `net.ipv4.tcp_rmem` were all set to 32 MiB (using 1 Mebibyte or MiB = 2^{20} bytes), the minimum, default and maximum values of parameter `net.ipv4.tcp_wmem` were set to 4 KiB, 10 MiB, 32 MiB, respectively, and auto-tuning and window scaling were enabled. The `net.ipv4.tcp_rmem` parameter determines the receive-side buffer size. It is set to at least twice Bandwidth-Delay Product (BDP) because TCP code halves the default value of `net.ipv4.tcp_rmem` for the initial receive-side window.

Besides the above-listed parameters that are common to both HTCP and CTCP, CTCP has an additional parameters, `fcwnd`, which is the fixed congestion window. Since the Round-Trip Time (RTT) on the UVA-to-IU end-to-end path was 26 ms and the path rate was 4 Gbps, using a scaling factor of 1.2 (to make the congestion window slightly larger than BDP), we set `fcwnd` to 14.8 MiB.

For L2 driver parameters, the following values were

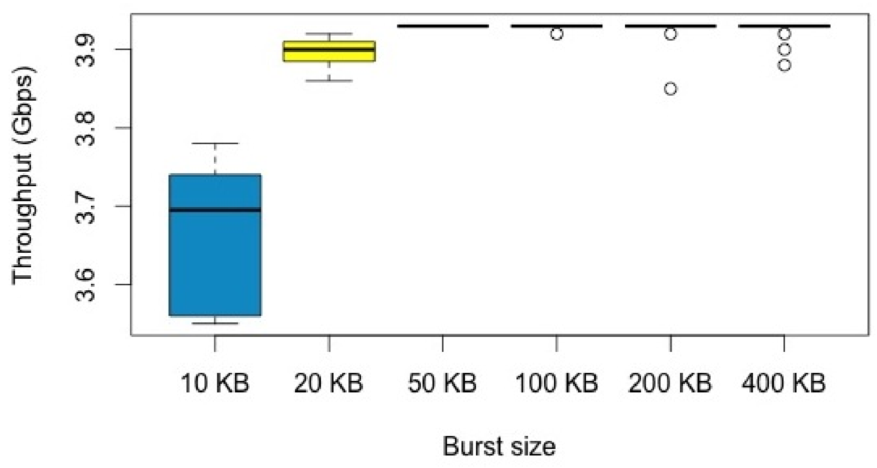


Figure 3.2: Impact of `burst` size on throughput

used: parameter `net.core.netdev_max_backlog` was set to 1500000, and `txqueuelen` was set to 10000. For the Linux traffic-control (`tc`) utility, the Token Bucket Filter (TBF) queueing discipline was selected because the Linux tc manual [34] states that TBF is "'suited for slowing traffic down to a precisely configured rate," and it "scales well to high bandwidth." The TBF rate was set to 4 Gbps, while the `limit` and the `burst` size were varied to determine ideal values as explained below.

3.3.2 Experiment results

First, we describe experiments conducted to understand the sensitivity of throughput and packet loss rates to the values chosen for the TBF `burst` size, TBF `limit`, and CTCP `fcwnd`. Next, we present results from experiments conducted to compare CTCP and HCTP for use on rate-guaranteed paths.

Parameter selection: The first set of experiments studied the impact of the *TBF burst size*. Fig. 3.2 presents boxplots for different TBF burst settings. It shows that for burst-size values of 50 KB or larger, end-to-end `iperf3` throughput is close to the 4 Gbps path rate. If the burst size is small, the token generator will not have free space in the token bucket to deposit additional tokens while waiting for the CTCP layer to pass down packets. Since the `iperf3` application/CTCP does not pass packets down to `tc` in a strictly clocked manner, it is beneficial to have a larger token bucket to hold tokens so that on average packets can be sent out at the L2 path rate.

Table 3.1: Impact of `burst` size on retransmission rate

Burst size (KB)	Median retx rate in first sec %	Median retx rate in later sec %	Mean retx rate in first sec %	Mean retx rate in later sec %
10	0	0	7.9e-4	4.1e-5
20	7.2e-4	0	166e-4	31.7e-5
50	0	0	11.9e-4	8.1e-5
100	9.3e-4	0	13.8e-4	17.9e-5
200	6.3e-4	0	158e-4	55.6e-5
400	0	0	238e-4	46.9e-5

Table 3.1 shows the negative effects of large burst size, i.e., there are more packet losses. We observed packet losses on the end-to-end L2 paths because current campus, regional and core RENs do not support the data-plane functionality needed for rate-guaranteed transfers (i.e., policing and scheduling are not enabled). Table 3.1 separates out packet retransmission rate in the first second from the retransmission rate in later seconds. This is because the token bucket starts in a full state, which causes a full burst of packets to be sent at the NIC rate (10 Gbs). Since the end-to-end L2 path traverses many types of switches, some of them could have small buffers [35] causing packet losses in the first second. This explains why the median packet retransmission rate is higher in the first second than in later seconds as seen in Table 3.1. The mean packet retransmission rates are 1-2 orders of magnitude higher in the first second than in later seconds. The mean values are higher than the median values indicating a right-skewed graph with outliers, which means that in some runs, there was sufficient background traffic to cause switch buffers to overflow more often than in other runs. Finally, we observe that in runs with larger burst sizes (200 KB and 400 KB), the mean packet retransmission rate was higher than in runs with smaller burst sizes. With the 20 KB burst size setting, there was 1 run (out of 100 runs) in which there was a 1.5% packet retransmission rate in the first second, which caused the mean rate to be considerably higher than for the 50 or 100 KB settings. Since our `iperf3` packets were sharing links with real traffic, there can be such anomalies.

What is clear is that at larger burst sizes, there is a higher risk of packet loss, which impacts not only the data-transfer flow itself, but also other flows. Therefore, while burst size values of 200

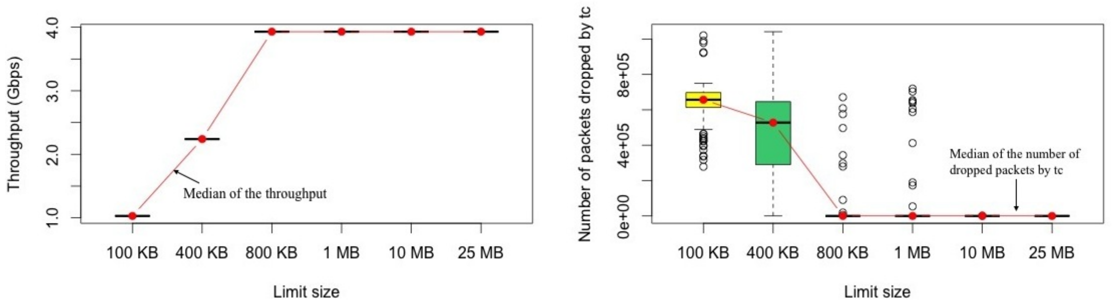


Figure 3.3: Boxplots for different values of the `limit` parameter; the total number of the packets sent was approximately 2.5e6 packets

or 400 KB still allow for the flows to enjoy close to 4 Gbps throughput (as seen in Fig. 3.2), the retransmission rates at these settings are also higher than at the 50 or 100 KB settings. Therefore, we recommend using smaller burst sizes such as 100 KB.

Next, to study the impact of *TBF limit parameter*, all other parameter values were held unchanged at the numbers specified in Section 3.3.1, and TBF burst size was set to 100 KB (based on the results presented above). Fig. 3.3 shows that at smaller values of `limit`, median throughput is lower and the number of packets dropped by **tc** is higher when compared to runs with larger settings for the `limit` parameter. Recall that the `limit` parameter minus the burst size is the size of the buffer used to hold packets in case the application sends packets to `tc` faster than the configured rate. Therefore, at smaller settings of `limit`, there is insufficient space to absorb bursts from the application. Also, there are time gaps during which there are no packets in the `tc` buffer waiting for service, which leads to the lower throughput. Further, Fig. 3.3 shows that there is more variability in the number of packets dropped by `tc` at smaller values of `limit`. Using a size more than 800 KB for `limit` appears to lower the number of packets dropped by `tc` to close to 0 in most runs, but even at the 10 MB setting, there was one run in which **tc** dropped 1556 packets (out of 2.5e6 packets). Therefore, if memory is not a constrained resource, we recommend using a value as large as twice BDP. Finally, the choice of this `limit` parameter depends upon the CPU resources available to the data-transfer application. For example, if the CPU is multi-tasking between the data-transfer application and other applications, data will be passed down to the `tc` layer only when the data-transfer application is scheduled. Therefore data will appear at the `tc` layer in bursts with silence periods in

Table 3.2: Impact of CTCP `fcwnd` parameter

BDP (MiB)	Mean Throughput (Gb/s)	Mean Retx rate in first sec. %	Mean Retx rate in later sec. %
14.8	3.93	17.1e-4	16.8e-5
12	2.97	16.6e-4	2.63e-5
6	1.61	14.5e-4	2.65e-5

between. In this case, a larger `limit` size may be required since in our experiments, `iperf3` was the sole application being executed on the hosts.

Next, an experiment was conducted to study the impact of the *CTCP fcwnd parameter*. The TBF burst size was set purposely to 400 KB so that some losses would occur, which would then allow us to determine whether the CTCP value influenced the packet loss rate. Table 3.2 shows the results. The BDP on the UVA-to-IU path is 12.4 MiB for the 4 Gbps rate setting. Therefore, setting the CTCP `fcwnd` value to a larger value, 14.8 MiB, allowed for the throughput to almost equal the path rate. But with smaller values of `fcwnd`, such as 12 MiB or 6 MiB, the number of outstanding bytes that the CTCP sender can send is less than BDP and hence throughput decreases.

The third and fourth columns of Table 3.2 show that the mean packet retransmission rate in the first second is not influenced by the `fcwnd` value, but the mean packet retransmission rate in later seconds is slightly influenced by this parameter. In the first second, a burst of size 400 KB is sent at NIC speed (10 Gbps) and hence packet loss rate is more influenced by this parameter as we observed earlier rather than by the `fcwnd` parameter.

Comparison of CTCP and HTCP: The following values were used for TBF parameters: `rate` was set to 4 Gbps, `burst` size was 100 KB, and `limit` was set to 25 MiB. The `fcwnd` value for CTCP was set to 14.8 MiB, and the remaining TCP parameters were set to values specified in Section 3.3.1.

Table 3.3 presents our test results. Since the congestion window is set to a small initial value and grows gradually with HTCP, we separated out results for the first two seconds from the results for later seconds. In the first two seconds, since HTCP **cwnd** was small due to Slow Start, throughput

Table 3.3: `iperf3` throughput and retransmission rate

Throughput (Gbps)				
TCP ver	no. of sec.	Min.	Median	Max.
CTCP	first 2 sec.	3.87	3.92	3.93
HTCP	first 2 sec.	0.34	0.69	1.37
CTCP	later sec.	3.93	3.93	3.94
HTCP	later sec.	2.73	3.93	3.94
CTCP	all sec.	3.93	3.93	3.93
HTCP	all sec.	2.57	3.61	3.68
Packet Retransmissions Rate %				
TCP ver	no. of sec.	Median	Mean	90th %
CTCP	first sec.	62e-5	118e-5	28.2e-4
HTCP	first sec.	0	1349e-5	0
CTCP	later sec.	0	5.79e-5	0.83e-4
HTCP	later sec.	1.72e-5	22.5e-5	1.4e-4
CTCP	all sec.	7.12e-5	11.5e-5	2.22e-4
HTCP	all sec.	1.72e-5	24.1e-5	2.59e-4

was low (between 340 Mbps to 1.37 Gbps) as seen in the second row of Table 3.3. On the other hand, with CTCP, since the congestion window was held fixed at the **fcwnd** value, which was 1.2 times BDP, the effective sending rate was 4 Gbps right from the start as seen in the first row of Table 3.3. In later sec, both CTCP and HTCP enjoyed throughput close to the 4 Gbps path rate.

The second part of Table 3.3 shows the measured packet retransmission rates. The first two rows show the packet retransmission rates in the first second, rather than in the first two seconds as was done for throughput, because it is only at the start of the run that the token bucket is full with tokens. As the burst size was set to 100 KB, in the first second a burst of 100 KB of data packets will be sent at the full NIC rate of 10 Gbps. This can cause buffer overflows in switches. As seen in the first row of Table 3.3, 50% of the CTCP runs had packet retransmission rates higher than 62e-5% in the first second, while less than 10% of the HTCP runs had packet retransmissions (as seen in the second row). At the beginning of the data transfer, since HTCP **cwnd** is small, the HTCP layer does not pass down sufficient data to the `tc` layer to allow the latter to send a whole 100-KB burst at the NIC speed. Out of the 50 HTCP runs, retransmissions occurred in the first second in only 4 runs. Nevertheless, the mean retransmission rate for HTCP runs shown in Table 3.3 was high at 1349e-5 % because of the small number of packets sent in the first second. For example,

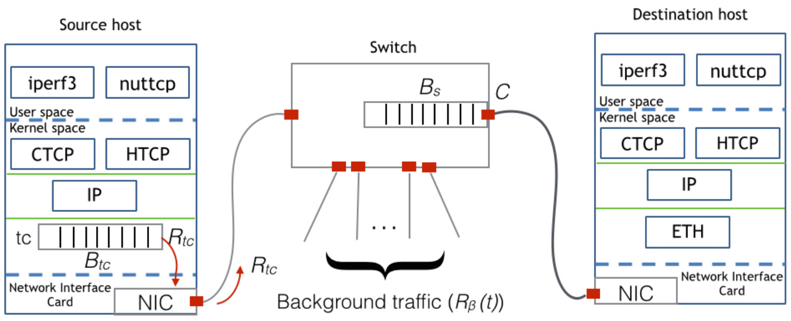


Figure 3.4: System model

the number of packets sent by HTCP in the first second in one run was 7182, while the number of retransmitted packets in the same second was 15 packets, which resulted in a high retransmission rate of 0.0021%. Rows 3 and 4 in the Packet Retransmission Rate section of Table 3.3 show that the reverse happens in later seconds. Given HTCP's dynamic modification of congestion window, bursts become more likely with HCTP than with CTCP, and therefore packet retransmission rates are slightly higher with HTCP.

In summary, CTCP is more suitable for use over rate-guaranteed paths because it does not incur a first-second hit in throughput, as does HTCP. Given that the penalty of path setup has been incurred, the dynamic search in the data plane for an appropriate TCP sending rate (as is required in a connectionless network) should be disabled. However, further experiments are required to understand the causes for the increased retransmission rate in the first second seen in Table 3.3 for CTCP. These experiments were conducted, and the results explained in the next chapter.

3.4 Cross-Layer Design

This section describes our cross-layer design for supporting high-speed large dataset transfers on circuits/VCs. Fig. 3.4 shows a system model with a source and a destination of an α-flow, and a representative switch whose egress interface happens to be the bottleneck on the α-flow path. The switch egress port has a capacity C and buffer size B_s as noted in Table 3.4. The α-flow is assigned to a circuit of rate R_{tc} and has round-trip time RTT_α. Background flows arriving at other interfaces of the switch that are destined to the same egress interface as the α flow are illustrated in Fig. 3.4.

Table 3.4: Parameters and Metrics

Parameter	Symbol
System-model parameters	
Switch bottleneck link capacity	C
Switch port buffer size	B_s
Circuit/VC rate (= source host `tc` rate)	R_{tc}
Round-trip time on α-flow path	RTT_α
Background traffic rate	$R_\beta(t)$
Protocol-layer parameters	
CTCP fixed congestion window size	`fcwnd`
Source host `tc` buffer size	B_{tc}
Output metrics	
Throughput of α-flow	T_α
Source host `tc` packet drop rate	d_{tc}
Packet loss rate of α-flow	p_α
Packet loss rate of background traffic	p_{bg}

Their cumulative rate is denoted $R_\beta(t)$ in Table 3.4.

A *control-plane* client (which could be executed on the source host or any external server, such as GlobusOnline [36]) sends a request for circuit/VC between the source and destination hosts to the SDN controller. If capacity is available on the path (depicted in this model with an abstraction of a single switch), the SDN controller configures the forwarding tables if the switches are packet switches, or configures the circuit if the switches are circuit switches. In case of virtual circuits, ideally, flow-classification, policing, and scheduling mechanisms should also be configured in the packet switches so that the α-flow packets are isolated into their own queue, and thus guaranteed the specified rate.

However, dynamic L2 path services being introduced by university campuses, regional and wide-area REN providers often support just the configuration of the forwarding tables, but not the configuration of QoS mechanisms such as policing and scheduling. In this case, the VCs can be viewed as *semi-rate-guaranteed* rather than fully rate-guaranteed. The SDN controller checks average utilization of links (e.g., by reading SNMP transmitted/received-byte counters in switches) before admitting a new circuit/VC request, but it does not configure the QoS mechanisms necessary for a full rate guarantee. If links are under utilized, this strategy will work most of the time.

However, if there are spikes in the background traffic that cause the cumulative arrival rate of flows destined to a specific output port of a switch to exceed the output-port capacity, then the switch buffer (as shown in Fig. 3.4) will fill up to the point where packets are dropped. Spikes can be just a few μs or ms in duration, while SNMP counter reads are spaced 10s or more apart to limit overhead. Therefore, even when average utilization, as estimated from SNMP counter values is deemed low, there can be short durations in which aggregate rates are high.

The *data-plane* path passes through the transport layer, IP layer (the IP header is processed only at the source and destination hosts if the end-to-end path is realized as an L2 VLAN or VLAN-MPLS-VLAN hybrid path), and Ethernet layer with the Linux traffic-control (`tc`) utility, as shown in Fig. 3.4. At the transport layer, we consider two protocol options: CTCP and HTCP. At the link-layer, we consider two `tc` queueing disciplines: TBF and HTB.

At the *transport-layer*, we used CTCP for rate-guaranteed circuits/VCs. But semi-rate-guaranteed VCs, we question whether CTCP is suitable, or whether HTCP is needed, and therefore compared CTCP and HTCP.

At the *link layer*, we propose the use of TBF for single circuits (single α flows), and HTB for multiple circuits (multiple α flows). For rare large dataset transfers between campuses, e.g., when a scientific researcher moves data between two university campus clusters, it is likely that there is only one α flow at a time, and TBF offers a good solution for controlling the sending rate of a single flow. On the other hand, dedicated Data Transfer Nodes (DTNs) deployed in large supercomputing centers often serve multiple elephant flows simultaneously. For outflows from such a DTN, HTB is required to control rates of individual flows based on their corresponding circuit/VC rates.

The next section describes the dependencies of the CTCP/HTCP and TBF/HTB parameters on path variables.

3.5 Parameter Selection

Besides the system model parameters, Table 3.4 lists the protocol-layer parameters and output metrics. At the transport layer, the parameter of interest is CTCP `fcwnd`, and at the `tc` layer, the parameter of interest is the buffer size, which is set by the `limit` parameter and `bfifo` in the TBF and HTB queueing disciplines, respectively. The output metrics are α-flow throughput, packet drop rate at the source by the `tc` layer, and overall packet loss rates of the α-flow and background traffic.

Cross-layer solutions are designed for three configurations: (i) single circuit/rate-guaranteed VC, (ii) multiple dynamic circuits/rate-guranteed VCs, and (iii) a semi-rate-guaranteed VC. These solutions are described below.

3.5.1 Single circuit/rate-guaranteed VC

To support a single data-transfer flow on a single circuit/rate-guaranteed VC from a sending host, we recommend using CTCP and TBF. HTCP is not required as no switch buffer overflows can occur, and TBF is sufficient as there is only one circuit and hence the multiple-class feature of the more-complex HTB is not required. The question addressed here is how should the application select suitable values for the CTCP parameter `fcwnd` and the TBF parameter `limit`. These values depend on circuit rate, R_{tc}, and round-trip time, RTT_{α}.

To keep the circuit full, `fcwnd` should be at least 1 BDP. If the delay (RTT_{α}) is measured using `ping`, the round-trip propagation delay will be accurately reflected in the obtained measurement. For an accurate measurement of transmission delay, `ping` should be executed with a packet of length equal to the maximum packet size (1500B for Ethernet) since maximum-sized packets will be used in large transfers. However, since `ping` sends only single packets, the packet transmission delay will be measured at the full NIC rate, rather than at the configured TBF `tc` rate (which is the circuit rate). Therefore, the transmission delay may be under-estimated even if maximum-sized packets are used by `ping`. Finally, if the circuit rate is high and round-trip propagation delay is small (i.e., in μs), then the processing delays in the TCP/IP layers, which will not be incurred by the ICMP packets used in `ping`, could become a factor. These factors will determine the multiplicative scaling parameter, m, used to set `fcwnd`:

$$fcwnd = m \times BDP \tag{3.1}$$

If the TBF `limit` parameter is set equal to `fcwnd`, there should be no packet drops at the `tc` layer. Since the congestion control code of TCP is disabled in CTCP, even the functionality that causes a TCP sender to enter the Congestion Window Reduced (CWR) state if it fails in its attempt to enqueue a packet in the link-layer buffer is disabled. Therefore, unlike in TCP, where packets are not dropped if the `tc` buffer is full, if a CTCP sender attempts to enqueue a packet in a full `tc` buffer, the packet will be dropped. The question is whether the `limit` parameter can be sized smaller than `fcwnd`.

