MMPTCP: A Novel Transport Protocol for Data Centre Networks

Morteza Kheirkhah Sabetghadam

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

University of Sussex.

Department of Informatics

University of Sussex

19 November, 2015

I, Morteza Kheirkhah Sabetghadam, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

I also declare that this thesis has not been and will not be submitted in whole or in part to another University for the award of any other degree.

19 November, 2015

Abstract

Modern data centres provide large aggregate capacity in the backbone of networks so that servers can theoretically communicate with each other at their maximum rates. However, the Transport Control Protocol (TCP) cannot efficiently use this large capacity even if Equal-Cost Multi-Path (ECMP) routing is enabled to exploit the existence of parallel paths. MultiPath TCP (MPTCP) can effectively use the network resources of such topologies by performing fast distributed load balancing. MPTCP is an appealing technique for data centres that are very dynamic in nature. However, it is ill-suited for handling short flows since it increases their flow completion time.

To mitigate these problems, we propose Maximum MultiPath TCP (MMPTCP), a novel transport protocol for modern data centres. Unlike MPTCP, it provides high performance for all network flows. It also decreases the bursty nature of data centres, which is essentially rooted in traffic patterns of short flows. MMPTCP achieves these nice features by randomising a flow's packets via all parallel paths to a destination during the initial phase of data transmission until a certain amount of data is delivered. It then switches to MPTCP with several subflows in which data transmission is governed by MPTCP congestion control. In this way, short flows are delivered very fast via the initial phase only, and long flows are delivered by MPTCP with several subflows.

We evaluate MMPTCP in a FatTree topology under various network conditions. We found that MMPTCP decreases the loss rate of all the links throughout the network and helps competing flows to achieve a better performance. Unlike MPTCP with a fixed number of subflows, MMPTCP offers high burst tolerance and low-latency for short flows while it maintains high overall network utilisation. MMPTCP is incrementally deployable in existing data centres because it does not require any modification to the network and application layers.

Acknowledgements

Working towards a PhD has been a very enriching experience; it has had its high moments filled with excitement, and its low moments filled with exhaustion. All of these have changed my character and influenced every aspect of my life: my mindset, my attitude and my life-style. This unforgettable experience would not have been possible without the help of many people (near and far), and I would like to extend to them my heartfelt thanks.

First and foremost, my supervisor, Ian Wakeman. I am not sure where to start and how to thank him. Without Ian, I would not have been able to reach this point and write this acknowledgement - his support over the course of my PhD has gone beyond technical aspects. Ian has given me an opportunity to think freely and creatively, and always guided me in the right direction when I needed it most - and always with positivity. His insights and discussions have helped me to shape this work. His empowering encouragement and patience have allowed me to easily embrace challenges and handle them successfully. For all of this, Ian, thank you.

I would also like to thank George Parisis, who joined our group - Foundation of Software Systems (FOSS) at the University of Sussex - when I was halfway through my PhD. Since then George has been kindly supporting and helping me. I have benefited from his experiences in the ns-3 simulator and Data Centres.

My PhD has been fully funded by the Department of Informatics at the University of Sussex for which I am very grateful. I would like to thank current and past members of the Department of Informatics who have helped and supported me over the course of my PhD, namely John Carroll, Dan Chalmers, Martin Berger and Bernhard Reus. A big thank you goes to Des Watson and Simon Fleming for their comments and suggestions on the final draft of this dissertation.

I would also like to thank Mark Handley at University College London (UCL) for our initial discussions and for sharing his valuable insights into MultiPath TCP and Data Centre Networks. Mark has helped me to reach an in-depth understanding of the problem spaces in these contexts. I am also very grateful to Mark as my external PhD examiner for his insightful comments and guidance on my thesis.

I am profoundly beholden to Damon Wischik, my MSc supervisor at UCL. Damon taught me how to undertake advanced research. His office door was always open to me, both during and long after I completed my studies at UCL. I had pleasant times discussing new ideas with him.