Experimental setup: Experiments were run on two setups: (i) Setup1: two bare-metal (non-virtualized) Linux hosts of the U. Utah DDC InstaGENI rack, (ii) Setup2: two bare-metal Linux hosts connected by a single switch located on our campus. The `nuttcp-7.3.3` application was used to perform 15 memory-to-memory data transfers, each of which lasted 20 seconds. The parameter `tcp_wmem` was set to twice the BDP because Linux halves this value when sizing the send-side buffer. Similarly, the parameter `tcp_rmem` was set to twice the BDP at the receiver. To emulate circuits of different RTTs, `netem` was used to inject a fixed delay for all packets. The `tc` rate R_{tc} was also varied. The Linux command `taskset` was used to set the CPU affinity of the application to one specific CPU to ensure that the application can send data as fast as possible.

Table 3.5: Throughput T_α in Mbps, R_{tc}=1 Gbps, B_{tc}=`fcwnd`

Setup	RTT (ms)	fcwnd			
		0.3 BDP	BDP	1.2 BDP	1.3 BDP
Setup1	0.5	306	804	904	939
	50	286	939	939	939
Setup2	0.5	443	940	940	941
	50	288	726	818	941

Results: Table 3.5 shows the impact of the CTCP `fcwnd` parameter on α-flow throughput T_α. In Setup1, when RTT is low, it appears that the m parameter should be larger to achieve maximum throughput. The final column of Table 3.5 is the maximum achievable throughput. Table 3.5 also shows the impact of the hosts on the m parameter as the results for the two setups differ.

Fig. 3.5 shows the TBF `limit` parameter value required to guarantee zero packet drops by the

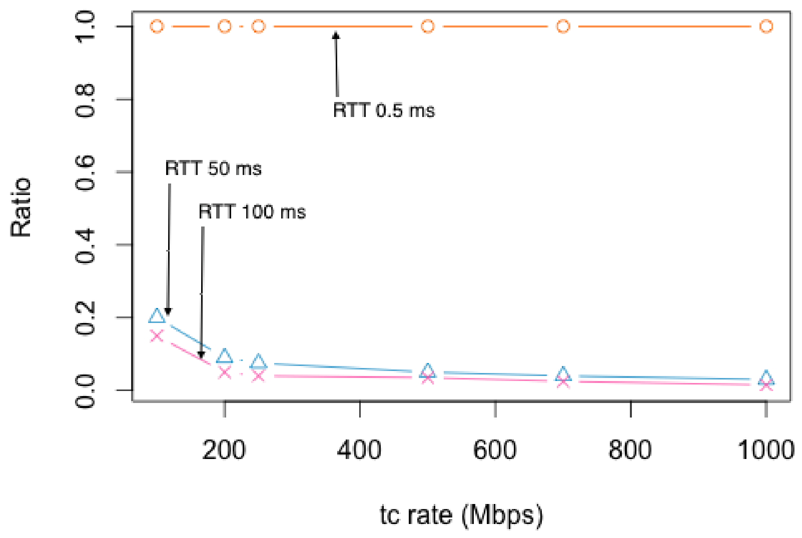


Figure 3.5: Required ratio of TBF `limit` to `fcwnd` to guarantee no `tc`-layer packet drops

`tc` layer. These graphs were generated from a series of experiments on Setup1 in which R_{tc} was varied as follows: 100 Mbps, 200 Mbps, 250 Mbps, 500 Mbps, 700 Mbps, and 1 Gbps, and the ratio of `limit` to `fcwnd` was varied from 0.01 to 1. There were no `tc`-layer packet drops when the ratio was 1.

Fig. 3.5 shows that a `limit` value as small as $0.2 \times$ `fcwnd` was sufficient on high-RTT paths to ensure no packet drops by `tc` even at the relatively low R_{tc} rate of 100 Mbps. At higher values of R_{tc}, the required value for B_{tc} decreases even reaching $0.015 \times$ `fcwnd` when the rate is 1 Gbps. On small-RTT paths (e.g., 0.5 ms), the required ratio was 1 (i.e., `limit` should be set to `fcwnd` to ensure no `tc`-layer packet drops), for all values of R_{tc}, even 1 Gbps. This is because B_{tc} is only 50 packets when RTT is 0.5 ms and R_{tc} is 1 Gbps. Therefore, a burst of packets from the CTCP layer to the `tc` layer can cause `tc` buffer overflows.

In *summary*, CTCP `fcwnd` should be set equal to at least $1.3 \times BDP$, and it is best to make TBF `limit` equal to `fcwnd` to ensure no `tc`-layer packet drops.

3.5.2 Multiple dynamic circuits/rate-guaranteed VC

The HTB queueing discipline is used as multiple classes are required at the link layer, one for each circuit, and CTCP is used for the transport layer. The application selects values for CTCP and HTB parameters for each circuit using the same principles described in Section 3.5.1. This section will address the challenge of handling dynamic arrivals and departures of α flows in DTNs.

The question addressed is what happens to ongoing α flows when: (i) HTB classes are added at the `tc` layer to support new circuits for newly arriving α flows, and (ii) circuits are released when α flows end. If multiple HTB child classes are defined under the same root class, bandwidth borrowing between the child classes is allowed. Such an approach would not work for circuits and rate-guaranteed VCs. Therefore, for multiple circuits, all HTB classes are defined as root classes, in which case bandwidth borrowing between classes is disallowed. Since bandwidth borrowing is disallowed, the HTB `ceil` parameter is set equal to `rate` for all classes. When an α flow ends, the bandwidth allocated to its circuit remains unused until another flow is assigned this bandwidth. To study the impact of new HTB class additions, an experimental study was executed.

A second question is whether different values of the CTCP `fcwnd` parameter should be set for different α flows as the circuit rate and RTT of α flows can vary. Ideally, such a customization of `fcwnd` should be done, but this requires modification of file-transfer applications to add the Linux `setsockopt` system call to modify `fcwnd` to the value appropriate for each α flow. For circuits, with complete path isolation and rate guarantees, there is no adverse impact of using a larger `fcwnd` value than necessary, and therefore in this experiment, `fcwnd` is set to a large value in a common Linux CTCP-module configuration file, which then applies to all CTCP flows.

Experimental setup: Two circuits were created from a source host: (i) Circuit-1: 500-Mbps circuit with RTT of 50 ms, and (ii) Circuit-2: 100-Mbps circuit with RTT of 10 ms. Different destinations were used for the two circuits. The CTCP `fcwnd` was set to $(1.2 \times BDP_1)$, where BDP_1 corresponds to the BDP of Circuit-1 (the larger of the two BDP values). A single HTB class was created with a rate R_{tc} corresponding to that of Circuit-1, with an attached `bfifo` leaf class whose size B_{tc} was set to the `fcwnd` value. The `nuttcp-7.3.3` application was used to create a CTCP flow (`flow1`) of

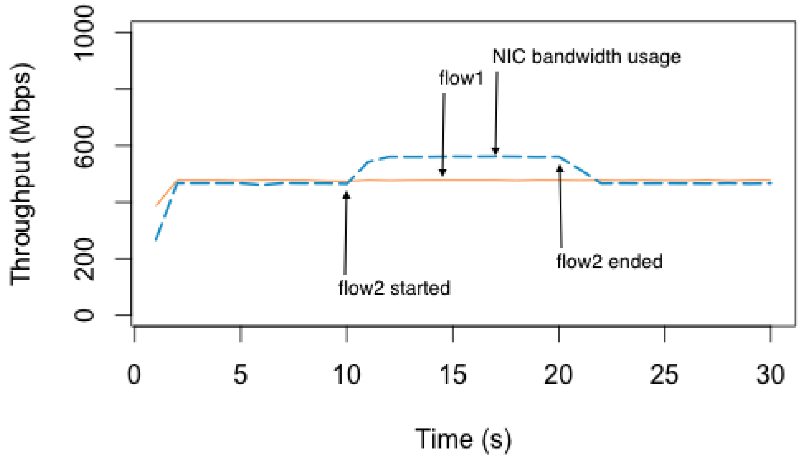
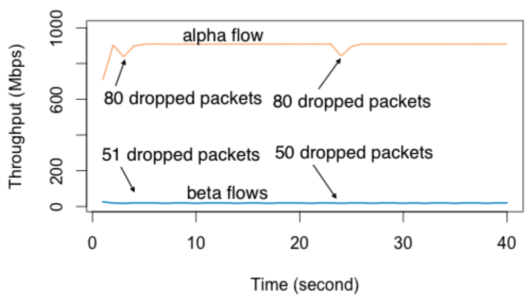


Figure 3.6: Throughput of two flows directed to two circuits from a single sender: `flow1` was started at time 0 and run for 30s, while `flow2` was started at time 10s and run for 10s

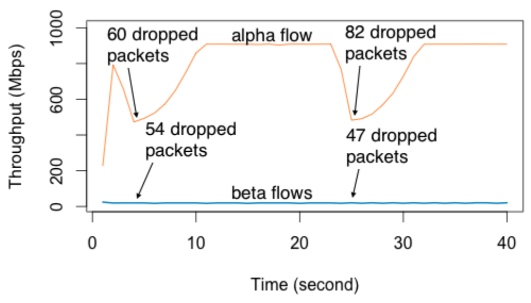
duration 30 seconds that was directed to Circuit-1. After 10 seconds, a second HTB root class was added with a rate R_{tc} corresponding to that of Circuit-2, with an attached `bfifo` leaf class whose size B_{tc} was set to the `fcwnd` value. A second CTCP flow (`flow2`), which was directed to Circuit-2, was then initiated and run for 10 seconds.

Results: Fig. 3.6 shows that `flow1` held its constant sending rate and achieved a throughput of 473.7 Mbps even as `flow2` was started and stopped. The sender NIC bandwidth usage was measured using the Linux utility `sar`, which was executed every sec. The total NIC bandwidth usage is the sum of the throughput values of `flow1` and `flow2`.

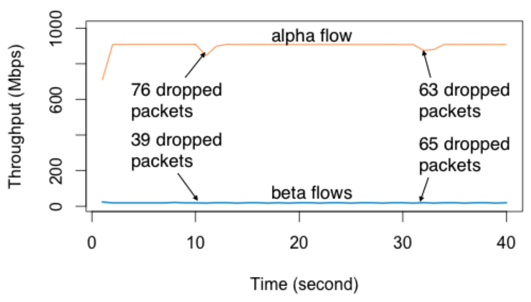
This experiment demonstrated that it is possible to dynamically add a new HTB class without affecting ongoing flows. No packet drops were observed in `flow1` during this dynamic addition of a new HTB class. Further when `flow2` ended, there was no bandwidth borrowing because both HTB classes were assigned as root classes, and hence `flow1` did not increase its sending rate to the 1 Gbps NIC rate. Similarly, there were no packet losses in `flow2`, and it achieved a throughput of 95.48 Mbps.



(a) CTCP



(b) HTCP, `tcp_wmem` $= 2 \times BDP$



(c) HTCP, `tcp_wmem` $= 4 \times BDP$

Figure 3.7: R_α = 950 Mbps, $R_\beta(t)$ = 20 Mbps, except for two 1-sec intervals when $R_\beta(t)$ = 100 Mbps

3.5.3 Semi-rate-guaranteed VC

As described in Section 3.4, university campus and REN providers often use coarse-granularity SNMP measurements of link utilization obtained to determine if there is sufficient headroom on a link to allow an α-flow to start sending data at a specified rate. The assumption is that if there is sufficient headroom, e.g., a link is utilized only at 30-40%, data-plane QoS mechanisms such as policing and scheduling are not required. However, on micro- and milli-sec, or even sec, scales, there can be spikes in the background traffic, which cause buffer overflows and hence losses in both the α-flow and in background flows. The purpose of this study is to determine whether CTCP or HTCP should be used on such semi-rate guaranteed VCs. Two experiments were conducted by using as background traffic: (i) TCP flows, and (ii) UDP flows.

Experiment 1: On a GENI slice consisting of three hosts in one rack, a CTCP α flow was generated from sender 1 to a receiver on a circuit of rate 950 Mbps, and a TCP flow was generated from sender 2 with a rate limit of 20 Mbps to the same receiver. Since the aggregate rate of both flows was less than the 1 Gbps link rate, there were no packet losses at the switch, and the α-flow throughput T_α was 908 Mbps.

Figs. 3.7a and 3.7b show that when a second TCP flow was generated from sender 2 with a rate limit of 80 Mbps, to create a total $R_\beta(t)$ of 100 Mbps traffic for two 1-sec intervals, at $t = 2$ and $t = 22$, this caused the switch buffer to fill up, and packets to be dropped on both the α flow and β flows. Fig. 3.7a shows that since CTCP does not change its sending rate when packet losses occur, the α-flow throughput recovered within 2 sec, while with HTCP, 8 sec were required to recover the full α-flow throughput as seen in Fig. 3.7b.

Since Linux halves the `tcp_wmem` value when setting the TCP sender-side buffer, we doubled `tcp_wmem` to $4 \times BDP$ to see its impact on HTCP throughput. To give the α flow a chance to increase its `cwnd` to $2 \times BDP$ before injecting the 80-Mbps packet-loss inducing flow, we changed the time of injection to $t = 10$ and $t = 31$. Fig. 3.7c shows that indeed when packet losses occurred, the HTCP sender dropped its `cwnd` by half, but since the `cwnd` had reached $2 \times BDP$ before the losses, the throughput drop was minimal. Therefore, with this `tcp_wmem` setting HTCP performed as well as CTCP retaining the high throughput of the α flow. From this experiment, it appears that if α-flow throughput is the primary concern, then either CTCP or HCTP can be used with appropriate parameter settings.

In this experiment, the background traffic $R_\beta(t)$ consisted of two TCP flows, both of which adjusted their sending rates when packet losses occurred. However, with UDP flows carrying real-time traffic, such a rate adjustment will not occur. Our next experiment compares CTCP and HTCP when the background traffic is UDP.

Experiment 2: On our campus setup, we generated an α flow with no rate limit (i.e., R_{tc} = 1 Gbps). A CTCP/HTCP α flow was run for 30 sec from sender 1 to a receiver with an emulated packet delay of 50 ms. A UDP `iperf3` flow from sender 2 to the same receiver was started at $t = 4s$ and stopped at $t = 14s$. The senders and receivers are connected by a single switch. The UDP-flow rate $R_\beta(t)$

was set to 60 Mbps in one run, and to 500 Mbps in another run.

Fig. 3.8a shows the results. The α-flow throughput T_α was initially 930 Mbps, but this throughput dropped to 881 Mbps in the presence of the 60-Mbps UDP flow. In this run, no packets were dropped. The switch buffer does fill up, but ACKs start arriving at the CTCP sender at a rate equal to the rate available to this flow on the switch-to-receiver link. This causes the effective sending rate of the CTCP α flow to drop preventing any buffer overflows. In other words, TCP self-clocking through ACKs appears to be working in CTCP.

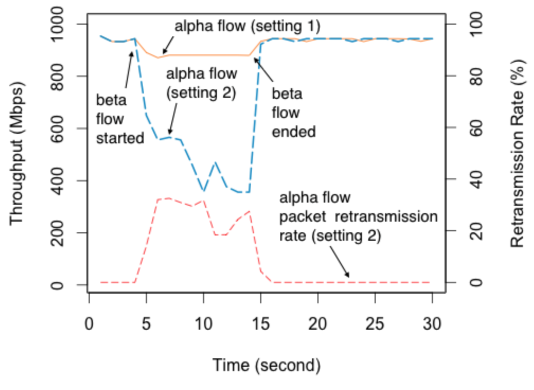
In the second run, when the UDP flow rate was 500 Mbps, packet losses occurred on both the α flow and the UDP flow. The CTCP α-flow packet retransmission rate p_α is shown in Fig. 3.8a, and the UDP-flow packet loss rate p_{bg} was 27%.

Fig. 3.8b shows the results from runs when HTCP was used instead of CTCP. It shows that on the run in which the UDP flow rate was 500 Mbps, HTCP dropped its throughput more quickly causing few packet losses both for itself and for the UDP flow. The packet loss rate on the UDP flow for this run was only 0.8%. However, the α-flow throughput was higher with CTCP (784 Mbps) than with HTCP (430 Mbps). When the UDP flow rate was 60 Mbps, HTCP had not increased its congestion window to a large enough value when the UDP flow started. Instead the HTCP congestion window kept growing making the α-flow throughput higher in the duration $t = (4, 14)s$ when the UDP-flow was present. On this run, there was a loss of 201 packets on the HTCP flow at $t = 20$, dropping the throughput to 734 Mbps at $t = 20$s, before increasing again.

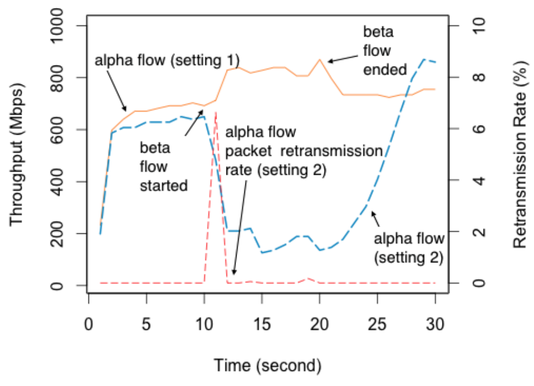
More broadly, these results show that in networks where spikes are not short lived but rather link utilization increases can last long periods such as 10s as in our above experiment, if priority is placed on β-flow performance, then HTCP should be used, while if priority is placed on α-flow performance, then CTCP should be used.

3.6 Conclusions

This chapter showed how to leverage transport- and link-layer protocols to enable the full use of high-rate circuits/VCs. High-rate large-sized data transfers were executed between hosts at the ends



(a) CTCP



(b) HTCP, `tcp_wmem`=4 $\times$ *BDP*

Figure 3.8: Throughput and packet retransmission rate of a CTCP α flow, with a UDP background (β) flow. The UDP flow was started at 4s and lasted for 10s. Setting 1: beta-flow rate = 60 Mbps, Setting 2: beta-flow rate = 500 Mbps

of an inter-domain L2 path that was previously provisioned across a multi-domain SDN. Throughput that almost matched the 4-Gbps circuit/VC rate was achieved, but packet losses were observed. The L2 path traverses multiple domains, and shares links and switch buffers with other real-traffic flows. The rate of the background flows cannot be controlled in this environment (i.e., the background flows is generated by other users of the network, e.g., Internet2). Designing a model to achieve a throughput that matches the circuit/VC rate with zero packet-loss is difficult giving the uncontrolled environment. Therefore, a new study was conducted in a testbed where we had more control over different parameters (e.g., background traffic rate, packet-loss rate, RTT).

A cross-layer design for support of large transfers across dynamically established circuits/VCs in Software Defined Networks (SDNs) was presented based on the new experimental study conducted in the controlled environment. For a single circuit/rate-guaranteed VC from a server, Circuit TCP (CTCP) and the Linux `tc` Token Bucket Filter (TBF) queueing discipline were recommended with methods for selecting parameter values. For a server handling multiple simultaneous large transfers over circuits/rate-guaranteed VCs, the combination of CTCP and Hierarchical Token Bucket (HTB) discipline was shown to be effective. Finally, for semi-rate-guaranteed VCs, CTCP was recommended if the primary consideration is for the large transfers, while HTCP is better if the primary consideration is for other flows.

Chapter 4

Lessons learned: Contributions to the experimental-networking research community

This chapter presents lessons learned while conducting the experiments presented in Chapter 3. Key findings are presented for three layers in the networking stack: (i) Application Layer, (ii) Transport Layer, and (iii) Data-Link Layer. Also, insights are provided on best practices for monitoring traffic and packet-trace analysis tools.

Section 4.1 describes the experimental setups. Sections 4.2.2, 4.3.5, and 4.4.4 present lessons learned for the application ayer, transport layer and data-link layers, respectively. Section 4.5.3 describes our insights on monitoring and packet-trace analysis.

4.1 Experimental Setup

Different setups were used to run the experiments described in this chapter:

Setup 1: This setup was created on the GENI infrastructure. Specifically, this experimental setup consists of two bare-metal (non-virtualized) hosts were used, one located at the University of California, Los Angeles (UCLA) and the other located at the University of Chicago (UChicago), interconnected via a 1-Gbps L2 path (created using VLAN IDs). Each host has a total of 32 cores, 49 GB RAM, 1 TB disk space, and two 1 Gbps Ethernet (GigE) network interface cards (NICs). One of the NICs is used for remote login access, while the other is used to carry the experimental traffic. The path RTT is 46 ms.

Setup 2: This setup was created on the multi-domain SDN described in Section 2.1.4. Two hosts were used, one located at the University of Virginia (UVA) and the other located at Indiana University (IU). The two hosts were connected by a 4-Gbps L2 path. Each host has two Intel Xeon E5620 four-core processors (for a total of eight cores), 24 GiB RAM, and two 10 GigE NICs and 1 GigE NIC (for remote login). Both hosts run kernel version 2.6.32. The path RTT is 26 ms.

Setup 3: This setup was created on the GENI UtahDDC, which is an InstaGENI rack. It consists of two bare-metal hosts interconnected via a single Ethernet top-of-the-rack switch. Each host has a total of 32 cores, 49 GB RAM, 1 TB disk space, and two 1 GigE NICs. The path RTT is 0.3 ms.

Setup 4: This setup was created on the GENI infrastructure. Two bare-metal hosts were used, one located at the University of Kentucky and the other located at the University of Illinois. The two hosts were connected by a 1-Gbps L2path. Each node has a total of 32 cores, 49 GB RAM, 1 TB disk space, and two 1 GigE NICs.

Setup 5: This setup was created on the GENI rack at the University of Kentucky. Three bare-metal hosts, H1, H2, H3, were used. H2 was configured as a gateway to perform IP-layer packet forwarding of all packets sent between H1 and H3. Each host has a total of 32 cores, 49 GB RAM, 1 TB disk space, and two 1 GigE NICs. The RTT between H1 to H2 is 0.3 ms, while the RTT between H1 to H3 is 0.5 ms.

4.2 Application Layer

4.2.1 Accuracy of network performance measurement tools (`iperf3` and `nuttcp`)

Objective: The `nuttcp` and `iperf3` tools are network performance measurement tools instrumented to report the achieved throughput and packet retransmissions. Such tools are used by researchers to test the performance of new protocols, new network designs, etc. In a previous experiment, we purposely injected packet losses on a path, and used `nuttcp-6.1.2` to measure the throughput and packet-loss rate. However, `nuttcp-6.1.2` reported 0 retransmissions, which made us question the accuracy of this tool, and motivated us to verify the number of retransmissions because these tools are used extensively by the GENI community and other networking researchers.

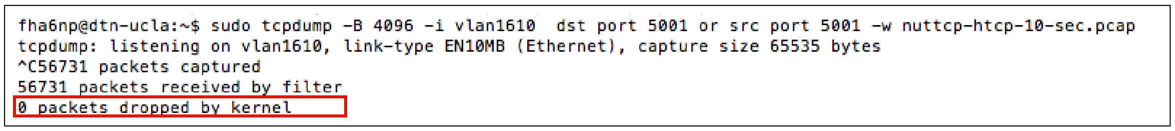
Key findings: Experimental results confirmed the accuracy of `iperf3` and `nuttcp-7.3.3`. Therefore, these applications are suitable for use as network performance measurement tools. However, we found the number of retransmissions reported by `nuttcp-6.1.2` to be inaccurate.

Methodology: The method consists of the following steps: (i) A packet sniffer was initiated at the sender to capture the application packets. (ii) The application (`iperf3` or `nuttcp`) was used to perform memory-to-memory data transfers. (iii) The packet sniffer was terminated. (iv) A packet analyzer was used to process the captured packet trace to determine the number of retransmissions on the application flow. (v) The number of retransmissions reported by the packet analyzer was compared against the number of retransmissions reported by the application.

Experiment setup and execution: The experiment was conducted using Setup 1. The following tools were used: `tcpdump` was used to collect the application packets, and `tshark` version 1.6.7 and `tcptrace` version 6.6.7 were used to analyze packet traces. Two versions of `nuttcp`: version 6.1.2 (the default GNU package) and version 7.3.3 (the latest version), and `iperf3` version 3.0.7 were tested. A shell script was used to run the following commands: (i) initiate `tcpdump` to capture all packets sent by the application, (ii) initiate the application (`nuttcp` or `iperf3`) to perform memory-to-memory data transfers for ten seconds from UCLA to UChicago, (iii) terminate `tcpdump`. HTCP was used as the congestion control scheme at the sender, and artificial packet losses were injected randomly at the receiver using the Linux traffic-control (`tc`) `netem` utility. For each application (`nuttcp-6.1.2`, `nuttcp-7.3.3`, and `iperf3`), a total of 100 runs were performed.

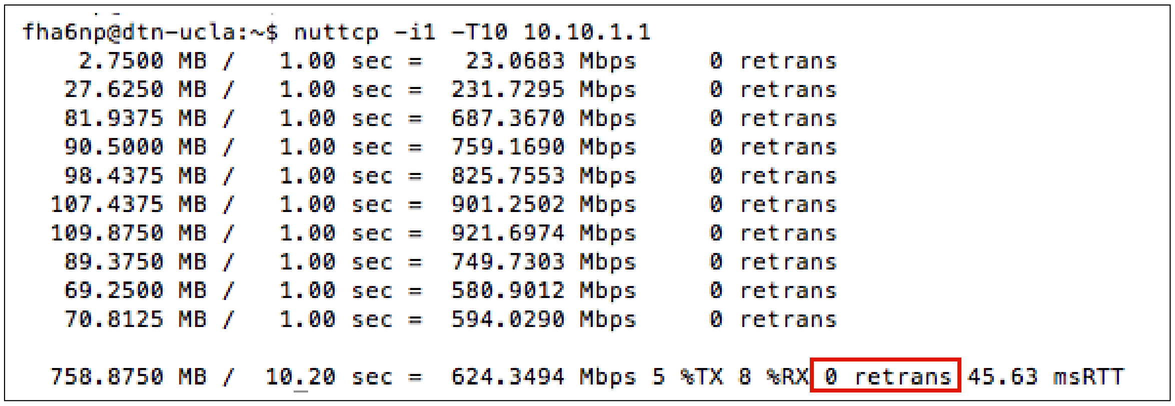
Results of `nuttcp-6.1.2` tests: Fig. 4.1 shows the standard output of `tcpdump`, which reports that no packets were dropped by the kernel, which means `tcpdump` was able to capture all packets of the application flow (a capture filter can be set to configure `tcpdump` to only capture packets of a particular flow, which may be necessary to capture all packets sent on a high-speed link). As no packets were dropped, an analysis of the packet trace should find the exact number of retransmissions.

Fig. 4.2 shows the standard output of `nuttcp-6.1.2`, which reports zero retransmissions. However, Fig. 4.3 shows that both `tshark` and `tcptrace` reported 45 retransmissions (refer to section 4.5.3 to understand why `tshark` reports all/some of the retransmitted packets as out-of-order). The `nuttcp-6.1.2` tool reported zero retransmissions in all the 100 runs, while the numbers reported by `tshark` and `tcptrace` were all non-zero, which was expected since packet losses were injected deliberately.



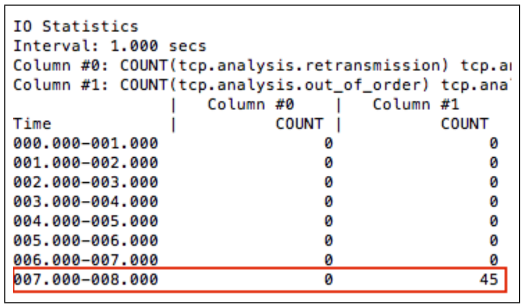
```
fha6np@dtn-ucla:~$ sudo tcpdump -B 4096 -i vlan1610  dst port 5001 or src port 5001 -w nuttcp-htcp-10-sec.pcap
tcpdump: listening on vlan1610, link-type EN10MB (Ethernet), capture size 65535 bytes
^C56731 packets captured
56731 packets received by filter
0 packets dropped by kernel
```

Figure 4.1: `tcpdump` standard output



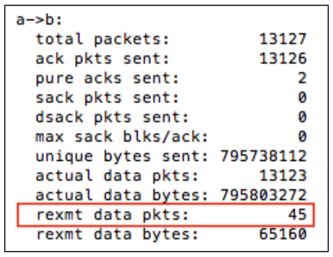
```
fha6np@dtn-ucla:~$ nuttcp -i1 -T10 10.10.1.1
    2.7500 MB /   1.00 sec =   23.0683 Mbps     0 retrans
   27.6250 MB /   1.00 sec =  231.7295 Mbps     0 retrans
   81.9375 MB /   1.00 sec =  687.3670 Mbps     0 retrans
   90.5000 MB /   1.00 sec =  759.1690 Mbps     0 retrans
   98.4375 MB /   1.00 sec =  825.7553 Mbps     0 retrans
  107.4375 MB /   1.00 sec =  901.2502 Mbps     0 retrans
  109.8750 MB /   1.00 sec =  921.6974 Mbps     0 retrans
   89.3750 MB /   1.00 sec =  749.7303 Mbps     0 retrans
   69.2500 MB /   1.00 sec =  580.9012 Mbps     0 retrans
   70.8125 MB /   1.00 sec =  594.0290 Mbps     0 retrans

  758.8750 MB /  10.20 sec =  624.3494 Mbps 5 %TX 8 %RX 0 retrans 45.63 msRTT
```

Figure 4.2: `nuttcp-6.1.2` standard output



```
IO Statistics
Interval: 1.000 secs
Column #0: COUNT(tcp.analysis.retransmission) tcp.ai
Column #1: COUNT(tcp.analysis.out_of_order) tcp.ana
                |   Column #0    |   Column #1
Time            |          COUNT |          COUNT
000.000-001.000                0                0
001.000-002.000                0                0
002.000-003.000                0                0
003.000-004.000                0                0
004.000-005.000                0                0
005.000-006.000                0                0
006.000-007.000                0                0
007.000-008.000                0               45
```

(a) `tshark`

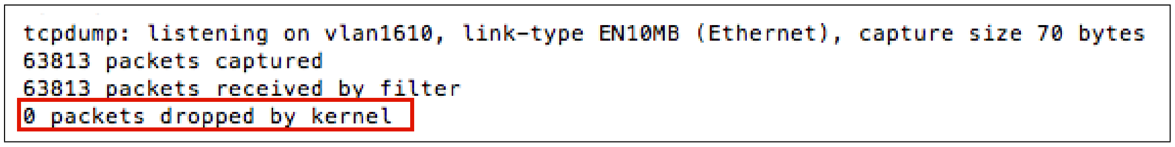


```
a->b:
  total packets:          13127
  ack pkts sent:          13126
  pure acks sent:             2
  sack pkts sent:             0
  dsack pkts sent:            0
  max sack blks/ack:          0
  unique bytes sent:  795738112
  actual data pkts:       13123
  actual data bytes:  795803272
  rexmt data pkts:           45
  rexmt data bytes:       65160
```

(b) `tcptrace`

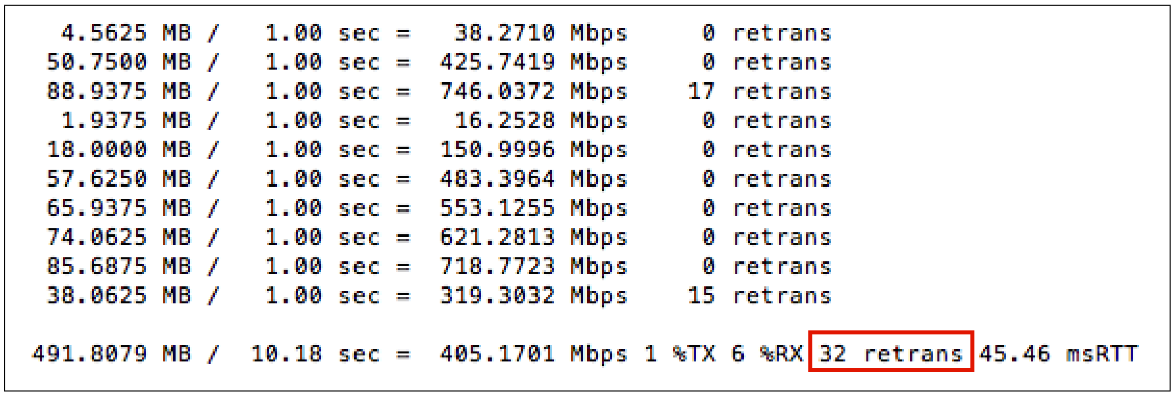
Figure 4.3: Number of retransmissions reported by `tshark` and `tcptrace`

Results of `nuttcp-7.3.3` tests: Fig. 4.4 shows the standard output of `tcpdump`, which confirms that all packets were captured. Fig. 4.5 shows that the standard output of `nuttcp-7.3.3` reports that were 32 retransmissions, which matched the `tshark` and `tcptrace` reports as seen in Fig. 4.6. In all the 100 runs, the number of retransmissions reported by `nuttcp-7.3.3` matched the number of retransmissions reported by `tshark` and `tcptrace`.



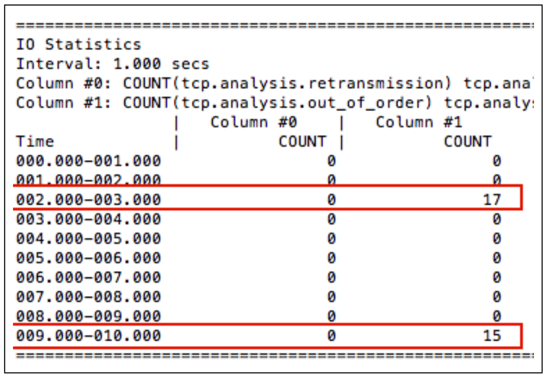
```
tcpdump: listening on vlan1610, link-type EN10MB (Ethernet), capture size 70 bytes
63813 packets captured
63813 packets received by filter
0 packets dropped by kernel
```

Figure 4.4: `tcpdump` standard output



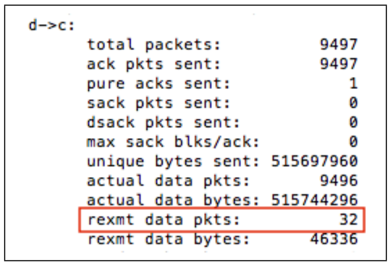
```
   4.5625 MB /   1.00 sec =   38.2710 Mbps     0 retrans
  50.7500 MB /   1.00 sec =  425.7419 Mbps     0 retrans
  88.9375 MB /   1.00 sec =  746.0372 Mbps    17 retrans
   1.9375 MB /   1.00 sec =   16.2528 Mbps     0 retrans
  18.0000 MB /   1.00 sec =  150.9996 Mbps     0 retrans
  57.6250 MB /   1.00 sec =  483.3964 Mbps     0 retrans
  65.9375 MB /   1.00 sec =  553.1255 Mbps     0 retrans
  74.0625 MB /   1.00 sec =  621.2813 Mbps     0 retrans
  85.6875 MB /   1.00 sec =  718.7723 Mbps     0 retrans
  38.0625 MB /   1.00 sec =  319.3032 Mbps    15 retrans

 491.8079 MB /  10.18 sec =  405.1701 Mbps 1 %TX 6 %RX 32 retrans 45.46 msRTT
```

Figure 4.5: `nuttcp-7.3.3` standard output



```
=======================================================
IO Statistics
Interval: 1.000 secs
Column #0: COUNT(tcp.analysis.retransmission) tcp.ana
Column #1: COUNT(tcp.analysis.out_of_order) tcp.analy
                        |   Column #0     |   Column #1
Time                    |           COUNT |           COUNT
000.000-001.000                         0               0
001.000-002.000                         0               0
002.000-003.000                         0              17
003.000-004.000                         0               0
004.000-005.000                         0               0
005.000-006.000                         0               0
006.000-007.000                         0               0
007.000-008.000                         0               0
008.000-009.000                         0               0
009.000-010.000                         0              15
=======================================================
```

(a) `tshark`



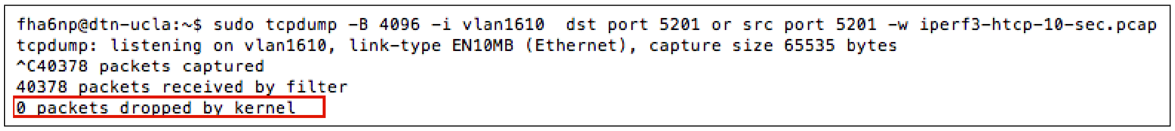
```
d->c:
      total packets:            9497
      ack pkts sent:            9497
      pure acks sent:              1
      sack pkts sent:              0
      dsack pkts sent:             0
      max sack blks/ack:           0
      unique bytes sent:   515697960
      actual data pkts:         9496
      actual data bytes:   515744296
      rexmt data pkts:            32
      rexmt data bytes:        46336
```

(b) `tcptrace`

Figure 4.6: Number of retransmitted packets reported by `tshark` and `tcptrace`, unnecessarily information was omitted

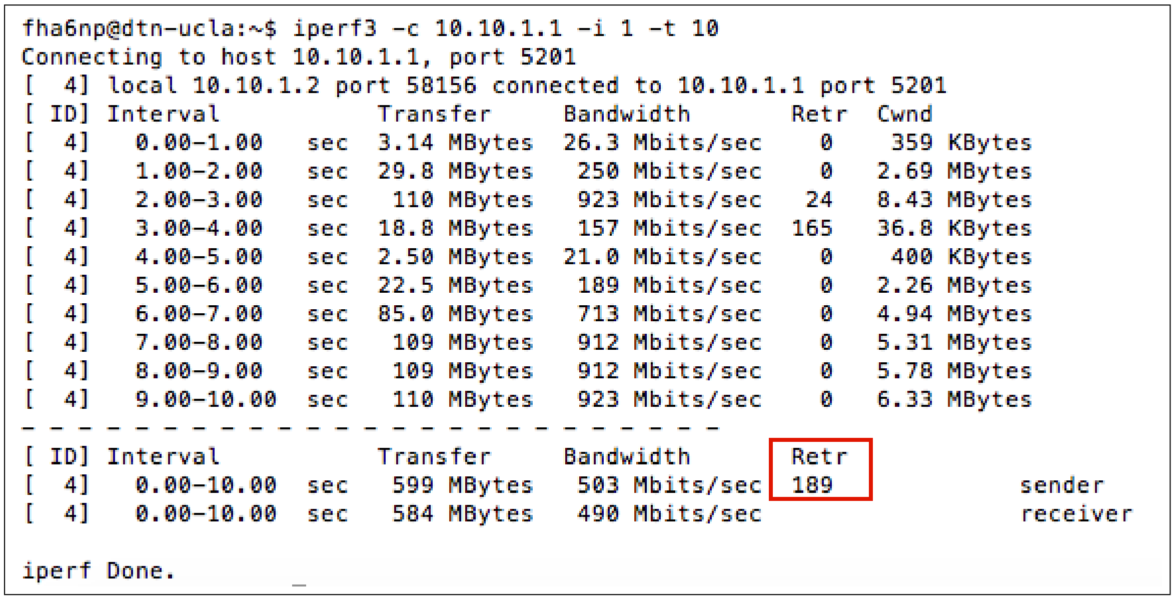
Results of `iperf3` tests: Fig. 4.7 shows the standard output of `tcpdump`, which confirms that all packets were captured. The standard output of `iperf3`, shown in Fig. 4.8, reports 189 retransmissions, which matches the `tshark` and `tcptrace` reports shown in Fig. 4.9. There were however, 2 runs out of the 100 runs, in which the number of retransmissions reported by `iperf3` did not match the number of retransmissions reported by `tshark` and `tcptrace`.

Conclusions: Of the three network performance measurement tools, `nuttcp-7.3.3` was the most trustworthy in its reports on the number of retransmissions, `nuttcp-6.1.2` was inaccurate all the time, and `iperf3` was inaccurate in 2 out of 100 runs. As the default GNU package includes `nuttcp-6.1.2`, we caution researchers on using this tool, and recommend downloading the source code of version 7.3.3 and building an executable.



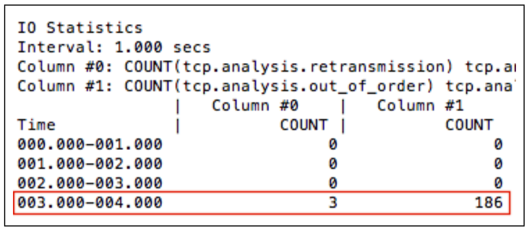
```
fha6np@dtn-ucla:~$ sudo tcpdump -B 4096 -i vlan1610  dst port 5201 or src port 5201 -w iperf3-htcp-10-sec.pcap
tcpdump: listening on vlan1610, link-type EN10MB (Ethernet), capture size 65535 bytes
^C40378 packets captured
40378 packets received by filter
0 packets dropped by kernel
```

Figure 4.7: `tcpdump` standard output



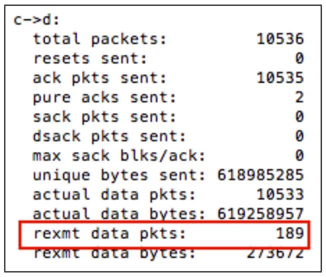
```
fha6np@dtn-ucla:~$ iperf3 -c 10.10.1.1 -i 1 -t 10
Connecting to host 10.10.1.1, port 5201
[  4] local 10.10.1.2 port 58156 connected to 10.10.1.1 port 5201
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec  3.14 MBytes  26.3 Mbits/sec    0    359 KBytes
[  4]   1.00-2.00   sec  29.8 MBytes   250 Mbits/sec    0   2.69 MBytes
[  4]   2.00-3.00   sec   110 MBytes   923 Mbits/sec   24   8.43 MBytes
[  4]   3.00-4.00   sec  18.8 MBytes   157 Mbits/sec  165   36.8 KBytes
[  4]   4.00-5.00   sec  2.50 MBytes  21.0 Mbits/sec    0    400 KBytes
[  4]   5.00-6.00   sec  22.5 MBytes   189 Mbits/sec    0   2.26 MBytes
[  4]   6.00-7.00   sec  85.0 MBytes   713 Mbits/sec    0   4.94 MBytes
[  4]   7.00-8.00   sec   109 MBytes   912 Mbits/sec    0   5.31 MBytes
[  4]   8.00-9.00   sec   109 MBytes   912 Mbits/sec    0   5.78 MBytes
[  4]   9.00-10.00  sec   110 MBytes   923 Mbits/sec    0   6.33 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec   599 MBytes   503 Mbits/sec  189             sender
[  4]   0.00-10.00  sec   584 MBytes   490 Mbits/sec                  receiver

iperf Done.
```

Figure 4.8: `iperf3` standard output



```
IO Statistics
Interval: 1.000 secs
Column #0: COUNT(tcp.analysis.retransmission) tcp.a
Column #1: COUNT(tcp.analysis.out_of_order) tcp.ana
                |   Column #0    |   Column #1
Time            |          COUNT |           COUNT
000.000-001.000                0                0
001.000-002.000                0                0
002.000-003.000                0                0
003.000-004.000                3              186
```

(a) `tshark`



```
c->d:
  total packets:          10536
  resets sent:                0
  ack pkts sent:          10535
  pure acks sent:             2
  sack pkts sent:             0
  dsack pkts sent:            0
  max sack blks/ack:          0
  unique bytes sent: 618985285
  actual data pkts:       10533
  actual data bytes: 619258957
  rexmt data pkts:          189
  rexmt data bytes:      273672
```

(b) `tcptrace`

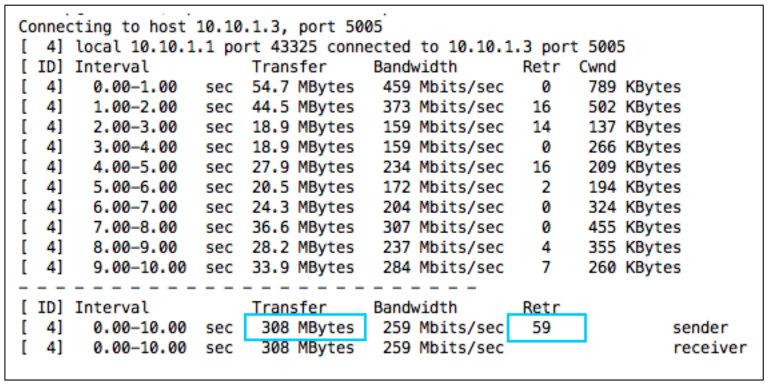
Figure 4.9: Number of retransmitted packets reported by `tshark` and `tcptrace`

4.2.2 Computing packet retransmission rate

While the previous section discussed issues related to the number of retransmitted packets, often researchers need to report packet retransmission rates. Accuracy in reporting packet retransmission rate is required in some contexts. For example, when comparing two transport protocols across wide-area shared networks, the congestion-control algorithms in these protocols may vary slightly causing different packet retransmission rates. A protocol that increases its sending rate more aggressively may cause higher packet losses at router buffers. Under such circumstances, even a seemingly small mistake in the assumptions could result in wrong conclusions.