Last but not least, I would like to thank my family, especially my sister, Maryam, and my brother, Daryoush, for their unwavering support during my studies in England. This PhD is devoted to my mother, Parvin, and my father, Mostafa, who raised me, loved me, and encouraged my talents and enthusiasm for science.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large-scale data centres consist of tens of thousands of networked computers which provide services to cloud applications. Examples of cloud applications are Facebook and Google Search Engine, and examples of cloud services are MapReduce and Google File System (GFS), to name only a few.

Each cloud application has its own communication patterns, such as different bandwidth and/or communication requirements. Some applications are latency-sensitive and others are bandwidth-hungry. For example, typical MapReduce services require loosely-synchronised all-to-all communication among some servers inside data centre networks, i.e. a subset of servers require to communicate with each other simultaneously. Furthermore, each server does not necessarily support only one service at a time. It is common to have multiple services running in a server via virtual machines. *Data centre networks should thus be able to support diverse communication patterns for servers needing to support diverse applications.*

A typical data centre consists of switches and routers in a two- or three-level hierarchical tree structure. That is, the network topology can be designed, built and expanded quite easily. In this type of topology, servers are located in racks and are connected to Top-of-Rack (ToR) switches, which are in turn connected to aggregation switches (Agg). Finally, aggregation switches are aggregated further up and connected to a core switch.

The network topology of a conventional data centre does not typically provide full bisection bandwidth[1] between all pairs of servers. This is because the core switches

[1] Full bisection bandwidth is achieved when a pair of servers communicating together with maximum capacity of their network interface cards (NICs).

do not have a sufficient number of ports to aggregate enough bandwidth between all servers due to the high cost and hardware limitations. Achieving full bisection bandwidth is possible only if links, especially in the network core, are not oversubscribed i.e. the oversubscription ratio in all links is 1:1. However, in practice the aggregated bandwidth in the access layer is far more than the aggregated bandwidth in the aggregation or core layer. Thus, each rack supports more than 20-40 servers [1]. The links between ToRs and Aggs therefore typically have an oversubscription ratio of more than 2:1, and links between Aggrs and Cores often have an oversubscription ratio of more than 8:1 i.e. links in the network core are oversubscribed by a factor of eight. Generally, by going up the hierarchical tree topology the oversubscription ratio is increased and the non-blocking bandwidth is decreased.[2]

It is argued that conventional data centre architectures are optimised for handling traffic into and out of data centres. However, this is not the case for intra-domain traffic that comes from bandwidth-hungry applications, such as Virtual Machine (VM) migrations,[3] video streaming and online file storage, which are becoming highly popular [2, 3, 4, 5, 6, 7, 8]. The increase in traffic matrices such as these has led to the persistent congestion of some paths due to the lack of path diversity in the network topology and multipath support. This can result in significant degradation to the overall network performance.

Recently proposed data centre architectures, such as FatTree [5] and VL2 [4], attempt to solve the traffic concentration problem by redesigning the network topology. Their network topologies provide full bisection bandwidth between any pair of nodes by leveraging the idea of replacing a single high-capacity and expensive core switch at the top of the hierarchical tree structure with many commodity switches. The large aggregated capacity in the backbone of the network thus becomes available cheaply. This implies that the density of interconnection in the network topology is significantly increased and therefore multiple equal-cost[4] paths between servers become available. Thus, *in theory*, each server is able to use up to the maximum speed of its network interface device when communicating with any other server at any time. However,

[2]The non-blocking bandwidth is the amount of bandwidth that a server can theoretically achieve in any network condition.

[3]VM migrations are typically used in data centres as a traffic engineering technique to mitigate traffic sparks in a service by moving some of the VMs from the hotspots [2].

[4]We refer to parallel paths with equal hop count as equal-cost paths.

even though this large aggregated capacity is provided at the physical layer, i.e. via network topology, network and transport protocols cannot use it effectively since they have been designed for single-path data communication. For example, congestion still occurs in FatTree and VL2 since Random Load Balancing (RLB),[5]which seems to be the most effective multipath packet-forwarding strategy in data centres so far, does not perfectly distribute TCP flows to the available paths.