To compute packet retransmission rate, the number of retransmitted packets, and the total number of transmitted packets, are required. But as seen in Section 4.2.1, network performance measurement tools, `iperf3` and `nuttcp` report only the number of retransmitted packets. Instead of the total number of transmitted packets, these tools report the number of bytes sent, as seen in Fig. 4.10. Packet size then becomes necessary to determine the total number of packets transmitted.

A commonly used assumption is that TCP payload is 1460 bytes because the maximum payload size of a standard Ethernet frame is 1500 B, and typically IP header is 20 bytes and TCP header is 20 bytes. However, by capturing and analyzing packet traces, with `tcpdump` and `tshark`, respectively, we found that the length of the TCP payload generated when using `iperf3` was 1448 bytes. This is because `iperf3` calls certain socket options, which causes TCP to add options in its header. The total size of these options is 12 bytes. When comparing packet retransmission rates on lossy paths, or for small data transfers, using the correct TCP payload length can increase accuracy of the



```
Connecting to host 10.10.1.3, port 5005
[  4] local 10.10.1.1 port 43325 connected to 10.10.1.3 port 5005
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec  54.7 MBytes  459 Mbits/sec    0    789 KBytes
[  4]   1.00-2.00   sec  44.5 MBytes  373 Mbits/sec   16    502 KBytes
[  4]   2.00-3.00   sec  18.9 MBytes  159 Mbits/sec   14    137 KBytes
[  4]   3.00-4.00   sec  18.9 MBytes  159 Mbits/sec    0    266 KBytes
[  4]   4.00-5.00   sec  27.9 MBytes  234 Mbits/sec   16    209 KBytes
[  4]   5.00-6.00   sec  20.5 MBytes  172 Mbits/sec    2    194 KBytes
[  4]   6.00-7.00   sec  24.3 MBytes  204 Mbits/sec    0    324 KBytes
[  4]   7.00-8.00   sec  36.6 MBytes  307 Mbits/sec    0    455 KBytes
[  4]   8.00-9.00   sec  28.2 MBytes  237 Mbits/sec    4    355 KBytes
[  4]   9.00-10.00  sec  33.9 MBytes  284 Mbits/sec    7    260 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec   308 MBytes  259 Mbits/sec   59             sender
[  4]   0.00-10.00  sec   308 MBytes  259 Mbits/sec                  receiver
```

Figure 4.10: `iperf3` log

reported packet retransmission rate.

4.3 Transport Layer

In Linux, TCP parameters are specified in `/proc/sys/net/ipv4`; these parameters can be changed to achieve different goals. Our objective is to study the impact of the following TCP parameters: `tcp_rmem`, `tcp_wmem`, `tcp_window_scaling`, and `tcp_moderate_rcvbuf` because these parameters impact the achieved throughput.

4.3.1 Experiment setup and execution

Setup 3 was used to conduct the experiments described in this section. An artificial delay of 10 ms was injected for each packet using `netem` at the receiver. The `iperf3` application was used to create an HTCP flow for 5 seconds. The `tcpdump` tool was used to collect all packets of the flow, and `tshark` was used to analyze the collected packet trace.

4.3.2 TCP receive buffer (`tcp_rmem`)

The TCP receive buffer size places an upper limit on the TCP (advertised) receive window size (`rwnd`), which in turn determines the TCP sending window size. The TCP sending window size is the smaller of two values, congestion window and receive window, i.e., min(`cwnd`, `rwnd`). Since the TCP sending window determines the instantaneous sending rate, the receive buffer size impacts TCP throughput.

In Linux, the receive buffer size is controlled by `tcp_rmem`, which is an integer vector of size 3: [`min`, `default`, `max`]. The `min` value of `tcp_rmem` is used when the system is under memory pressure. The `default` value influences the initial receive buffer size, which is allowed to grow up to the `max` value.

To study the impact of `tcp_rmem` on the receive window size, we conducted two sets of experiments. In the first set of experiments, the same value was used to set `default` and `max`. In the second set of experiments, we used different values for `default` and `max`. In both sets of experiments, at the sender, the `default` and `max` values of `tcp_wmem` were set to $2 \times BDP$ to ensure that the congestion window was not a limiting factor. The receive window values were determined from the collected packet traces. The `Win` field, which is part of the TCP header, in ACKs (and other segments) flowing from the receiver to the sender carries the receive window size that is relevant to the flow from the sender to the receiver.

Experiment 1: Table 4.1 shows the impact of `tcp_rmem` on the receiver window size and on the achieved throughput. In all the runs, we found that the initial receive window size was 29184 bytes (which is less than half of `tcp_rmem default`), but the value grew to almost half of the `max` value of `tcp_rmem`. Then the receive window size stayed constant at half of the `max` value of `tcp_rmem`. An examination of Linux Network-stack code confirmed this observation. Table 4.1 shows that when `tcp_rmem default` and `max` were set equal to the path BDP the achieved throughput was less than 1 Gbps. To achieve the maximum possible throughput on a path, TCP sending window size should grow to equal the path BDP. If the receive window size is limited to a value smaller than BDP, which will occur if the TCP `tcp_rmem` values are small, the maximum possible throughput will not be achieved. Therefore, to allow the receive window size to reach BDP, `tcp_rmem max` should be set to $2 \times BDP$.

Experiment 2: In this experiment, we study the impact of `tcp_rmem default` by executing two runs: run 1: `tcp_rmem default`=BDP, run 2: `tcp_rmem default`=$2 \times BDP$. Figure 4.11a and Figure 4.11b show the receive window size carried in all ACKs and in the first 115 ACKs, respectively.

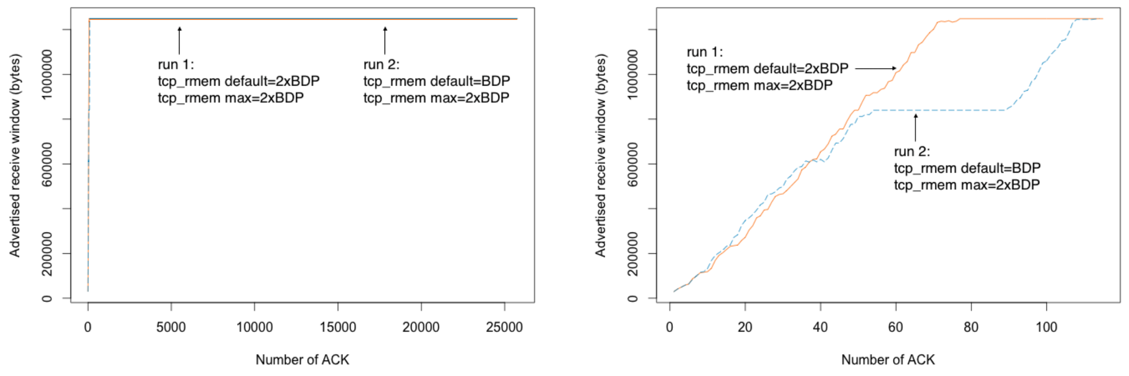
Table 4.1: Experiment 1: Impact of `tcp_rmem` on the maximum achieved receive window size and throughput. The `tcp_wmem` was set to $2 \times BDP$ to ensure that the TCP congestion window was not a limiting factor

`tcp_rmem default` (bytes)	`tcp_rmem max` (bytes)	maximum `rwnd` (bytes)	Throughput (Mbps)
BDP (1250000)	BDP (1250000)	$0.5 \times BDP$ (623616)	452
$2 \times BDP$ (2500000)	$2 \times BDP$ (2500000)	BDP (1248768)	910

The difference between the two runs is seen in Figure 4.11b. In run 1, since `tcp_rmem default` was set to $2 \times BDP$, the receive window grew faster than in run 2 where `tcp_rmem default` was set to just BDP. However, the time taken by the receive window size to reach its maximum value of 1248768 bytes was 189 μs for run 1 and 200 μs for run 2. Therefore, the difference between these times was relatively small.

The previous experiment showed that `tcp_rmem default` had only a small impact on the growth of the receive window size. When using a congestion control protocol such as HTCP, which goes through a slow-start phase to build its congestion window, starting with a receive window size that is smaller than the path BDP may not significantly degrad throughput. However, when CTCP is used, in which Slow Start is not executed, and the congestion window is fixed to a value slightly larger than BDP, if the receive window size was smaller than BDP, the CTCP sending window would be limited by the receive window size. Therefore, when using CTCP we suggest setting `tcp_rmem default` and `max` to $2 \times BDP$.

Other Linux `rmem` parameters (`rmem_max` and `rmem_default`) can be found in `/proc/sys/core/`. These parameters are general and used to allocate buffers for all protocols, e.g., UDP. TCP Linux manual page [37] mentions that `tcp_rmem default` overwrites `rmem_default`, while `tcp_rmem max` does not overwrite `rmem_max`. However, in all the previous experiments, `rmem_default` and `rmem_max` were set to 625000 bytes, which is smaller than the maximum achieved receive window size (1248768 bytes). If `tcp_rmem max` does not overwrite `rmem_max`, then the receive window size would have grown to 625000 bytes and stayed at this value.



(a) For all ACKs

(b) For the first 115 ACKs

Figure 4.11: Experiment 2: receive window size obtained from ACKs in the packet trace

4.3.3 TCP send buffer (`tcp_wmem`)

The TCP send buffer size places an upper limit on the TCP congestion window size (`cwnd`), which in turn determines the TCP sending window size. The TCP sending window size is the smaller of two values, congestion window and receive window, i.e., min(`cwnd`, `rwnd`). Since the TCP sending window determines the instantaneous sending rate, the send buffer size impacts TCP throughput.

In Linux, the receive buffer size is controlled by `tcp_wmem`, which is an integer vector of size 3: [`min`, `default`, `max`]. The `min` value of `tcp_wmem` is used when the system is under memory pressure. The `default` value influences the initial send buffer size, which is allowed to grow up to the `max` value.

To study the impact of `tcp_wmem` on the congestion window size, we conducted a set of experiments. The same value was used to set `default` and `max`. At the receiver, the `default` and `max` values of `tcp_rmem` were set to $2 \times BDP$ to ensure that the receive window was not a limiting factor. The congestion window values were determined from `iperf3` standard output, in which the congestion window size is reported in bytes every second.

Table 4.2 shows the impact of `tcp_wmem` on the congestion window size (`cwnd`) and on the achieved throughput. In all the runs, we found that the initial congestion window size was 14600 bytes (10 packets, but `cwnd` grew to a value that is slightly higher than half of the `max` value of

Table 4.2: Impact of `tcp_wmem` on the maximum achieved congestion window size and throughput. The `tcp_rmem` was set to $2 \times BDP$ to ensure that the TCP receive window was not a limiting factor

`tcp_wmem default` (bytes)	`tcp_wmem max` (bytes)	`cwnd` (bytes)	Throughput (Mbps)
BDP (1250000)	BDP (1250000)	1060000	659
$2 \times BDP$ (2500000)	$2 \times BDP$ (2500000)	1270000	911

`tcp_wmem`). An examination to Linux Network-stack code confirmed that TCP initial window size is 10 packets. The sending host TCP implementation followed RFC 6928, which increased the TCP initial window size to 10 packets instead of 1, 2 or 4 packets as in the previous TCP standard.

Table 4.2 shows that when `tcp_wmem default` and `max` were set equal to the path BDP, the achieved throughput was less than 1 Gbps. To achieve the maximum possible throughput on the path, TCP `cwnd` should grow to equal the path BDP. If the `cwnd` is limited to a value smaller than BDP (e.g., by a small `tcp_wmem` setting), the maximum possible throughput will not be achieved. Therefore, to allow `cwnd` to reach BDP, `tcp_rmem max` should be set to $2 \times BDP$. Table 4.2 shows that with this setting, throughput reaches close to 1 Gbps.

An interesting finding is that current TCP implementations follow RFC 2861, which states that the congestion window size is not increased unless the application has provided sufficient data to fill up the current congestion window. In our experiment, as we used HTCP, the `htcp_cong_avoid()` function is called when an ACK is received, and the TCP connection is in the congestion avoidance state. This function check whether TCP is limited by the application or not by calling another function named `tcp_is_cwnd_limited()`. This point is important because when sending large datasets, the application will keep providing file data to the TCP layer, and as long as `tcp_wmem` is large enough, the congestion window size will keep growing. If the TCP sending window is larger than switch buffers, packet losses could occur.

Other Linux `wmem` parameters (`wmem_max` and `wmem_default`) can be found in `/proc/sys/core/`. These parameters are general and are used to allocate buffers for all protocols, e.g., UDP. The TCP Linux manual page [37] mentions that `tcp_wmem default` overwrites `wmem_default`, while `tcp_wmem max` does not overwrite `wmem_max`. However, in all the previous experiments, `wmem_default` and `wmem_max` were set to 625000 bytes, which is smaller than the maximum achieved congestion window size (1270000 bytes). If `tcp_wmem max` does not overwrite `wmem_max`, then the congestion window size would have grown to 625000 bytes and stayed at this value.

4.3.4 TCP window scaling (`tcp_window_scaling`)

The `tcp_window_scaling` parameter is used to allow TCP to advertise a receive window size larger than 64 KB. If the receive window size is limited to 64 KB, since the TCP sending window size is the smaller of the congestion window and receive window sizes, the TCP sending window will be limited to 64 KB. Hence, the maximum possible throughput will not be achieved on high-BDP paths.

To study the impact of this window scaling feature, an experiment was conducted. The `tcp_wmem` and `tcp_rmem` (`default` and `max`) values were set to $2 \times BDP$ to achieve 1 Gbps throughput. HTCP was used as the congestion-control protocol, and `iperf3` was used to create a TCP flow for 5 seconds. The receive window values were obtained from the collected packet traces.

Table 4.3 shows that when the window scaling feature was disabled, TCP was not able to advertise a receive window size larger than 65536 bytes. The TCP sending window size was limited by the small receive window size. Therefore, the achieved throughput was only 51.7 Mbps, far smaller than the maximum possible path throughput of 1 Gbps. On the other hand, when the window scaling feature was enabled, the maximum achieved receive window size was 1250048 bytes, and throughput reached close to 1 Gbps. The BDP on this path was 1250000 bytes, and hence window scaling was needed, since $2 \times BDP$ should be used for both the send and receive side buffers as recommended in Sections 4.3.2 and 4.2.

In the TCP header, only 16 bits are used to represent the window size. The maximum value that can be represented with 16 bits is 65535, which means that the maximum receive window size is 65536 bytes. To overcome this limitation, a TCP option was used to increase the possible maximum window size. A one-byte option is used to represent the window scaling value, which is then used to compute the actual receive window size by performing a left shift operation on the window value. For example, if TCP at the receiver want to advertise a window of size of 131070 bytes, the window

Table 4.3: `tcp_window_scaling` impact on the maximum receive window size and throughput

`tcp_window_scaling`	maximum `rwnd` (bytes)	Throughput (Mbps)
0 (disabled)	65160	51.7
1 (enabled)	1250048	910

size in TCP header would be set to 65535, and the window scaling field would be set to 1. As a result, the window size advertised in the TCP header will be left shifted by one bit (which is the same as multiplying it by 2), and the window size would be $65535 \times 2 = 131070$ bytes. The TCP receiver sets the scaling value in the option field based on the available TCP receive buffer size as long as the `tcp_window_scaling` feature is enabled. Therefore, the window scaling feature should not be disabled on high-speed networks to effectively serve high-BDP paths.

4.3.5 TCP receive buffer auto-tuning (`tcp_moderate_rcvbuf`)

The term *auto-tuning* is used to denote a functionality in Linux in which the receive-side buffer is dynamically adjusted. As described in Section 4.3.2, the `tcp_rmem` variable has three fields: `min`, `default` and `max`, and that the initial size of the receive buffer is the `default` value. The growth of the receive buffer from the `default` value to the `max` value is enabled by the the auto-tuning feature.

The Linux variable `tcp_moderate_rcvbuf` controls whether-or-not auto-tuning should be executed. If this variable is set to 0, the TCP receive buffer size will be held unchanged at the default value of `tcp_rmem`, but if this variable is set to 1, the buffer size will be increased, if needed, to the `max` value of `tcp_rmem` provided memory is available.

To study the impact of the auto-tuning feature, an experiment was conducted. The `tcp_wmem` `default` and `max` values were set to $2 \times BDP$ to ensure that TCP send buffer size is not a limiting factor. The `tcp_rmem` `default` and `max` values were varied. HTCP was used as the congestion-control protocol, and `iperf3` was used to create a TCP flow for 5 seconds. The receive window values were obtained from the collected packet traces.

Table 4.4 shows that when `tcp_moderate_rcvbuf` was disabled, and `tcp_rmem` `default` was set to BDP (1250000 bytes), while the `max` was set to $2 \times BDP$ (2500000 bytes), the receive window (`rwnd`) grew from 28960 bytes up to half `tcp_rmem default` value (623552 bytes). As noted in Section 4.3.2, the initial value of the receive window (`rwnd`) is smaller than half the initial receive buffer size (assuming the initial receive buffer size was equal to the default value). We only have visibility into the receive window size (`rwnd`) (from the packet traces), but not into the receive buffer size. Table 4.4 shows that the receive window did not grow up to half `tcp_rmem max`. Since the auto-tuning feature was disabled, the receive buffer size did not grow to the `max` value, and consequently the receive window size `rwnd` reported in the packet headers was never larger than the maximum `rwnd` shown in Table 4.4. Since this maximum value was below BDP, the throughput was only 444 Mbps.

The second row of Table 4.4 shows a case in which the auto-tuning feature was still disabled, but the `tcp_rmem default` value was set to $2 \times BDP$ (2500000 bytes). In this case, the initial receive buffer was large enough to allow the receive window size to grow from its initial value of 28960 bytes up to half `tcp_rmem default` value (1248576 bytes), which was equal to the path BDP. Hence, close to 1 Gbps throughput was achieved.

The third row of Table 4.4 shows a case when the auto-tuning feature was enabled, and the `tcp_rmem default` value was only 1 BDP, and yet the receive window reached a maximum value of 1250048 bytes, which is half the maximum receive buffer size as set in `tcp_rmem max` value. Therefore, we conclude that disabling the auto-tuning feature prevents the receive buffer size from increasing to the `tcp_rmem max` value. However, if the `tcp_rmem default` value is set to $2 \times BDP$, then even if the auto-tuning feature is disabled, the receive buffer size is already large enough to allow the receive window size to be sufficient for TCP to realize the maximum possible throughput.

Table 4.4: `tcp_moderate_rcvbuf` impact on TCP receive window and throughput

`tcp_window_scaling`	`tcp_rmem` `default` (bytes)	`tcp_rmem` `max` (bytes)	initial `rwnd` (bytes)	maximum `rwnd` (bytes)	Throughput (Mbps)
0 (disabled)	1250000	2500000	28960	623552	444
0 (disabled)	2500000	2500000	28960	1248576	916
1 (enabled)	1250000	2500000	28960	1250048	910

4.4 Data-Link Layer

4.4.1 Background

4.4.1.1 Traffic-control (`tc`) utility

Linux traffic control (`tc`) supports multiple queueing disciplines. This chapter describes in-depth experiments with the Token-Bucket Filter (TBF) queuing discipline. TBF is a simple queuing discipline that is used to shape traffic to a specific rate [38]. The three basic parameters are `rate`, `limit`, and `burst`. The `rate` parameter represents the rate at which tokens are generated and held in the token bucket. The `limit` parameter represents the size of a buffer that holds packets while waiting to be transmitted by the NIC. The `burst` parameter represents the token bucket size, which determines the maximum number of packets that can be sent out back-to-back at the NIC rate.

4.4.1.2 High-resolution timers

In Linux systems, a periodic system timer (also called software clock) expires, and all system calls that set timers will be notified via signals. The software clock is set by a Linux system variable called HZ, which is a compile-time constant. For example, if HZ = 250, then system calls that set timeouts will be signaled every 4 ms [39].

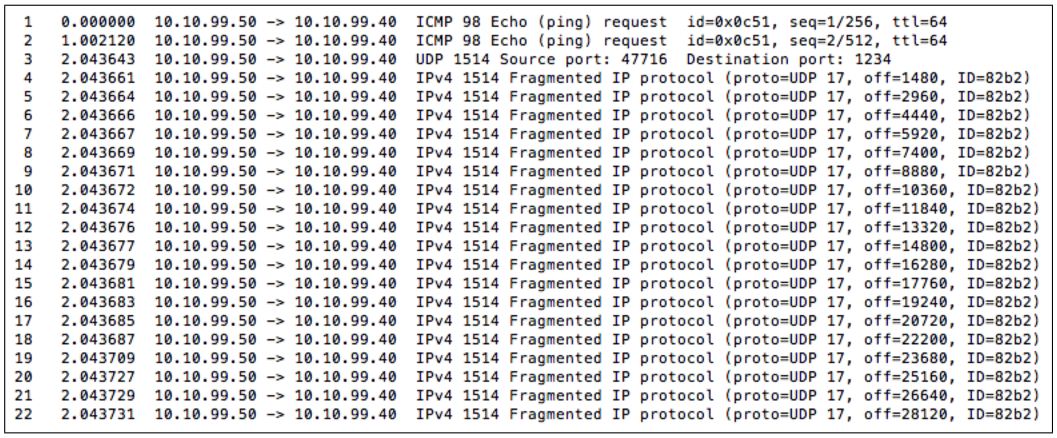
In Linux 2.6.21, high-resolution timers (HRTs) were introduced as a new timer subsystem. As noted on a Web site [39], "On a system that supports HRTs, the accuracy of sleep and timer system calls is no longer constrained by the software clock, but instead can be as accurate as the hardware allows (microsecond accuracy is typical of modern hardware)." To enable the HRT feature, the kernel should be compiled with the `CONFIG_HIGH_RES_TIMERS` option. With current systems, the hardware clock allows for timers with fine granularity, e.g., 1 ns. For example, if the periodic system timer, set by the HZ variable, is set to be 1 ms, without HRTs, a process cannot set events for time intervals smaller than 1 ms. But with the HRT subsystem, a process can register an event at time intervals of 1 ns.

4.4.2 Initial condition of the token bucket in the TBF queueing discipline

Objective: Documentation about the TBF queueing discipline [38] does not specify whether the token bucket is full at the start, or whether the token bucket fills up gradually with tokens being added at the specified TBF `rate`. The objective of this experiment is to make this determination. Our motivation for addressing this question is as follows. The TBF `rate` value could be smaller than the NIC rate. For example, a Layer-2 (L2) path could have been provisioned at the TBF `rate`, or there could be a bottleneck link on the end-to-end path whose rate is lower than the NIC rate. If the `burst` value is large, the transmitter could send out a large amount of data in back-to-back packets at the NIC rate. Such a burst could cause packets to be dropped at the router/switch buffer feeding the bottleneck link. Such behavior was observed in experimental results reported in our prior work [8].

Key finding: The results of the experiment confirmed that the token bucket is full of tokens at the start. Hence, at the beginning of a flow that is rate-shaped by `tc` using the TBF queueing discipline, there will be a packet burst (of size equals to the `burst` parameter) that is sent at the NIC rate. Packets will be sent out at the TBF `rate` only after the initial burst of packets.

Methodology: The key idea was to examine the differences in consecutive packet timestamps to determine whether packets were sent at the NIC rate or at the TBF `rate`. The overall method consists of the following steps: (i) A packet sniffer was used at the receiver to capture packets of the `tc` rate-shaped flow. (ii) A ping was performed for two seconds to ensure that the Address Resolution Protocol (ARP) table contains the necessary IP-address-to-MAC-address mapping. If the ARP request-response sequence was executed after the `tc` rate-shaped flow was initiated, we cannot determine whether-or-not the token bucket was full at the start because the token bucket



```
1    0.000000  10.10.99.50 -> 10.10.99.40  ICMP 98 Echo (ping) request  id=0x0c51, seq=1/256, ttl=64
2    1.002120  10.10.99.50 -> 10.10.99.40  ICMP 98 Echo (ping) request  id=0x0c51, seq=2/512, ttl=64
3    2.043643  10.10.99.50 -> 10.10.99.40  UDP 1514 Source port: 47716  Destination port: 1234
4    2.043661  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=1480, ID=82b2)
5    2.043664  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=2960, ID=82b2)
6    2.043666  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=4440, ID=82b2)
7    2.043667  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=5920, ID=82b2)
8    2.043669  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=7400, ID=82b2)
9    2.043671  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=8880, ID=82b2)
10   2.043672  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=10360, ID=82b2)
11   2.043674  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=11840, ID=82b2)
12   2.043676  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=13320, ID=82b2)
13   2.043677  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=14800, ID=82b2)
14   2.043679  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=16280, ID=82b2)
15   2.043681  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=17760, ID=82b2)
16   2.043683  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=19240, ID=82b2)
17   2.043685  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=20720, ID=82b2)
18   2.043687  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=22200, ID=82b2)
19   2.043709  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=23680, ID=82b2)
20   2.043727  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=25160, ID=82b2)
21   2.043729  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=26640, ID=82b2)
22   2.043731  10.10.99.50 -> 10.10.99.40  IPv4 1514 Fragmented IP protocol (proto=UDP 17, off=28120, ID=82b2)
```

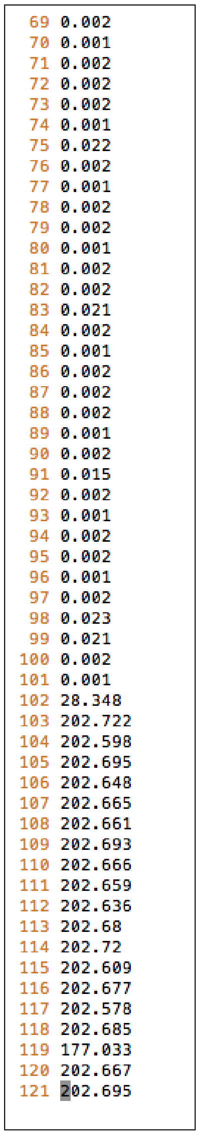
Figure 4.12: UDP packets captured at the receiver. The first two packets are the ping packets. The UDP packets start from packet number 3. Col. 1: packet number, col. 2: packet arrival time (relative), col. 3: source IP address, col. 4: destination IP address, col. 5: protocol number, `socat` message of size 57520 was fragmented into 39 packets starting from packet 3

would have time to fill up during the ARP request-response sequence. (iii) The `tc` command to configure the TBF queuing discipline was executed at the sender. (iv) An application was used to create a UDP flow immediately after issuing the command to run `tc`. UDP was used instead of TCP because TCP uses a 3-way handshake to establish a connection, which offers the token generator an opportunity to fill up an initially empty bucket. In other words, if the experimental `tc` rate-shaped flow was a TCP flow, the results would be inconclusive because the token bucket could have been empty at the start and then filled up during the TCP connection establishment phase, or the token bucket could have been full at the start. Therefore, a UDP flow was used instead of a TCP flow. (v) The packet sniffer at the receiver was terminated. (vi) A packet analyzer was used to analyze the packet trace and output the arrival time differences between consecutive packets. (vii) These time differences showed that a burst of packets were sent back-to-back at the NIC rate, and that the size of the burst matched the TBF `burst` parameter value.

Experiment setup and execution: The experiment was conducted using Setup 2. The application `socat` (version 1.7.2.3) was used to execute file transfers using UDP. The `tcpdump` tool was used at the receiver to capture UDP flow packets, and `tshark` version 1.6.7-1 was used to process the collected packet traces.

TBF parameters were set as follows: `limit`: 25 MB (large enough to ensure packets are not dropped at the sender by `tc`), `burst`: 151400 bytes (large enough to burst out 100 packets at the NIC rate), `rate`: 60 kbps. At this rate, it will take 20 seconds to fill the token bucket. Therefore, even if there is a millisecond gap between the execution time of the `tc` command to configure TBF and the time instant when the `socat` flow starts sending data, this time gap is not sufficient for the bucket to fill up. A shell script was used to run the following commands back-to-back: (i) ping, to update the ARP table, (ii) `tc`, to configure TBF, and (iii) `socat`, to send data.

Results and discussion: Figure 4.12 shows the first twenty received UDP packets, each of size 1514 bytes. The first two packets shown in the packet trace are ping packets, used to update the ARP table. The `socat` application was initiated with 57520 bytes, therefore, the IP layer fragmented the message into 1514-byte packets.



```
 69 0.002
 70 0.001
 71 0.002
 72 0.002
 73 0.002
 74 0.001
 75 0.022
 76 0.002
 77 0.001
 78 0.002
 79 0.002
 80 0.001
 81 0.002
 82 0.002
 83 0.021
 84 0.002
 85 0.001
 86 0.002
 87 0.002
 88 0.002
 89 0.001
 90 0.002
 91 0.015
 92 0.002
 93 0.001
 94 0.002
 95 0.002
 96 0.001
 97 0.002
 98 0.023
 99 0.021
100 0.002
101 0.001
102 28.348
103 202.722
104 202.598
105 202.695
106 202.648
107 202.665
108 202.661
109 202.693
110 202.666
111 202.659
112 202.636
113 202.68
114 202.72
115 202.609
116 202.677
117 202.578
118 202.685
119 177.033
120 202.667
121 202.695
```

Figure 4.13: Col .1: packet number; col. 2: $T_b - T_{b-1}$, where T_b is arrival time of packet b; unit: ms

Figure 4.13 shows the inter-arrival times between packets of the UDP flow. If the initial condition was an empty token bucket, since tokens were generated at 60 kbps, the inter-arrival times between packets would have been 200 ms (*1514 bytes/60 kbps = 200 ms*). On the other hand, if the initial condition was a full token bucket, a burst of 100 packets would arrive with inter-arrival times of 1.2 μs because of the 10 Gbps NIC rate (*1514 bytes/10 Gbps = 1.2 μs*). The results in Figure 4.13 confirm that the initial condition is a full token bucket. After the first 100 packets were transmitted (packet 3 to packet 102), the token bucket was empty. Starting from packet 103, packets were transmitted with inter-arrival times of 200 ms as seen in Figure 4.13.

Conclusions: Our experiment proved that in the TBF queueing discipline, the initial condition is a full token bucket. Therefore, users should be aware that when using UDP to transfer data, or when using TCP with a large initial congestion window (cwnd), as with CTCP [33], a packet burst whose size is determined by the `burst` parameter of TBF will be sent back-to-back at the NIC rate. If switches/routers on the end-to-end path have small buffers, packet loss is possible in this initial burst phase.

4.4.3 How TBF sends out packets

Objective: The objective of this experiment was to determine whether packets are sent in bursts based on the Linux system timer, or packets are sent at fixed intervals whose duration is determined by the TBF `rate`. The motivation was to design a method to select the TBF `burst` size. The TBF manual page [38] specifies that the minimum value for `burst` should be computed as follows: $\texttt{burst} = \texttt{rate} \times HZ$, where `rate` is the desired sending rate, and HZ defines the system timer as noted earlier. However, the impact of the `burst` value is reduced if the TBF code can be called at shorter time intervals.

Key findings: Packets are sent in bursts based on the system timer if the inter-packet transmit time is is less than the system timer. For example, to send data at a 20 Mbps rate, one maximum-sized Ethernet frame (1500B) should be sent every 0.6 ms. But if the system HZ value was set to 250, then the system timer is 4 ms, which means the TBF-code set timer can only be set to 4 ms. Further, the TBF `burst` should be at least 10500 bytes to achieve the 20 Mbps average sending rate, i.e., 10500 bytes are sent every 4 ms. But these bytes are sent at the full NIC rate, which means 7 packets will be sent within 8.4 μs, since each 1500B packet needs only 1.2 μs for transmission by a 10-Gbps NIC. The NIC will be idle for for (4000 - 8.4) = 3991.6 μs in ever 4 ms interval.

On the other hand, if the High-Resolution Timer (HRT) feature, described in Section 4.4.1.2, is enabled, then since 0.6 ms, which is 600 μs, is larger than 1 ns (assuming the hardware clock allows for this small timer value), the TBF code will send only a single packet when its 600 μs timer expires. In this case, packets will not be sent in bursts, and corresponding the TBF `burst` value does not play a significant role.

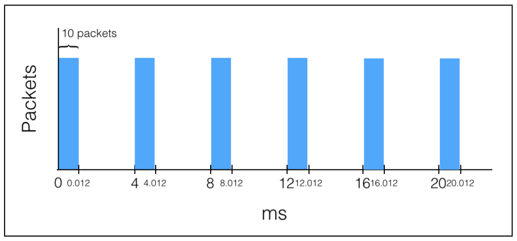
Methodology: The method consists of the following steps: (i) A packet sniffer was used at the receiver to capture packets of the `tc` rate-shaped flow. (ii) The `tc` command was executed at the sender to configure the TBF queuing discipline. (iv) An application was used to create a TCP flow. (v) The packet sniffer at the receiver was terminated. (vi) A packet analyzer was used to analyze the packet trace to determine packet inter-arrival times. (vii) Based on the packet inter-arrival times, we determined how packets are metered out on to the NIC in a `TBF`-controlled flow.

Experiment setup and execution: The experiment was conducted using Setup 2. The application `iperf3` was used to transfer data on a TCP connection. The `tcpdump` tool was used at the receiver to capture TCP flow packets, and `tshark` version 1.6.7 was used to process the collected packet trace.

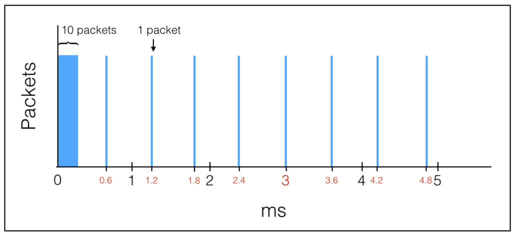
TBF parameters were set as follows: `limit`: 25 MB (large enough to ensure packets are not dropped at the sender by `tc`), `burst`: 15140 bytes (large enough to burst out 10 packets of size 1514 bytes at the NIC rate), `rate`: 20 Mbps. CTCP was used as the congestion-control module, and the fixed congestion window size was set to 45 packets. The RTT on the path was 26 ms, and therefore BDP was 520 KB, which is 43 maximum-sized packets, and hence the CTCP congestion window size was made slightly larger. The TBF `limit` was left unchanged at the large value used for a prior 4 Gbps experiment. A shell script was used to run the `tc` command to configure TBF at the sender, and then initiate `iperf3` to send data.

Results and discussion: Fig. 4.14 shows how TBF metered out packets under two different assumptions. Fig. 4.14a shows the behavior of TBF when the system timer was 4 ms and the HRT feature was not enabled. Since the burst size was set to 15140 bytes (10 packets), and the time needed by a 10-Gbps NIC to send these 10 packets is 12 μs, there is a silence period for the remaining (4000-12 = 3988) μs in each 4-ms interval as seen in Fig. 4.14a. More importantly, a burst of 10 packets was sent at the full NIC rate, which makes it more likely for switch buffers to overflow.

Fig. 4.14b shows TBF metered out packets when a high-resolution timer was available. In this case, the TBF code was signaled every 600 μs, and exactly one packet was transmitted each time. As packets are 1500B, one packet needs to be sent every 600 μs to achieve the 20 Mbps rate. Fig. 4.14b shows while packets are sent exactly one every 600 μs, in the beginning, we see 10 packets being sent out in a burst. This behavior was explained in Section 4.4.2 as being attributed to a full token



(a) System timer (software clock): 4 ms; HRT was unavailable



(b) HRT feature was available

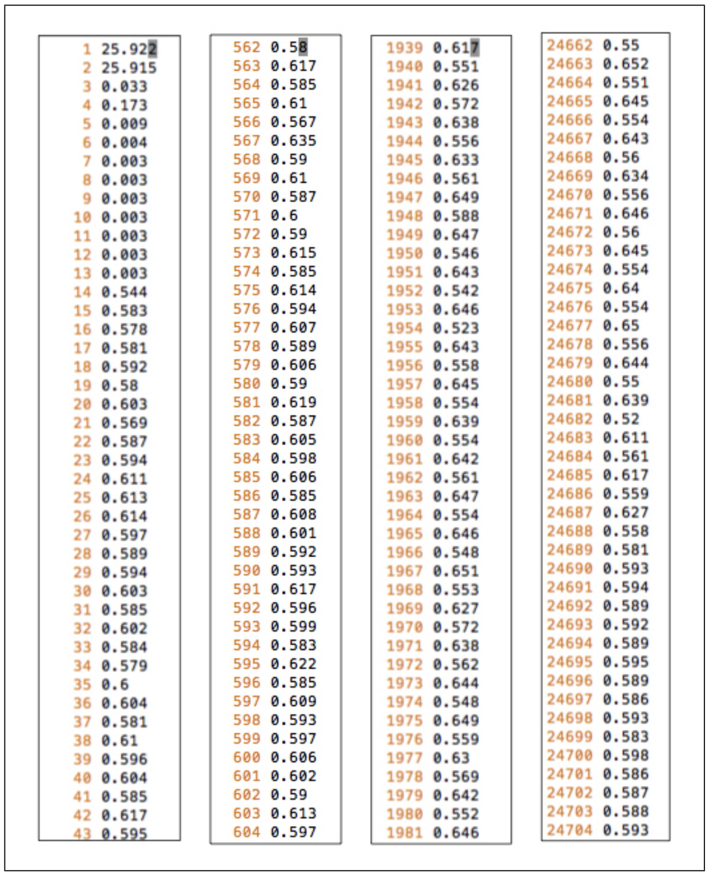
Figure 4.14: How `TBF` sends packets; `rate`=20 Mbps, `burst`=15140 bytes; `limit` = 25 MB

bucket at the start.

Fig. 4.15 shows multiple snapshots of the analyzed packet trace collected at a host with an HRT. The first 3 packets were control packets that do not carry data (SYN, SYN ACK, and ACK/PSH packet). Packet 4 was the first packet that carried user data. From packet 4 to packet 13, ten packets (their total size equals the TBF `burst` value) were sent back-to-back at the NIC rate ($1500 \times 8bit/10Gbps = 1.2\mu s$). Inter-packet arrival times were slightly more at 3 μs due to other delays on the path. Starting from packet 14, one packet was sent approximately every 0.6 ms, which is the time required to send a packet of size 1500 bytes while not exceeding TBF `rate` of 20 Mbps.

If the hardware clock allows a high-resolution timer of 1 ns, then the time needed by TBF to send one packet is greater than the HRT even if the TBF `rate` is 40 Gbps or 100 Gbps, since packet intervals should be 300 ns or 120 ns, respectively. Therefore, one packet will be sent out based on TBF `rate` even with high rates as long as the machine is capable of sending packets down to the `tc` layer, and the CPU is not loaded with many other processes that require soft interrupts. TBF uses a `watch_dog` to set up a timer, which then uses a soft interrupt to wake up TBF. If many other processes set soft interrupts, then the time taken by the CPU to handle each such interrupt would delay when the TBF code interrupt can be handled. Such delays could cause generating more tokens to fill the bucket, in which case multiple packets would be sent in a burst. Therefore, in multi-tasking systems, it is difficult to achieve deterministic behavior.

Conclusions: The specifics of how TBF meters out packets depends upon the system timer, TBF `rate` and TBF `burst` values. For the specified `rate`, if the inter-packet transmit time is less than the system timer and the HRT feature is not available, then a sufficiently large `burst` size is



```
 1 25.922
 2 25.915
 3 0.033
 4 0.173
 5 0.009
 6 0.004
 7 0.003
 8 0.003
 9 0.003
10 0.003
11 0.003
12 0.003
13 0.003
14 0.544
15 0.583
16 0.578
17 0.581
18 0.592
19 0.58
20 0.603
21 0.569
22 0.587
23 0.594
24 0.611
25 0.613
26 0.614
27 0.597
28 0.589
29 0.594
30 0.603
31 0.585
32 0.602
33 0.584
34 0.579
35 0.6
36 0.604
37 0.581
38 0.61
39 0.596
40 0.604
41 0.585
42 0.617
43 0.595
```

```
562 0.58
563 0.617
564 0.585
565 0.61
566 0.567
567 0.635
568 0.59
569 0.61
570 0.587
571 0.6
572 0.59
573 0.615
574 0.585
575 0.614
576 0.594
577 0.607
578 0.589
579 0.606
580 0.59
581 0.619
582 0.587
583 0.605
584 0.598
585 0.606
586 0.585
587 0.608
588 0.601
589 0.592
590 0.593
591 0.617
592 0.596
593 0.599
594 0.583
595 0.622
596 0.585
597 0.609
598 0.593
599 0.597
600 0.606
601 0.602
602 0.59
603 0.613
604 0.597
```

```
1939 0.617
1940 0.551
1941 0.626
1942 0.572
1943 0.638
1944 0.556
1945 0.633
1946 0.561
1947 0.649
1948 0.588
1949 0.647
1950 0.546
1951 0.643
1952 0.542
1953 0.646
1954 0.523
1955 0.643
1956 0.558
1957 0.645
1958 0.554
1959 0.639
1960 0.554
1961 0.642
1962 0.561
1963 0.647
1964 0.554
1965 0.646
1966 0.548
1967 0.651
1968 0.553
1969 0.627
1970 0.572
1971 0.638
1972 0.562
1973 0.644
1974 0.548
1975 0.649
1976 0.559
1977 0.63
1978 0.569
1979 0.642
1980 0.552
1981 0.646
```

```
24662 0.55
24663 0.652
24664 0.551
24665 0.645
24666 0.554
24667 0.643
24668 0.56
24669 0.634
24670 0.556
24671 0.646
24672 0.56
24673 0.645
24674 0.554
24675 0.64
24676 0.554
24677 0.65
24678 0.556
24679 0.644
24680 0.55
24681 0.639
24682 0.52
24683 0.611
24684 0.561
24685 0.617
24686 0.559
24687 0.627
24688 0.558
24689 0.581
24690 0.593
24691 0.594
24692 0.589
24693 0.592
24694 0.589
24695 0.595
24696 0.589
24697 0.586
24698 0.593
24699 0.583
24700 0.598
24701 0.586
24702 0.587
24703 0.588
24704 0.593
```

Figure 4.15: Four snapshots of the analyzed packet trace for the case with HRT. For each snapshot, col. 1: packet number; col. 2: $T_b - T_{b-1}$, where T_b is arrival time of packet b; unit: ms

required to achieve the specified `rate`, and packets will be sent in bursts at the NIC rate on every expiration of the system timer. On the other hand, if the time-interval required between packets to achieve the specified `rate` is greater than the high-resolution timer if available, (or system timer), then a single packet will be sent on every timer expiration, and packets will not appear at the NIC rate in bursts. The implication of this finding is that with HRT, probability of packet loss due to a switch buffer overflow will be smaller since packets are not sent in bursts.