Hotspots might still occur frequently despite the fact that the network is providing full bisection bandwidth between all pairs of servers.

Central flow scheduling, Hedera [8], has been recently proposed to alleviate this problem. Hedera only takes care of long flows (since they are the main cause of persistent congestion).[6] However, the mechanism it uses to detect and react to congested links may not be fast enough for the dynamic nature of data centres [12, 13].

An alternative solution to central flow scheduling and TCP is to use MultiPath TCP (MPTCP) [14, 15], which delivers data via multiple paths *simultaneously* and moves most of the traffic from highly-congested to least-congested paths within a couple of round-trip times (RTTs).[7] In other words, unlike TCP and Hedera, MPTCP delivers a single flow via multiple paths and deals with possible congestion in the network gracefully by shifting its traffic away from those congested areas as much as possible very quickly. As shown in [12], MPTCP can improve overall network throughput and makes the network more robust in the event of failure. However, it is argued that MPTCP performance is very much related to the network topology construction [3]. For example, in VL2 its performance is only as good as regular TCP, whereas in FatTree [5] and BCube [6], MPTCP significantly improves overall network throughput. Furthermore, MPTCP is not well-suited to delivering short flows as their flow completion time may increase significantly. The problem is that the congestion window of the subflows of a MPTCP flow might be very small over the course of data transmission since the total number of packets of a short flow is small. Thus, even a single packet drop from a single subflow causes an entire MPTCP connection to wait for a retransmission timeout

[5]Equal-Cost Multi-Path (ECMP) [9] and Valiant Load Balancing (VLB) [10] are examples of RLB.

[6]Hedera requires the detection of long TCP flows at the edge switches. Hedera assumes a flow is large when that flow occupies 10% of the host-NIC bandwidth. The central controller only schedules long flows, while the switches route short flows using hashed-based ECMP to randomise their routes. Hedera can only be implemented on programmable switches, such as OpenFlow [11].

[7]RTTs in the data centres are often in the order of microseconds.

to be triggered on that subflow since the lost packet cannot be recovered by the Fast Retransmit mechanism due to the small number of packets in transit.

An alternative solution to MPTCP is the Packet Scatter (PS) [12] protocol. PS is inspired by Valiant Load Balancing (VLB) [10]. The idea is that network switches randomise traffic on a *per-packet*, instead of a *per-flow*, basis, which, as mentioned above, may still cause hotspots. In this way, a flow can be delivered via all available paths to the destination. As argued in [12], the packet-scattering approach does not create hotspots in the core layer of the *multi-rooted* tree topology, such as VL2 or FatTree, if the load is the same among all the servers and hardware failures do not occur. PS can thus achieve a higher overall network throughput than MPTCP. However, PS is sensitive to network congestion, i.e. it cannot react to congestion gracefully because it is a conventional TCP connection and only holds a single congestion window per connection. Thus, if any packet gets dropped because of network congestion as a result of hardware failures or bursty traffic, PS reduces its congestion window by half and goes to the congestion avoidance phase in which the data rate increases linearly per RTT; i.e. it has a significant negative effect on the overall network throughput.

Another area closely related to our research is traffic management inside data centres. Prior work in this area has tried to decrease flow completion time, especially for short flows that are latency-sensitive and have deadlines in their flow completion time. Web search, retail, social network content composition and advertisement selection are a few notable examples of such services. These online services typically leverage *partition/aggregate* workflows, dividing queries across many workers and aggregating the computations to produce the user reply. Workers typically have the shortest deadlines to complete their tasks/computations (e.g. 10-100ms) and if they cannot finish them before elapse of their deadlines, the aggregators ignore their computations as the search operation is also limited by a deadline (e.g. 200-300ms). This has a direct impact on search quality and hence revenue (e.g. an added 500ms latency in Google online search reduced its traffic by 20%. Amazon sales dropped by 1% by adding 100ms latency. Online brokers could lose 4 million US dollars per millisecond if they fall 5ms behind their competitors [16]). Furthermore, flows that missed their deadlines also wasted valuable network resources [17, 18, 13]. Unfortunately, the proposed solutions in this area either employ single-path mechanisms [17] and/or do not support

incremental deployment [18] or even require advanced switches or high capacity links (i.e. at least 10 Gbps) not only between switches, but also between hosts and switches, (i.e. they require each server to have 10Gbps NIC) [13]. These requirements make the proposed solutions less attractive.

In this thesis, we investigate a solution to prevent transient congestion due to bursty short flows while reacting to persistent congestion in a short timescale. Our primary goals are that long flows should achieve a high connection throughput and short flows should achieve fast data delivery. Achieving these two goals is not trivial, as we have observed that a tension exists between them. For example, the existing solutions that provide a high overall network throughput, such as Hedera and MPTCP, fail to prevent transient congestion and hence fast data delivery for short flows. Those existing solutions that do provide fast data delivery for short flows, such as DCTCP, fail to provide a very high throughput for long flows. Our approach for achieving these goals is to effectively use path diversity in modern data centres.

To this end, we present Maximum MultiPath TCP (MMPTCP), a novel transport protocol for the latest data centre networks [19, 20]. We were inspired to design MMPTCP by the following observations:

- MPTCP is a well-suited alternative to the single-path TCP in data centres, whose topologies provide dense interconnectivity in the network. MPTCP has been designed to effectively use such networks. It is shown that MPTCP can achieve more than double the overall network throughput in FatTree [5] and BCube [6] topologies compared to single-path TCP [12]. Figure 1.1 is a showcase for this performance enhancement via a simulation in the network simulator-3 (ns-3) with our custom implementation of MPTCP.[8] MPTCP with eight subflows almost doubles the overall goodput of MPTCP with a single subflow (i.e. TCP).

  Another important benefit of MPTCP in data centres is that its reaction time to congestion is very fast. It can remove traffic from congested links within a few RTTs (unlike Hedera) and thereby solves the traffic concentration problem to a great extent. MPTCP is thus an appealing approach for data centres characterised by an extremely dynamic nature. However, our examinations in this

[8] The simulation setup in Figure 1.1 is: the FatTree topology with 128 nodes running a Permutation traffic matrix of long MPTCP flows.

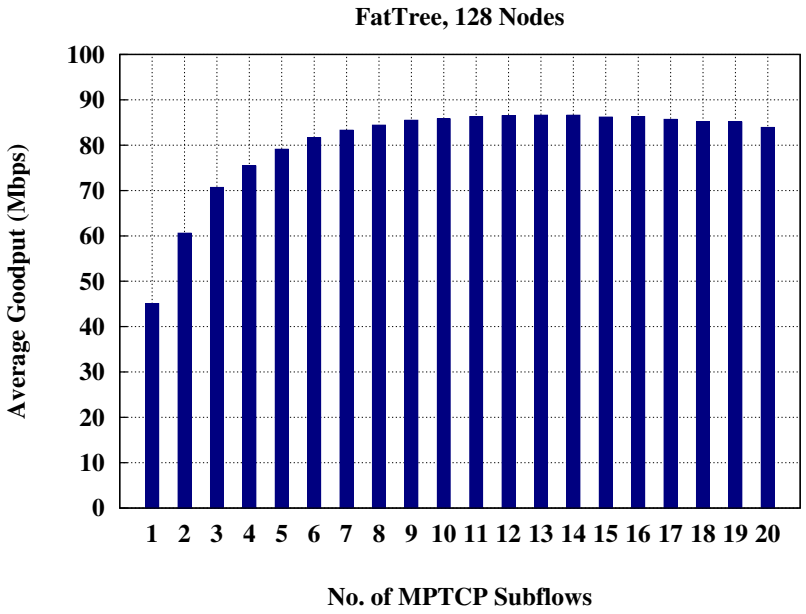


Figure 1.1: Long flows with a varying number of subflows. MPTCP with eight subflows almost doubles the overall goodput of long flows of single-path TCP.