4.4.4 TBF `limit` vs `txqueuelen`

The `txqueuelen` parameter represents the size of the queue in which packets are held for transmission by the NIC. The TBF `limit` is also a parameter that is used to determine the size of a buffer that holds packets waiting for transmission by the NIC. The objective of this experiment was to understand the relation between `txqueuelen` and TBF `limit`.

An article by Dan Simeon [40] states that "txqueuelen is only used as a default queue length for some of the queueing disciplines." The following queueing disciplines are listed: `pfifo_fast` (Linux default queueing discipline) `sch_fifo`, `sch_gred`, `sch_htb`, `sch_plug`, `sch_sfb`, and `sch_teq`. The same article goes on to state "For most of these queueing disciplines, the `limit` argument on the `tc` command line overrides the `txqueuelen` default. In summary, if you do not use one of the above queueing disciplines or if you override the queue length then the `txqueuelen` value is meaningless." As we recommended TBF for circuits/VCs in Chapter 3, and this queueing discipline is not listed in the set specified by the article, we conducted this experiment to determine whether TBF `limit` overrides `txqueuelen`.

The following method was used: (i) determine the value of TBF `limit` at which packets are dropped by `tc`, which can be obtained using `-statistics` argument when calling `tc`, for a particular data-transfer application, (ii) set the `txqueuelen` to the size determined from step 1, (iii) set TBF `limit` to a larger value than the size determined from step 1, (iii) run the same application as in step (i), (iv) obtain the number of dropped packets from `tc` statistics, (v) if `tc` did not drop packets, it implies that TBF `limit` overrides `txqueuelen`. However, if there are packet drops, then TBF `limit` does not override `txqueuelen`, and both `txqueuelen` and TBF `limit` should be tuned properly to ensure that no packets are dropped by `tc`.

Setup 3 was used to conduct the experiment. An artificial RTT of 10 ms was injected at the receiver using `netem`. TBF parameters were set as follows: `limit`: varied, `burst`: 10 KB, and `rate`: 1 Gbps. CTCP was used as the congestion-control protocol with `fcwnd` set to 1000 packets. The `txqueuelen` value at the sender was varied. The `iperf3` tool was used to transfer data for 10 seconds over a CTCP connection.

Table 4.5 shows the results of the experiment. It provides `txqueuelen` value in bytes, by multiplying the actual `txqueuelen` value, which is specified in Ethernet frames, by 1500B, to enable easier comparison with the `limit`, which is specified in bytes. We found that a `limit` value of 15 KB caused packet drops by `tc` as evidenced by the throughput drop to 352 and 349 Mbps in

Table 4.5: Impact of `txqueuelen` and TBF `limit` on the number of dropped packets at the sender

TBF `limit`	`txqueuelen`	number of packets dropped by `tc`	Throughput (Mbps)
1.5 MB	1.5 MB	0	917
1.5 MB	15 KB	0	917
15 KB	15 KB	44171	352
15 KB	1.5 MB	43412	349

the last two rows of Table 4.5. Furthermore, row 3 shows that if `txqueuelen` is dropped to 15 KB, there is no impact on throughput as long as `limit` is 1.5 MB. These results confirm that TBF `limit` overrides `txqueuelen` value. Therefore, only TBF `limit` value should be tuned properly to ensure no packets are dropped by `tc`.

4.5 Monitoring

Applications such as `iperf3` and `nuttcp` can create TCP flows and report the number of retransmissions of the created flow. However, If other applications such as GridFTP was used to transfer data and we are interested in finding the number of retransmissions, or we want to capture all flow packets for further analysis (e.g., inter-arrival time of flow packets), then packets of the flow should be collected and analyzed. The objective of this section is to provide insights on how to use `tcpdump`, `tshark`, and `tcptrace` to achieve theses goals.

4.5.1 Background

Large-Receive Offload (LRO) and TCP-Segmentation Offload (TSO) are techniques used to increase TCP flows throughput. The idea of the offload techniques is to merge multiple consecutive packets from the same flow into one large packet to reduce the processing overhead.

LRO technique is used specifically to merge incoming packets into a larger packet before passing it to the upper layer for processing. On the other hand, TSO is used to send a large packet (larger than the path MTU) to the NIC, which takes care of segmenting the large packet into smaller packets with a size that matches the path MTU.

Two other techniques were introduced later: Generic-Receive Offload (GRO) and Generic-Segmentation Offload (GSO). Both were introduced to overcome the LRO/TSO limitation to TCP over IPv4 and to solve the problem of merging packets in a way that does not preserve all the flow states. GRO/GSO are generic and can be used with other protocols such as UDP, and are less aggressive than LRO/TSO in combining packets, which preserve flow states.

4.5.2 On capturing all packets of a flow

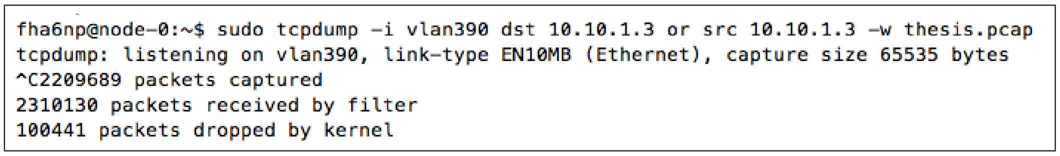
Objective: The objective of this experiment was to determine the settings required to capture all packets of a flow. For example, to determine the initial state of the token bucket, which was described in Section 4.4.2, we needed a trace consisting of all packets of a flow.

Key recommendations: Based on lessons learned from this set of experiments, we recommend the following: (i) when `tcpdump` is to used to capture packets, the standard output should be directed to a log file, which should then be parsed and analyzed automatically to check whether any packets were dropped by the kernel and consequently not included in the capture trace, (ii) careful thought should be given on whether to capture packets at the sender, at the receiver, or at both places, as some types of analyses require the packet trace from the sender, some others require the packet trace from the receiver, and finally some require both, and (iii) offload features as TSO/LRO and GSO/GRO should be disabled if information about the packets transmitted on the wire is required from packet traces, but disabling these features will reduce TCP throughput.

Experiment setup and execution: The experiment was conducted using Setup 3. The `iperf3` tool was used to send data for 30 seconds, while `tcpdump` was running and configured to collect all packets for that particular `iperf3` flow. Packet traces were collected at both the sender and the receiver. Experiment 1 demonstrates how to capture all flow packets without dropping any packets, while Experiment 2 shows the impact of the TSO/LRO and GSO/GRO features.

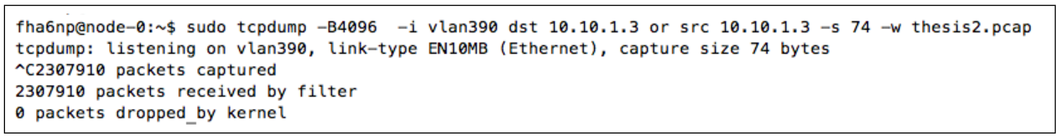
Results and discussions:

Experiment 1: Fig. 4.16 shows the execution of two `tcpdump` commands and their corresponding screen logs. Fig. 4.16a shows the case when the `tcpdump` was invoked without the `-B` and `-s` flags, and Fig. 4.16b shows the case when these flags were used. Fig. 4.16a shows that 100K



```
fha6np@node-0:~$ sudo tcpdump -i vlan390 dst 10.10.1.3 or src 10.10.1.3 -w thesis.pcap
tcpdump: listening on vlan390, link-type EN10MB (Ethernet), capture size 65535 bytes
^C2209689 packets captured
2310130 packets received by filter
100441 packets dropped by kernel
```

(a) Packets were dropped by the kernel before `tcpdump` collect them



```
fha6np@node-0:~$ sudo tcpdump -B4096  -i vlan390 dst 10.10.1.3 or src 10.10.1.3 -s 74 -w thesis2.pcap
tcpdump: listening on vlan390, link-type EN10MB (Ethernet), capture size 74 bytes
^C2307910 packets captured
2307910 packets received by filter
0 packets dropped_by kernel
```

(b) No packets were dropped by the kernel

Figure 4.16: `tcpdump` standard output

packets were dropped by the kernel, which means that 100K packets would be missing from the `tcpdump` packet trace collected in the run without the `-B` and `-s` flags. Analysis of a packet trace with missing packets could lead to inaccurate interpretations.

The `tcpdump` tool uses a buffer to hold all packets before applying filters. To avoid dropping packets, the size of the buffer should be increased, which can be done using the `-B` flag of `tcpdump`. Moreover, since only packet headers are required to understand the behavior of protocol layers such as TCP, the payload data can be dropped to reduce the size of the packet capture file. The `-s` flag of `tcpdump` can be used to specify the maximum size of saved bytes for each packet. We used 74 bytes, which is sufficient to capture all the required headers (14 bytes for the Ethernet header, 20 bytes for the IP header, and 40 bytes for the TCP header with 20 bytes for the mandatory fields and 20 bytes for the options field). Fig. 4.16b shows that when `-B` and `-s` flags were used when invoking `tcpdump`, no packets were dropped.

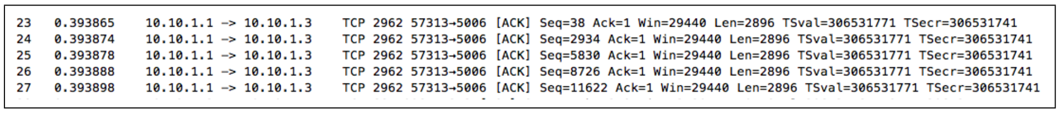
Experiment 2: Fig. 4.17 shows a part of sender-side packet traces for two cases: (i) when the offload features (LRO/GRO, TSO/GSO) were enabled (which is the default setting in Linux), and (ii) when the offload features were disabled using `ethtool`. Fig. 4.17a shows all packet sizes to be 2962 bytes, which is larger than the path MTU (1500 bytes). The packet trace shows large packet sizes because LRO/TSO were enabled when the packet trace was collected.

Fig. 4.17b shows the size of each packet as 1514 bytes (also shown is a `Len` field of 1448 bytes, which represents the TCP payload size as explained in Section 4.2.2). When the offload features are disabled, the size of packets handed to the Ethernet NIC is 1514 bytes. Additional bytes are added by the NIC for CRC and a physical-layer preamble before the packets are transmitted on to the wire. If information is needed about each packet that is transmitted on to the wire, e.g., packet timestamps for the exact transmission instants, then the offload features should be disabled.

A second, but complementary, point is that careful consideration should be given as to whether packets should be captured at the sender or the receiver. This is because a copy of the packet is passed to sniffers such as `tcpdump` before the `hard_start_xmit` function is called for packet transmission [41]; in other words, if `tcpdump` is run at the sender, packets are captured before they are transmitted on to the wire. Therefore, if an analysis requires consideration of actual (or relative) time instants when packets were transmitted on to the wire, it is better to capture and analyze packet traces collected at the receiver. For ordinary NICs in which a received packet is delivered to the higher layers via at the time of arrival through interrupts, the receiver-captured time instants offer a better of inter-packet transmission times than do transmitted-captured time instants. A word of caution that this finding holds only if the path does not include many switches with interfering traffic because on such paths, inter-arrival times at the receiver will be influenced by queueing delays at switches, and hence may not reflect transmission times at the sender. Packet capture at both the sender and receiver is required if dispersion between consecutive packets is of interest [42].

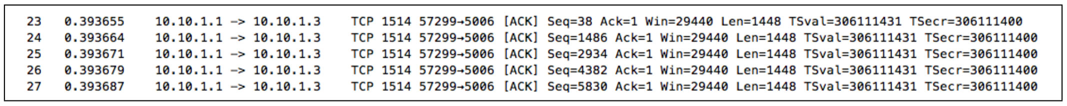
For other types of analyses, packets should be captured at the sender. For example, for determining packet retransmission rates, as described in Section 4.2.2, packets need to be captured at the sender, because it is the sending TCP that knows when a packet is retransmitted. If a packet was dropped by a network router, a receiver would have no record of the original packet transmission and hence could not determine from a trace whether a retransmission occurred or not (If the analyzing tool depends only on the occurrence of a duplicate packet to classify it as a retransmitted packet, e.g., `tcptrace`).

While offload features, LRO/TSO and GRO/GSO, should be disabled while debugging, disabling them will have an adverse impact on the achieved throughput and the packet capture trace size. Fig. 4.18 shows the impact of enabling and disabling the offload features. As seen in Figs. 4.18b and 4.18c, the flow throughput dropped when TSO/GSO or LRO/GRO were disabled because of the overhead of processing small packets. Furthermore, disabling the offload features



```
23   0.393865    10.10.1.1 -> 10.10.1.3    TCP 2962 57313→5006 [ACK] Seq=38 Ack=1 Win=29440 Len=2896 TSval=306531771 TSecr=306531741
24   0.393874    10.10.1.1 -> 10.10.1.3    TCP 2962 57313→5006 [ACK] Seq=2934 Ack=1 Win=29440 Len=2896 TSval=306531771 TSecr=306531741
25   0.393878    10.10.1.1 -> 10.10.1.3    TCP 2962 57313→5006 [ACK] Seq=5830 Ack=1 Win=29440 Len=2896 TSval=306531771 TSecr=306531741
26   0.393888    10.10.1.1 -> 10.10.1.3    TCP 2962 57313→5006 [ACK] Seq=8726 Ack=1 Win=29440 Len=2896 TSval=306531771 TSecr=306531741
27   0.393898    10.10.1.1 -> 10.10.1.3    TCP 2962 57313→5006 [ACK] Seq=11622 Ack=1 Win=29440 Len=2896 TSval=306531771 TSecr=306531741
```

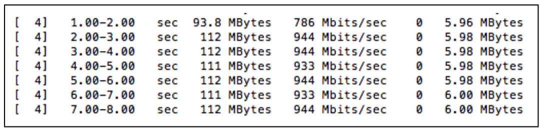
(a) Offload features were on



```
 23   0.393655    10.10.1.1 -> 10.10.1.3    TCP 1514 57299→5006 [ACK] Seq=38 Ack=1 Win=29440 Len=1448 TSval=306111431 TSecr=306111400
 24   0.393664    10.10.1.1 -> 10.10.1.3    TCP 1514 57299→5006 [ACK] Seq=1486 Ack=1 Win=29440 Len=1448 TSval=306111431 TSecr=306111400
 25   0.393671    10.10.1.1 -> 10.10.1.3    TCP 1514 57299→5006 [ACK] Seq=2934 Ack=1 Win=29440 Len=1448 TSval=306111431 TSecr=306111400
 26   0.393679    10.10.1.1 -> 10.10.1.3    TCP 1514 57299→5006 [ACK] Seq=4382 Ack=1 Win=29440 Len=1448 TSval=306111431 TSecr=306111400
 27   0.393687    10.10.1.1 -> 10.10.1.3    TCP 1514 57299→5006 [ACK] Seq=5830 Ack=1 Win=29440 Len=1448 TSval=306111431 TSecr=306111400
```

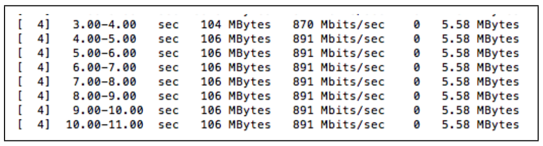
(b) Offload features were off

Figure 4.17: Offload feature impact of the size of captured packets



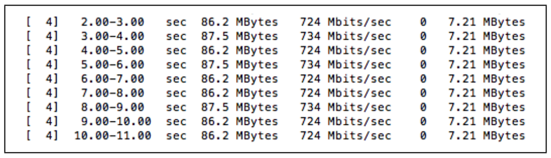
```
[  4]   1.00-2.00   sec  93.8 MBytes   786 Mbits/sec    0    5.96 MBytes
[  4]   2.00-3.00   sec   112 MBytes   944 Mbits/sec    0    5.98 MBytes
[  4]   3.00-4.00   sec   112 MBytes   944 Mbits/sec    0    5.98 MBytes
[  4]   4.00-5.00   sec   111 MBytes   933 Mbits/sec    0    5.98 MBytes
[  4]   5.00-6.00   sec   112 MBytes   944 Mbits/sec    0    5.98 MBytes
[  4]   6.00-7.00   sec   111 MBytes   933 Mbits/sec    0    6.00 MBytes
[  4]   7.00-8.00   sec   112 MBytes   944 Mbits/sec    0    6.00 MBytes
```

(a) Offload features were on



```
[  4]   3.00-4.00   sec   104 MBytes   870 Mbits/sec    0    5.58 MBytes
[  4]   4.00-5.00   sec   106 MBytes   891 Mbits/sec    0    5.58 MBytes
[  4]   5.00-6.00   sec   106 MBytes   891 Mbits/sec    0    5.58 MBytes
[  4]   6.00-7.00   sec   106 MBytes   891 Mbits/sec    0    5.58 MBytes
[  4]   7.00-8.00   sec   106 MBytes   891 Mbits/sec    0    5.58 MBytes
[  4]   8.00-9.00   sec   106 MBytes   891 Mbits/sec    0    5.58 MBytes
[  4]   9.00-10.00  sec   106 MBytes   891 Mbits/sec    0    5.58 MBytes
[  4]  10.00-11.00  sec   106 MBytes   891 Mbits/sec    0    5.58 MBytes
```

(b) TSO and GSO were off at the sender



```
[  4]   2.00-3.00   sec  86.2 MBytes   724 Mbits/sec    0    7.21 MBytes
[  4]   3.00-4.00   sec  87.5 MBytes   734 Mbits/sec    0    7.21 MBytes
[  4]   4.00-5.00   sec  86.2 MBytes   724 Mbits/sec    0    7.21 MBytes
[  4]   5.00-6.00   sec  87.5 MBytes   734 Mbits/sec    0    7.21 MBytes
[  4]   6.00-7.00   sec  86.2 MBytes   724 Mbits/sec    0    7.21 MBytes
[  4]   7.00-8.00   sec  86.2 MBytes   724 Mbits/sec    0    7.21 MBytes
[  4]   8.00-9.00   sec  87.5 MBytes   734 Mbits/sec    0    7.21 MBytes
[  4]   9.00-10.00  sec  86.2 MBytes   724 Mbits/sec    0    7.21 MBytes
[  4]  10.00-11.00  sec  86.2 MBytes   724 Mbits/sec    0    7.21 MBytes
```

(c) LRO and GRO were off at the receiver

Figure 4.18: The impact of offload features on the achieved throughput

increases the captured packet trace size. In this experiment, when the offload features were enabled, the packet trace size was only 17.6 MB, but when the offload features were disabled, the size of the packet trace increased to 206 MB.

Conclusions: We have three findings with regards to capturing all packets of flow to understand the detailed behavior of the flow. *First*, the `tcpdump` tool should be used with `-B` and `-s` flags, and the `tcpdump` log should be checked to ensure that the kernel did not drop any packets, and correspondingly the collected trace contains information about all packets of the flow. *Second*, the TSO/LRO and GSO/GRO features should be disabled when collecting packets for detailed analysis. However, these offload features should be enabled for the actual data transfers especially for high-throughput flows. *Third*, consideration should be given on where to collect packet traces. For some types of analyses, the sender is the better choice, while for others, packets should be captured at the receiver.