thesis demonstrate that MPTCP with more than one subflow does not perform well for short flows and hurts their flow completion times. Figure 1.2 depicts this behaviour.[9] As the number of subflows increases, the standard deviation also increases significantly. This implies that a fraction of short flows experiences a series of retransmission timeouts. Additionally, as the number of subflows increases, the mean flow completion also increases; this is better shown in the small plot in Figure 1.2.

We find ourselves at an impasse: MPTCP is damaging short flows while a majority of data centre flows are short-lived. However, it performs very well for long-lived flows, which comprise a majority of data centre bytes [4, 3]. This is our main motivation for designing the MMPTCP protocol for data centres.

- Data centre traffic patterns are very bursty and unpredictable. The bursty traffic pattern of data centre networks originates from short flows, which comprise 99% of total flows [4]. The dynamic nature of data centres may create tran-

[9]The simulation setup in Figure 1.2 is: the 4:1 oversubscribed FatTree topology with 512 nodes running a Permutation traffic matrix. 33% of nodes send continuous traffic and the remainder send short flows as they are assigned by central flow scheduler with arrival rate of 250 per second in average (Poisson arrival).

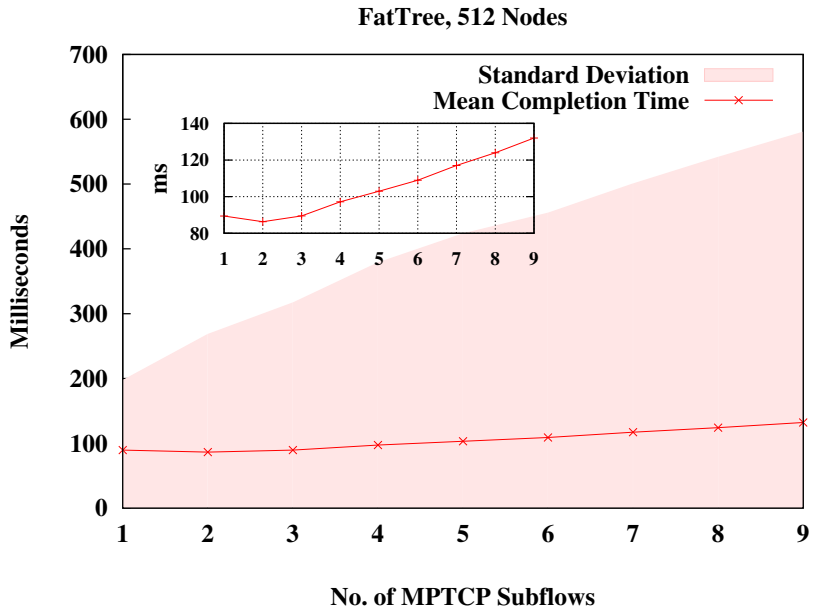


Figure 1.2: Short flows with a varying number of subflows. The small plot is the zoom version of the big plot with the aim of showing the mean flow completion times clearly.

sient congestion in any link in the network.[10] This transient congestion occurs when a switch buffer is suddenly saturated due to a surge of traffic from several sources. This is particularly the case in MapReduce- or partition/aggregate-like traffic patterns [17]. Transient congestion significantly degrades the overall network throughput and damages the flow completion time of short flows.

- The majority of short flows in data centres are latency-sensitive and must complete their flow within a predefined deadline, typically in a few milliseconds. If they cannot deliver all data before their deadlines some computations/results are discarded, lowering the quality of results or restarting tasks and wasting network resources. Flows subject to a deadline typically miss that deadline due to encountering transient and/or persistent congestion in their paths.

MMPTCP is an extension of MPTCP, which can provide the following nice features to data centre networks:

- It allows MPTCP to perform well for all network flows.
- It accommodates any burst of traffic gracefully by utilising path diversity in modern data centre fabrics and so significantly preventing transient congestion.

[10]The congestion due to collision between synchronised flows, typically from applications with partition/aggregate workflow, in the access layer's bottleneck links is referred to as the *Incast* problem.

- It reacts to congestion by moving traffic away from congested paths, so that it achieves a high overall throughput and largely prevents persistent congestion.

- It can handle deadline flows without any knowledge from applications, if Quality of Service (QoS) is present in the data centre switches.

- It can be incrementally deployed in existing data centres since it can coexist with other transport protocols, such as legacy TCP.

1.1 Contributions

Our key contributions in this research are as follows:

- Examination of MultiPath TCP for short flows in a FatTree network topology. This includes teasing apart the participation of each of the mechanisms used by MPTCP. This led us to realise that MPTCP with a fixed number of subflows falls short of handling short flows without hurting their flow completion times.

- Design, implementation and evaluation of MMPTCP in a wide range of scenarios in a FatTree topology. This includes the study of the automatic adjustment of *dupthresh* and limited transmit in such topologies.

- In-depth analysis of per packet load balancing (PS) in a FatTree topology. This has led us to think a new congestion control may significantly improve the performance of PS.

- Studying the feasibility of running MPTCP/MMPTCP simulation, via the packet-level even-driven simulator (ns-3), as the network size and link rate increase.

- Implementation of MultiPath TCP in the ns-3 simulator [21]. This includes the implementation of several multipath congestion controls. We have written more than 10K lines of code (C++ language) in this thesis. The source code of our implementations can be found using the following link: https://github.com/mkheirkhah/.

1.2 Published and Presented Works

- M.Kheirkhah, I.Wakeman and G.Parisis. MMPTCP: A Multipath Transport Protocol for Data Centres. In Proceeding of the 2016 IEEE International Conference on Computer Communications (INFOCOM '16), San Francisco, CA, USA.

- M.Kheirkhah, I.Wakeman and G.Parisis. Short vs. Long Flows: A Battle That Both Can Win. In Proceeding of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM '15), pages 349-350.

- M.Kheirkhah, I.Wakeman and G.Parisis. Multipath-TCP in ns-3. In the 2014 Workshop on ns-3 (WNS3 '14), Atlanta GA, USA.

- M.Kheirkhah. MMPTCP: A Novel Transport Protocol for Data Centre Networks. Presentation in the Work In Progress Seminars (WIPS), School of Engineering and Informatics, University of Sussex, Brighton, UK. June 2015. Also in the Multi-Service Networks (MSN15) workshop, Abingdon, UK. July 2015.

- M.Kheirkhah. Implementation of MultiPath TCP in Network Simulator-3. Presentation in the Multi-Service Networks (MSN14) workshop, Abingdon, UK. July 2014.

- M.Kheirkhah and G.Parisis. Data Centre Networking and Storage: Challenges and Our Current Research. Presentation in the Work In Progress Seminars (WIPS), School of Engineering and Informatics, University of Sussex, Brighton, UK. May 2014.

- M.Kheirkhah. Routing Domains in Data Center Networks. Presentation in the Multi-Service Networks (MSN11) workshop, Abingdon, UK. July 2011.

1.3 Thesis Structure

The remainder of this thesis is structured as follows:

Chapter 2 provides an overview of the TCP and MPTCP protocols, including their congestion controls. This is followed by an overview of the data centre problems and existing solutions.

Chapter 3 presents our novel transport protocol, including the goals, challenges, and benefits of MMPTCP in data centres.

Chapter 4 presents a high-level overview of our implementations within the ns-3 simulator; our implementation of MPTCP has also been presented in the Workshop on ns-3 (WNS3 '2014) [21]. Showcasing our implementations through a series of simple simulations follows this.