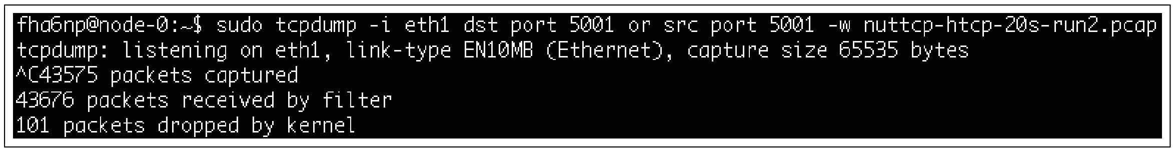
4.5.3 Reconciling differences in `tshark` and `tcptrace` output

Objective: The objectives of the experiment are to understand the differences in some of the numbers reported by `tshark` and `tcptrace`, and to determine how to estimate the number of retransmissions from the results reported by these analyses tools. For example, in Section 4.2.1, Figs 4.6 and 4.9 show a discrepancy between the number of out-of-order packets and number of retransmitted packets reported by these two analysis tools. Three types of traces were analyzed: (i) a trace with missing packets, (ii) a trace with all packets of a flow that was captured at the sender and (iii) a trace with all packets of a flow that was captured at the receiver. For high-speed flows, it may be difficult to capture all packets of a flow, which explains our consideration of the first trace.

Key recommendations: (i) If all that a user has is an incomplete packet trace collected at the sender, the sum of the number of retransmissions and the number of out-of-order packets reported by either `tcptrace` or `tshark` can be used as an estimate of the total number of retransmissions with the caveat that the percentage of packets missing in the trace should be relatively small. Incomplete packet traces collected at the receiver are not suitable for estimating the number of retransmissions. (ii) With a complete trace collected at the sender, the total number of retransmissions can be determined with both `tcptrace` and `tshark`; the former is faster but offers only aggregate information for the whole trace, while the latter can offer information on parts of the trace. (iii) Receive-side traces should not be used to determine number of retransmissions.

Experiment setup and execution: The experiment was conducted using Setup 5. The application `iperf3` (version 3.1) was used to transfer data via a TCP connection. The `tcpdump` tool was used at the sender and at the receiver to capture the TCP-flow packets, and `tshark` version 1.10.6 and `tcptrace` version 6.6.7 were used to analyze the collected packet traces. Since `iperf3` reported number of retransmissions were verified by experiments described in Section 4.2.1, here we use the `iperf3` number as the ground-truth to validate the `tcpdump` and `tshark` reported numbers.

Case 1: Trace with missing packets: The number of retransmitted packets reported by `iperf3` was 92 packets. Fig. 4.19 shows the `tcpdump` standard output log, which indicates that 92 packets were dropped by the kernel. On this run, neither the `-B` nor `-s` flags were used to deliberately create a packet trace with missing packets. The packet trace was analyzed using both `tshark` and `tcptrace`, and the results are reported in Figs. 4.20b and 4.20a, respectively. Fig. 4.20a shows that 20 packets were retransmitted and 72 packets were out-of-order. These two numbers add up to 101, matching the number reported by `iperf3`.



```
fha6np@node-0:~$ sudo tcpdump -i eth1 dst port 5001 or src port 5001 -w nuttcp-htcp-20s-run2.pcap
tcpdump: listening on eth1, link-type EN10MB (Ethernet), capture size 65535 bytes
^C43575 packets captured
43676 packets received by filter
101 packets dropped by kernel
```

Figure 4.19: `tcpdump` standard output

We first provide an explanation for the packets reported as out-of-order by `tcptrace` and then compare the (Column 4) numbers with `tshark` numbers. Consider an example flow in which a sender transmits the following sequence of packets 1, 2, 3, 4, 5 and 6, and `tcpdump` does not capture packet 3 and 5 because the buffer holding the packets to be processed by `tcpdump` was full. Now assume that packet 3 was retransmitted, the sequence of the packets in the packet trace would be: 1, 2, 4, 5, 3. If `tcptrace` captures packet 3, it reports the packet as an out-of-order packet. But since packets captured at the sender cannot be transmitted out-of-order (in a serial implementation of the Linux network stack), we can assume that out-of-order packets are actually retransmitted packets, and that the kernel dropped the first transmission. Had `tcpdump` captured all packets, then the packet trace would consist of the following sequence: 1, 2, 3, 4, 5, 6, 3, in which case `tcptrace` would classify the second packet 3 as a retransmitted packet. This analysis points to the conclusion that if the packet drop rate by the capture tool is small, the probability of both the original transmission and a retransmission being dropped is correspondingly small, and therefore, a user could use this estimate of the sum of the numbers of out-of-order packets and retransmitted packets as reported by `tcptrace` as a fairly accurate estimate of the number of retransmitted packets. Clearly, it is preferable to run a capture tool that does not drop packets.

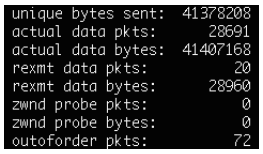
Next, we compare the output of `tshark` with that of `tcptrace`. Fig. 4.20b shows the number of out-of-order packets (Column 4) is 9, while the number of retransmitted packets (Column 0) is 83, which together also sum to 92 packets, which matches the number reported by `iperf3`.

However, the numbers reported for out-of-order packets by `tshark` and `tcptrace` do not match. An explanation for this finding is as follows. The `tshark` tool performs advanced analysis on a packet trace, which allows it to identify a retransmitted packet even if the packet did not appear twice in the packet trace. Consider the same previous example where `tcpdump` did not capture packets 3 and 5, and packet 3 was retransmitted. Now assume the receiver sent triple duplicate ACKs for packet 3. When `tshark` sees the triple duplicate ACKs, it concludes that packet 3 was sent previously even if did not appear in the packet trace because `tcpdump` was not able to capture it. Therefore, when packet 3 appears after packet 6, `tshark` considers it as a retransmitted packet instead of an out-of-order packet. In spite of the advanced analysis, `tshark` still classified 9 packets as out-of-order instead of retransmissions. One possible scenario for this misidentification is if one of the triple duplicate ACKs for packet 3 were not captured by `tcpdump`, in which case `tshark` would classify packet 3 as an out-of-order packet.

Consider the number of lost segments reported by `tshark`. Fig. 4.20b shows this number to be 7 (Column 2). This measure is a report of the number of missing blocks of packets. For example, if the sender transmitted the following packets: 1, 2, 3, 4, 5, 6, 7, 8, and the kernel dropped packets 3, 4 and 5, the packet trace would contain the following packets: 1, 2, 6, 7, 8. When `tshark` sees packet 2, it expects the following packet to be 3. However, since the next packet is 6, `tshark` marks packet 6 with a flag indicating that there was a burst of dropped packets without specifying exactly how many packets were dropped.

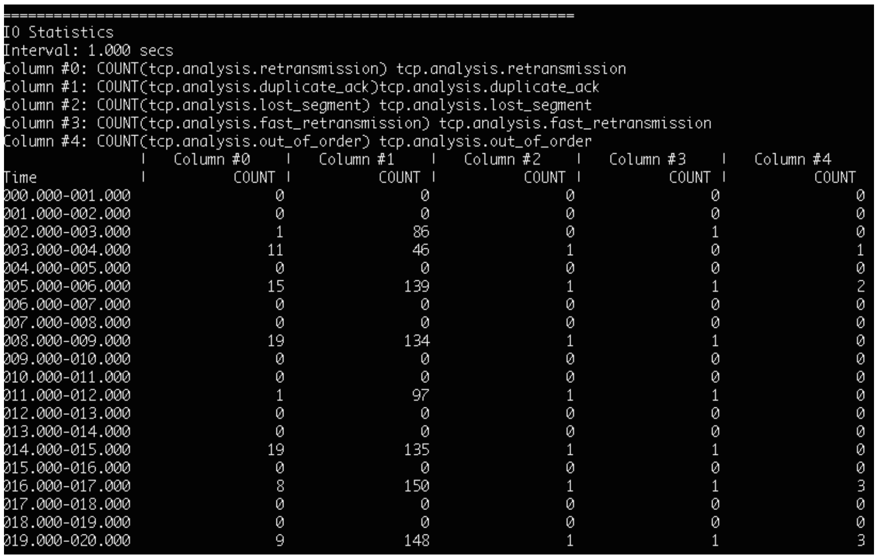
We do not recommend this probabilistic estimation of number of retransmissions from an incomplete traced collected at the receiver, because there are two problems. First, a receiver trace will not include retransmissions for packets that were dropped in router buffers on their transfer from the sender to the receiver, which makes it challenging for an analysis tool like `tcptrace` or `tshark` to accurately determine the number of retransmissions. Second, packets can arrive out-of-order in datagram networks such as IP-routed networks, and therefore the number of out-of-order packets reported by these tools could be genuine out-of-order packets and not retransmissions. Therefore, packet traces collected at receivers should not be used to determine the number of retransmissions.

Case 2: Sender-side trace with all packets of a flow: In this experiment, the `iperf3` flow was



```
unique bytes sent:  41378208
actual data pkts:      28691
actual data bytes:  41407168
rexmt data pkts:          20
rexmt data bytes:      28960
zwnd probe pkts:           0
zwnd probe bytes:          0
outoforder pkts:          72
```

(a) `tcptrace` output



```
====================================================================
IO Statistics
Interval: 1.000 secs
Column #0: COUNT(tcp.analysis.retransmission) tcp.analysis.retransmission
Column #1: COUNT(tcp.analysis.duplicate_ack)tcp.analysis.duplicate_ack
Column #2: COUNT(tcp.analysis.lost_segment) tcp.analysis.lost_segment
Column #3: COUNT(tcp.analysis.fast_retransmission) tcp.analysis.fast_retransmission
Column #4: COUNT(tcp.analysis.out_of_order) tcp.analysis.out_of_order
                |    Column #0    |    Column #1    |    Column #2    |    Column #3    |    Column #4
Time            |           COUNT |           COUNT |           COUNT |           COUNT |           COUNT
000.000-001.000                 0                 0                 0                 0                 0
001.000-002.000                 0                 0                 0                 0                 0
002.000-003.000                 1                86                 0                 1                 0
003.000-004.000                11                46                 1                 0                 1
004.000-005.000                 0                 0                 0                 0                 0
005.000-006.000                15               139                 1                 1                 2
006.000-007.000                 0                 0                 0                 0                 0
007.000-008.000                 0                 0                 0                 0                 0
008.000-009.000                19               134                 1                 1                 0
009.000-010.000                 0                 0                 0                 0                 0
010.000-011.000                 0                 0                 0                 0                 0
011.000-012.000                 1                97                 1                 1                 0
012.000-013.000                 0                 0                 0                 0                 0
013.000-014.000                 0                 0                 0                 0                 0
014.000-015.000                19               135                 1                 1                 0
015.000-016.000                 0                 0                 0                 0                 0
016.000-017.000                 8               150                 1                 1                 3
017.000-018.000                 0                 0                 0                 0                 0
018.000-019.000                 0                 0                 0                 0                 0
019.000-020.000                 9               148                 1                 1                 3
```

(b) `tshark` output

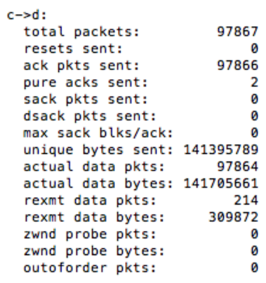
Figure 4.20: Case1: Analysis of a trace with missing packets

routed through a Linux host in the middle of the path between the source and destination as described in Setup 5 (see Section 4.1). The `netem` utility was used to randomly reorder packets of the flow to deliberately cause out-of-order packet arrivals at the receiver. For this case, in which the packet trace was collected at the sender, there should be no impact from the deliberate re-ordering of packets. But a packet trace was also collected at the receiver in this experiment for the Case 3 analysis, which will be presented next. The number of retransmissions reported by `iperf3` was 214 packets.

tcptrace: Fig. 4.21a shows that the number of retransmissions reported by `tcptrace` is also 214 packets. Since this trace contained all packets of the flow, if a packet was retransmitted, it would appear twice in the collected packet trace. This allowed `tcptrace` to easily count duplicate packets and report the number of retransmissions.

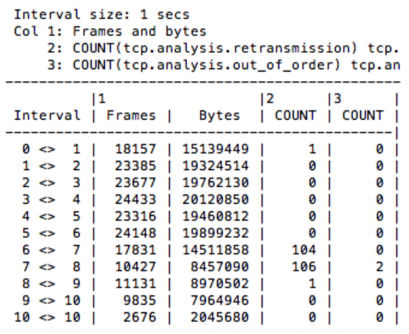
tshark: Fig. 4.21b shows the number of retransmitted packets (Column 2) and number of out-of-order packets (Column 3) reported by `tshark`. These numbers add up to 214 packets. A bug was reported that retransmitted packets were counted as out-of-order packets in the `tshark` tool versions before 1.12.x. Developers of `tshark` fixed this bug by considering the initial RTT value. However, we tested `tshark` version 1.12.10, but found there were still some packets reported as out-of-order packets instead of being classified as retransmitted packets.

Fig. 4.22 shows another packet trace collected at the sender, where `iperf3` reported 149 retransmissions. Two versions of `tshark` were used to analyze the same packet trace. In the older version (1.10.6), all retransmitted packets were reported as out-of-order packets, while in the newer version (1.12.10), 10 packets were reported as retransmissions, but all the other retransmissions (139 packets) were reported as out-of-order packets. Therefore, when using `tshark` reports of a sender-collected trace to find the total number of retransmissions, out-of-order packets should be added to the number of retransmitted packets regardless of the `tshark` version used.



```
c->d:
  total packets:          97867
  resets sent:                0
  ack pkts sent:          97866
  pure acks sent:             2
  sack pkts sent:             0
  dsack pkts sent:            0
  max sack blks/ack:          0
  unique bytes sent: 141395789
  actual data pkts:       97864
  actual data bytes: 141705661
  rexmt data pkts:          214
  rexmt data bytes:      309872
  zwnd probe pkts:            0
  zwnd probe bytes:           0
  outoforder pkts:            0
```

(a) `tcptrace` output



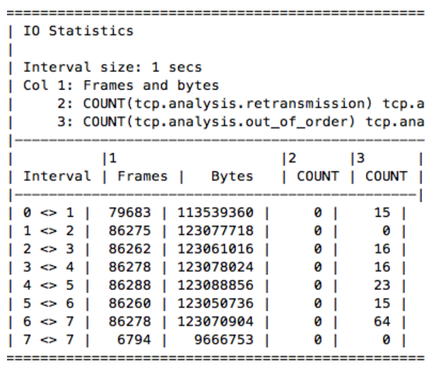
```
Interval size: 1 secs
Col 1: Frames and bytes
    2: COUNT(tcp.analysis.retransmission) tcp.
    3: COUNT(tcp.analysis.out_of_order) tcp.an
-----------------------------------------------
          |1                   |2       |3       |
Interval  | Frames |    Bytes  | COUNT  | COUNT  |
-----------------------------------------------|
 0 <>  1  |  18157 | 15139449  |      1 |      0 |
 1 <>  2  |  23385 | 19324514  |      0 |      0 |
 2 <>  3  |  23677 | 19762130  |      0 |      0 |
 3 <>  4  |  24433 | 20120850  |      0 |      0 |
 4 <>  5  |  23316 | 19460812  |      0 |      0 |
 5 <>  6  |  24148 | 19899232  |      0 |      0 |
 6 <>  7  |  17831 | 14511858  |    104 |      0 |
 7 <>  8  |  10427 |  8457090  |    106 |      2 |
 8 <>  9  |  11131 |  8970502  |      1 |      0 |
 9 <> 10  |   9835 |  7964946  |      0 |      0 |
10 <> 10  |   2676 |  2045680  |      0 |      0 |
```

(b) `tshark` output

Figure 4.21: Case 2: Analysis of a sender-side trace that contained all packets of a flow

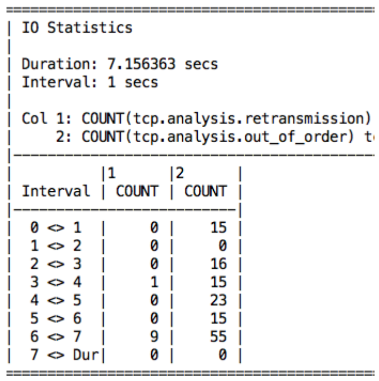
Case 3: Receiver-side trace with all packets of a flow: We analyzed the packet trace collected at the receiver during the experiment described under Case 2. Since an analysis of a complete sender-side trace can provide an accurate value for the number of retransmissions (which was 214 packets for this experiment), this number can be used as ground-truth in the receive-side trace analysis.

Fig. 4.23 shows that both `tshark` and `tcptrace` reported an incorrect number of retransmitted packets. As expected, the receive-side trace will not have information about packets that were dropped from the sender to the receiver, and hence any analysis of a receive-side trace cannot



```
=================================================
| IO Statistics
|
| Interval size: 1 secs
| Col 1: Frames and bytes
|     2: COUNT(tcp.analysis.retransmission) tcp.a
|     3: COUNT(tcp.analysis.out_of_order) tcp.ana
|------------------------------------------------
|          |1                   |2      |3      |
| Interval | Frames |    Bytes  | COUNT | COUNT |
|-----------------------------------------------|
| 0 <> 1 |  79683 | 113539360 |     0 |    15 |
| 1 <> 2 |  86275 | 123077718 |     0 |     0 |
| 2 <> 3 |  86262 | 123061016 |     0 |    16 |
| 3 <> 4 |  86278 | 123078024 |     0 |    16 |
| 4 <> 5 |  86288 | 123088856 |     0 |    23 |
| 5 <> 6 |  86260 | 123050736 |     0 |    15 |
| 6 <> 7 |  86278 | 123070904 |     0 |    64 |
| 7 <> 7 |   6794 |   9666753 |     0 |     0 |
=================================================
```

(a) `tshark` version 1.10.6



```
============================================
| IO Statistics
|
| Duration: 7.156363 secs
| Interval: 1 secs
|
| Col 1: COUNT(tcp.analysis.retransmission)
|     2: COUNT(tcp.analysis.out_of_order) t
|-------------------------------------------
|          |1      |2      |
| Interval | COUNT | COUNT |
|--------------------------|
|  0 <> 1  |     0 |    15 |
|  1 <> 2  |     0 |     0 |
|  2 <> 3  |     0 |    16 |
|  3 <> 4  |     1 |    15 |
|  4 <> 5  |     0 |    23 |
|  5 <> 6  |     0 |    15 |
|  6 <> 7  |     9 |    55 |
|  7 <> Dur|     0 |     0 |
============================================
```

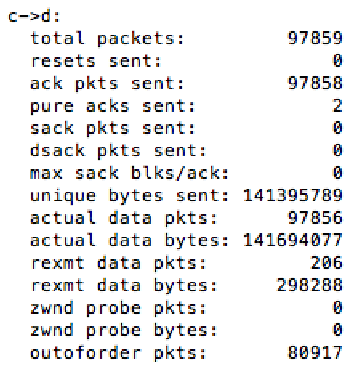
(b) `tshark` version 1.12.10

Figure 4.22: Case 2: A comparison of reports from two `tshark` versions

determine the number of retransmitted packets accurately. Furthermore, some of the out-of-order packets received are due to retransmissions and some are due to packets taking different routes through the network. Therefore, neither number provided by an analysis tool can be interpreted correctly.

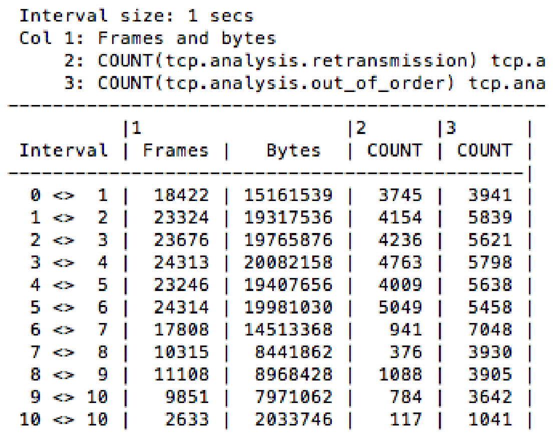
Fig. 4.24 shows analysis results for another packet trace that was collected at the receiver (using Setup 3). This trace corresponds to an `iperf3` flow that was executed from a sender and receiver that were connected via a single switch, which means there is only one path between the sender and the receiver, and hence packets cannot arrive out-of-sequence. At the sender, `netem` was used to inject packet losses. The number of retransmitted packets reported by `iperf3` was 529 packets. The `tcptrace` tool considered all retransmissions as out-of-order packets in this case, probably because the original packets did not appear in the packet trace. On the other hand, `tshark` classified some packets as retransmitted packets and others as out-of-order packets (`tshark` can guess that a packet was retransmitted even if the original packet did not appeared in the trace by observing triple duplicate ACKs). Therefore, in this case where no real out-of-order packets can arrive at the receiver, the number of out-of-order packets reported by `tcptrace` can be interpreted as the number of retransmissions, and the summation of numbers of retransmitted and out-of-order packets reported by `tshark` could be intrepreted in the same manner. It is possible, though unlikely, for retransmissions to arrive in sequence, and hence the estimate of the number of retransmissions, is a lower bound.

In all the three cases, we used both `tshark` and `tcptrace` to analyze the packet traces. As seen in the figures, `tshark` can report per-second statistics, while `tcptrace` can only report aggregate numbers. On the other hand, `tcptrace` is faster than `tshark`. For example, to analyze a packet trace of size 287 MB, `tcptrace` took only 1.4 sec, while `tshark` took 39 sec.



```
c->d:
  total packets:          97859
  resets sent:                0
  ack pkts sent:          97858
  pure acks sent:             2
  sack pkts sent:             0
  dsack pkts sent:            0
  max sack blks/ack:          0
  unique bytes sent: 141395789
  actual data pkts:       97856
  actual data bytes: 141694077
  rexmt data pkts:          206
  rexmt data bytes:      298288
  zwnd probe pkts:            0
  zwnd probe bytes:           0
  outoforder pkts:        80917
```

(a) `tcptrace` output

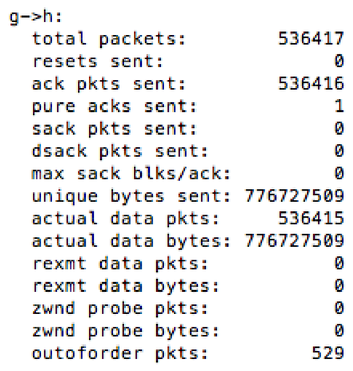


```
 Interval size: 1 secs
 Col 1: Frames and bytes
     2: COUNT(tcp.analysis.retransmission) tcp.a
     3: COUNT(tcp.analysis.out_of_order) tcp.ana
------------------------------------------------
          |1                  |2       |3       |
 Interval | Frames |    Bytes  | COUNT  | COUNT  |
-----------------------------------------------|
  0 <>  1 |  18422 | 15161539 |  3745 |  3941 |
  1 <>  2 |  23324 | 19317536 |  4154 |  5839 |
  2 <>  3 |  23676 | 19765876 |  4236 |  5621 |
  3 <>  4 |  24313 | 20082158 |  4763 |  5798 |
  4 <>  5 |  23246 | 19407656 |  4009 |  5638 |
  5 <>  6 |  24314 | 19981030 |  5049 |  5458 |
  6 <>  7 |  17808 | 14513368 |   941 |  7048 |
  7 <>  8 |  10315 |  8441862 |   376 |  3930 |
  8 <>  9 |  11108 |  8968428 |  1088 |  3905 |
  9 <> 10 |   9851 |  7971062 |   784 |  3642 |
 10 <> 10 |   2633 |  2033746 |   117 |  1041 |
```

(b) `tshark` output

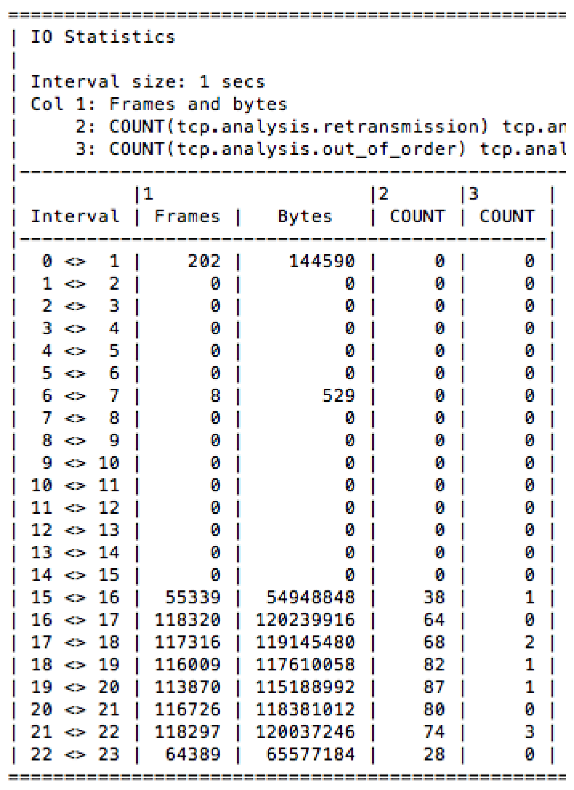
Figure 4.23: Case 3: Analysis of the receive-side trace from the same experiment as in Case 2; the trace that contained all packets of the flow

Conclusions: The number of retransmitted packets of a TCP flow can be determined by analyzing the packet trace collected at the sender. If the collected packet trace has some missing packets, the reported numbers of retransmitted packets and out-of-order packets should be added to determine the total retransmissions when using `tcptrace` or `tshark`. If the packet trace is not missing any packets, the reported number of retransmitted packets can be obtained directly from a `tcptrace` analysis, or through a summation of the number of retransmitted and out-of-order packets from a `tshark` analysis. Packet traces collected at the receiver should not be used to determine the number of retransmissions. The `tcptrace` is faster but provides only aggregate analysis, while `tshark` requires more time but can provide detailed analysis information, e.g., reports on retransmissions, out-of-order packets, etc. for 1-sec clips of the trace.



```
g->h:
  total packets:        536417
  resets sent:               0
  ack pkts sent:        536416
  pure acks sent:            1
  sack pkts sent:            0
  dsack pkts sent:           0
  max sack blks/ack:         0
  unique bytes sent: 776727509
  actual data pkts:     536415
  actual data bytes: 776727509
  rexmt data pkts:           0
  rexmt data bytes:          0
  zwnd probe pkts:           0
  zwnd probe bytes:          0
  outoforder pkts:         529
```

(a) `tcptrace` output



```
=================================================
| IO Statistics
|
| Interval size: 1 secs
| Col 1: Frames and bytes
|     2: COUNT(tcp.analysis.retransmission) tcp.an
|     3: COUNT(tcp.analysis.out_of_order) tcp.anal
|------------------------------------------------
|          |1                   |2       |3       |
| Interval | Frames |   Bytes   | COUNT  | COUNT  |
|-----------------------------------------------|
|  0 <>  1 |    202 |    144590 |      0 |      0 |
|  1 <>  2 |      0 |         0 |      0 |      0 |
|  2 <>  3 |      0 |         0 |      0 |      0 |
|  3 <>  4 |      0 |         0 |      0 |      0 |
|  4 <>  5 |      0 |         0 |      0 |      0 |
|  5 <>  6 |      0 |         0 |      0 |      0 |
|  6 <>  7 |      8 |       529 |      0 |      0 |
|  7 <>  8 |      0 |         0 |      0 |      0 |
|  8 <>  9 |      0 |         0 |      0 |      0 |
|  9 <> 10 |      0 |         0 |      0 |      0 |
| 10 <> 11 |      0 |         0 |      0 |      0 |
| 11 <> 12 |      0 |         0 |      0 |      0 |
| 12 <> 13 |      0 |         0 |      0 |      0 |
| 13 <> 14 |      0 |         0 |      0 |      0 |
| 14 <> 15 |      0 |         0 |      0 |      0 |
| 15 <> 16 |  55339 |  54948848 |     38 |      1 |
| 16 <> 17 | 118320 | 120239916 |     64 |      0 |
| 17 <> 18 | 117316 | 119145480 |     68 |      2 |
| 18 <> 19 | 116009 | 117610058 |     82 |      1 |
| 19 <> 20 | 113870 | 115188992 |     87 |      1 |
| 20 <> 21 | 116726 | 118381012 |     80 |      0 |
| 21 <> 22 | 118297 | 120037246 |     74 |      3 |
| 22 <> 23 |  64389 |  65577184 |     28 |      0 |
=================================================
```

(b) `tshark` output

Figure 4.24: Case 3: Analysis of a packet trace from an experiment on a path with guaranteed sequential delivery

4.6 Conclusions

In high-speed networks, end-to-end data-transfer throughput is affected not only by conditions on the network links, but also by the choice and configuration of the various protocol layers within the end hosts. This chapter describes lessons learned and offers insights into the Linux TCP/IP stack that the data traverses before being transmitted on to the communication medium and after being received. Lessons learned are presented for three layers: (i) Application Layer, (ii) Transport Layer, and (iii) Data-Link Layer, in addition to recommendations on how to monitor flows. These lessons and recommendations are not limited to the use of circuits/VCs, but are beneficial to network researchers in general.

One of our key findings with regards to the *Application Layer* is that `nuttcp-6.1.2` reports an inaccurate number of retransmitted packets. This version of `nuttcp` is part of the default GNU package. But we found the reports from the latest version, `nuttcp-7.3.3` to be accurate. The reports from `iperf3` showed a 2% inaccuracy rate in our measurements. Therefore, we recommend the use of `nuttcp-7.3.3`.

For the *Transport Layer*, we found that the TCP receive-side flow window and TCP send-side congestion window are limited to half the maximum value of the TCP receive buffer and sender buffer, respectively. Therefore, to achieve maximum throughput, the maximum of these buffers should be set to twice the bandwidth-delay product. The ESnet website [17], which offers users tips for high-speed transfers, suggests setting `tcp_rmem max` to a large value, e.g., 64 MB on a 10-Gbps, 100-ms path. But we caution that while this large value is necessary to achieve high throughput on high bandwidth-delay-product (BDP) paths, the maximum buffer size should be sized according to each path's BDP. This is because when the TCP sending window is in tens of MBs, the probability of packet loss increases as switch buffers, especially in campus and top-of-rack, switches, are often just a few MBs or even smaller. But to set the maximum buffer size for each TCP socket based on the path BDP, requires modification of the application code.

For the *Data-Link Layer*, two key findings were presented. First, in the TBF queueing discipline, the initial condition is a full token bucket. Second, for a TBF specified `rate`, if the inter-packet transmit time is less than the system timer and the HRT feature is not available, then a sufficiently large `burst` size is required to achieve the specified `rate`, and packets will be sent in bursts at the NIC rate on every expiration of the system timer. On the other hand, if the inter-packet transmit time to achieve the specified `rate` is greater than the system time (or high-resolution timer if available), then a single packet will be sent on every timer expiration, and packets will not appear at the NIC rate in bursts.

This first finding explains the high packet-loss rate observed in the experimental results presented in Chapter 3, where Table 3.3 showed that in more than 50% of the runs, CTCP had packet retransmissions in the first second and zero packet retransmissions in the other seconds. This behavior can be explained by our finding that TBF starts with a bucket that is full of tokens. For the experiments presented in Chapter 3, TBF `burst` size (bucket size) was 100 KB, which means a burst of 100 KB was sent at the NIC rate (10 Gbps) starting from the first second. We know that the UVA WAN-access link has approximately 4 Gbps of background traffic. Therefore, even this small burst could have caused the router buffer, and/or campus switch buffers including the UVA DYNES switch, to overflow, causing packet losses. The second finding implies that with HRT, probability of packet loss due to a switch buffer overflow will be smaller since packets are not sent in bursts. Therefore, we recommend compiling the kernel with the `CONFIG_HIGH_RES_TIMERS` option to enable HRT.

Finally, the *Monitoring* section provided recommendations on how to capture flow packets for detailed analysis. The recommendations can be summarized as follows: (i) the `tcpdump` tool should be used with `-B` and `-s` flags, and the `tcpdump` log should be checked to ensure that the kernel did not drop any packets, and correspondingly the collected trace contains information about all packets of the flow, and (ii) the offload (TSO/LRO and GSO/GRO) features should be disabled when collecting packets for detailed analysis. However, these offload features should be enabled for the actual data transfers especially for high-throughput flows. In addition to the previous recommendations, we demonstrated the best methods for determining the number of retransmitted packets using `tcptrace` and/or `tshark`.

Chapter 5

Conclusions and Future Work

Problems arise when transferring large datasets at high speeds using the best-effort IP service offered on the Internet. The first problem is related to the throughput of the large-data transfers. On high bandwidth-delay-product (BDP) paths, even a low packet-loss rate can cause significant drops in throughput because the TCP sender lowers its sending rate in an effort to be fair to other flows. Since packet losses could occur on some transfers, and not on others, the throughput experienced by transfers on the same path could have significant variance. Difficulties in predicting throughput lead to unpredictable transfer completion times, which make it challenging for scientific high-performance computing workflow management systems to schedule computing resources if network transfers are involved. The second problem is that high-speed, large-sized transfers can have adverse effects on the real-time (audio/video) flows, which typically have delay constraints.

To address these problems, this thesis presented an approach that leverages rate-guaranteed Layer-2 (L2) services for high-speed large-dataset transfers. Transfers on a rate-guaranteed L2 path should suffer from packet losses due to router/switch buffer overflows because bandwidth resources are reserved for the flow during L2 path setup prior to data transfer. Hence the flow can experience high throughput (provided a high-rate path was available), and transfer completion times will be more predictable. In addition, with L2 paths, large data-transfers flows can be isolated to prevent adverse effects on other flows. Mechanisms such as policing flows, scheduling packets, and directing packets of high-speed large-sized transfers to separate switch/router buffers can be used to isolate these flows. The disadvantage is that while rate-guaranteed Layer-2 (L2) service is being offered by some providers, it is not yet ubiquitous. The emergence of OpenFlow/SDN technologies has improved prospects for this service deployment to spread, but the deployment is still work-in-progress.

An experimental study was conducted on an inter-domain L2 path provisioned across production (operational) networks. An application was used to generate a flow at 4 Gbps on the end-to-end path. Throughput close to the L2 path rate was achieved, but packet losses were observed. In this environment, the L2 path was semi-rate guaranteed, i.e., policing and scheduling mechanisms at the routers/switches on the path were not implemented, and real background traffic, on which we had no control, shared the same links with our experimental high-rate large-sized flows. Hence, we carried out our experiments in a more controlled environment (single rack with emulated delays and packet losses) to design a solution for conducting high-speed large-sized transfers over L2 paths.

We developed a cross-layer design based on experimental studies that considered three cases: (i) single circuit/rate-guaranteed VC from a server, where we recommended Circuit TCP (CTCP) and the Linux `tc` Token Bucket Filter (TBF) queueing discipline with methods for selecting parameter values. (ii) multiple dynamic circuit/rate-guaranteed VC from a server to multiple receivers, where we recommended the combination of CTCP and Hierarchical Token Bucket (HTB) discipline. (iii) semi-rate-guaranteed VCs, where we recommended CTCP if the primary consideration is for the large transfers, and HTCP if the primary consideration is for other flows.

Experimental network studies require an understanding of the end-host operating systems and networking protocol stacks. Lessons learned and insights gained about the Linux TCP/IP stack were presented in the thesis to help other experimental networking researchers. Our key findings relate to software in three layers: (i) application layer, (ii) transport layer, and (iii) data-link layer. In addition, key findings related to using `tcpdump`, `tshark`, and `tcptrace` for flow monitoring were also presented in this thesis.

Future work items include: (i) extending our cross-layer design to include disk-to-disk transfers, (ii) modeling packet loss rate as a function of TCP sending window size and bottleneck-link switch buffer size, and (iii) answering the questions raised about where to capture packets, at the sender or at the receiver.

The cross-layer designed presented in this thesis considered only memory-to-memory transfers. Methods were proposed for selecting parameters of transport-layer protocols and link-layer rate-control mechanisms. In disk-to-disk transfers, contention for disk access should be considered as well.

We recommended the use of large TCP buffers at the sender and receiver on high BDP-paths. We also ran experiments to determine the size of switch buffers, and found that campus and top-of-rack switches often have fairly small buffers (on order of 1-10 MB). If the TCP sending window is much larger than switch buffers, there can be packet losses whenever aggregate traffic exceeds link capacity.

In Section 4.5.2, we recommended that careful thought should be given on whether to capture packets at the sender, or at the receiver, or at both ends. We plan to execute experiments to provide better guidance on where to capture packets based on the intended use.

Bibliography

[1] "Internet2 Advanced Layer 2 Services (AL2S)." http://www.internet2.edu.

[2] "Open Exchange Software Suite (OESS)." http://globalnoc.iu.edu/sdn/oess.html.

[3] "On-Demand Secure Circuits and Advance Reservation System (OSCARS)." http://www.es.net/OSCARS/docs/index.html.

[4] V. Jacobson and M. J. Karels, "Congestion avoidance and control," in *Proceedings of the Sigcomm '88 Symposium*, pp. 314–329, 1988.

[5] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The science DMZ: A network design pattern for data-intensive science," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 85:1–85:10, ACM, 2013.

[6] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger, "Pegasus: a workflow management system for science automation," *Future Generation Computer Systems*, vol. 46, pp. 17–35, 2015. Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575.

[7] Z. Yan, C. Tracy, M. Veeraraghavan, T. Jin, and Z. Liu, "A network management system for handling scientific data flows," *J. Netw. Syst. Manage.*, vol. 24, Jan. 2016.

[8] S. Tepsuporn, F. Al-Ali, M. Veeraraghavan, X. Ji, B. Cashman, A. J. Ragusa, L. Fowler, C. Guok, T. Lehman, and X. Yang, "A multi-domain SDN for dynamic layer-2 path service,"

[9] F. A. M. Veeraraghavan, “A cross-layer design for large transfers in sdns,” *International Conference on Ubiquitous and Future Networks (ICUFN)*, 2016.

[10] “OpenFlow Discovery Protocol and Link Layer Discovery Protocol.” http://groups.geni.net/geni/wiki/.

[11] “IEEE 802.1Q standard specifying Virtual LANs and VLAN Bridges.” http://www.ieee802.org/1/pages/802.1Q.html.

[12] J. Zurawski, R. Ball, A. Barczyk, M. Binkley, J. Boote, E. Boyd, A. Brown, R. Brown, T. Lehman, S. McKee, B. Meekhof, A. Mughal, H. Newman, S. Rozsa, P. Sheldon, A. Tackett, R. Voicu, S. Wolff, and X. Yang, “The DYNES instrument: A description and overview,” *Journal of Physics: Conference Series*, vol. 396, no. 4, p. 042065, 2012.

[13] “perfSONAR-PS.” http://psps.perfsonar.net/.

[14] D. X. Wei, C. Jin, S. H. Low, and S. Hegde, “Fast tcp: Motivation, architecture, algorithms, performance,” *IEEE/ACM Trans. Netw.*, vol. 14, pp. 1246–1259, Dec. 2006.

[15] S. Ha, I. Rhee, and L. Xu, “CUBIC: a new TCP-friendly high-speed TCP variant,” *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.

[16] D. Leith and R. Shorten, “H-TCP: TCP for high-speed and long-distance networks,” in *Protocols for Fast Long Distance Networks Workshop (PFLDnet)*, Feb. 16-17, 2004.

[17] ESnet, “ESnet Fasterdata Knowledge Base.” http://fasterdata.es.net/.

[18] Y. Gu and R. L. Grossman, “UDT: UDP-based data transfer for high-speed wide area networks,” *Computer Networks*, vol. 51, no. 7, pp. 1777–1799, 2007.

[19] “A. Feng, ”RAPID: Rate-Adjusting Protocol for Internet Delivery”, 2001.” http://public.lanl.gov/radiant/research/hpn/rapid.html.

[20] E. He, J. Leigh, O. Yu, and T. A. Defanti, “Reliable blast udp : predictable high performance bulk data transfer,” in *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pp. 317–324, 2002.

[21] B. Tierney, E. Kissel, M. Swany, and E. Pouyoul, “Efficient data transfer protocols for big data,” in *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pp. 1–9, Oct 2012.

[22] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, “The globus striped gridftp framework and server,” in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, SC ’05, (Washington, DC, USA), pp. 54–, IEEE Computer Society, 2005.

[23] “Fast-Data Transfer (FDT).” http://monalisa.cern.ch/FDT/.

[24] F. Schmuck and R. Haskin, “GPFS: A Shared-Disk File System for Large Computing Clusters,” in *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST ’02, (Berkeley, CA, USA), USENIX Association, 2002.

[25] “The Lustre file system.” http://www.internet2.edu/media/medialibrary/2013/08/20/Proposed_Internet2_Fee_Model_Changes.pdf.

[26] GridFTP. http://globus.org/toolkit/docs/3.2/gridftp/.

[27] A. Rajendran, P. Mhashilkar, H. Kim, D. Dykstra, G. Garzoglio, and I. Raicu, “Optimizing Large Data Transfers over 100Gbps Wide Area Networks,” in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pp. 26–33, May 2013.

[28] M. Beck, T. Moore, and J. S. Plank, “An end-to-end approach to globally scalable network storage,” *SIGCOMM Comput. Commun. Rev.*, vol. 32, pp. 339–346, Aug. 2002.

[29] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, “How hard can it be? designing and implementing a deployable multipath tcp,” in *Pro-*

ceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, (Berkeley, CA, USA), pp. 29–29, USENIX Association, 2012.

[30] G. Khanna, U. Catalyurek, T. Kurc, R. Kettimuthu, P. Sadayappan, I. Foster, and J. Saltz, "Using overlays for efficient data transfer over shared wide-area networks," in *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for*, pp. 1–12, Nov 2008.

[31] N. Hanford, B. Tierney, and D. Ghosal, "Optimizing data transfer nodes using packet pacing," in *Proceedings of the Second Workshop on Innovating the Network for Data-Intensive Science*, p. 4, ACM, 2015.

[32] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar, "Geni: A federated testbed for innovative network experiments," *Computer Networks*, vol. 61, no. 0, pp. 5 – 23, 2014. Special issue on Future Internet Testbeds – Part I.

[33] M. McGinley, H. Bhuiyan, T. Li, and M. Veeraraghavan, "An in-depth cross-layer experimental study of transport protocols over circuits," in *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pp. 1–6, Aug 2010.

[34] B. Hubert, "Linux Advanced Routing & Traffic Control." http://www.lartc.org/manpages/tc.pdf.

[35] "Brocade ICX 6610 Switches Frequently Asked Questions." http://www.brocade.com/content/brocade/en/backend-content/pdf-page.html?/content/dam/common/documents/content-types/faqs/icx-6610-switch-faq.pdf.

[36] "Globus online."

[37] "TCP Linux Man Page." http://linux.die.net/man/7/tcp.

[38] A. N. Kuznetsov, "Token Bucket Filter - Linux man page."

[39] “Time - linux man page.” http://linux.die.net/man/7/time.

[40] “Queueing in the Linux Network Stack.” https://www.coverfire.com/articles/queueing-in-the-linux-network-stack/.

[41] “Understanding the Kernel Network Layer.” http://raisama.net/talks/fisl10/kernel-hacking/network.pdf.

[42] C. Dovrolis, P. Ramanathan, and D. Moore, “What do packet dispersion techniques measure?,” in *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 2, pp. 905–914 vol.2, 2001.