Chapter 5 describes our simulation setup, including traffic matrices, network topology and simulation templates. Comparing MMPTCP with other existing transport protocols in a wide variety of network scenarios follows this.

Chapter 6 concludes our research and discusses our future plans.

Chapter 2

Background

2.1 Introduction

Our novel transport protocol (MMPTCP) is based on the TCP and MPTCP protocols and designed to operate in modern data centres, which provide dense interconnectivity in the network. This chapter overviews concepts that are relevant to the design and operation of the MMPTCP protocol.

This chapter is organised as follows:

- Section 2.2 presents an overview of the basic TCP and MPTCP functionality, focusing on TCP variants and multipath congestion control mechanisms.
  The TCP variants described in this section are: Tahoe [22], Reno, NewReno [23], SACK [24] and DSACK [25]. The aim of describing TCP variants is to explore their effectiveness in various network conditions. For example, DSACK can potentially operate more efficiently than TCP NewReno when packet reordering is the norm. One of the challenges faced in designing MMPTCP is to deal with packet reordering. The multipath congestion controls described in this section are: Uncoupled-TCP [12, 26, 27], Fully Coupled [28], Semi-Coupled [12] and Linked Increases [29].

- Section 2.3 overviews large-scale data centre architectures, operations and objectives. This section focuses on data centre applications/services, network topologies, communication patterns and network environments.

- Section 2.4 starts with a discussion of the traffic concentration problem within conventional data centre topologies and then explores solutions for mitigating it.

Such solutions are: localising traffic into racks, redesigning network topology to provide full bisectional bandwidth between all pairs of servers, and dynamic resource allocation. Providing full bisectional bandwidth in the network is a desirable trait since it can potentially solve the traffic concentration problem to a great extent.

- Section 2.5 reviews solutions to improve the overall network utilisation within modern data centres, which provide full bisectional bandwidth between hosts. These solutions are: MPTCP and Hedera (central flow scheduler). MPTCP is a particularly appealing solution since its reaction time to network congestion is much faster than Hedera. Thus, MPTCP is a better solution for dealing with the dynamic nature of data centres. We then discuss the limitations of MPTCP in the data centre context, which have essentially driven us to design our novel transport protocol, MMPTCP.

- Section 2.6 overviews another area of research within data centres, which is closely related to our research. It is related to short flows that contain deadlines in their flow completion time. The solutions considered in this section are DCTCP [17] and D^3 [18], both of which attempt to improve the flow completion time of short flows and help them deliver their data before their deadlines. One of the goals of MMPTCP is to allow short flows to deliver their data with a minimal experience of congestion in their paths.

2.2 Transport Protocols

2.2.1 TCP Protocol

TCP is a transport layer protocol that operates on top of the network layer (e.g. Internet Protocol) in the Open System Interconnection (OSI) model. TCP is the main transport protocol in today's Internet and provides reliable, in-order and error-checked data transmission between two end-hosts connected to the Internet.

TCP achieves its reliability and in-order data transmission in a packet switch network, in which packets can be delivered out-of-order, by assigning sequence numbers to data segments. Data bytes carried in a segment can be identified by the TCP header's sequence number and by the data length field, indicating a sequence number of the first

byte and the length of data bytes respectively. That is, if a packet gets dropped/delayed, the TCP receiver can easily detect the missing/delayed packet since the data should be reconstructed in an orderly fashion in the receiver before it can be delivered to the application layer.

TCP achieves its error-checked data transmission by a checksumming mechanism in which an entire stream of data bytes in a packet can be validated by the receiver. If even one part of the data bytes in a segment is altered, the receiver can easily detect and discard this corrupted segment. To achieve this, the TCP sender calculates a checksum, based on an agreed algorithm with the receiver, and places it in the checksum field of the TCP header per sent data segment. The receiver then applies the same checksum algorithm to validate the received data, and ensure that it has not been corrupted during data transmission.

TCP supports two important end-to-end data transmission rate control mechanisms: *flow control* and *congestion control*. The former mechanism prevents the TCP sender from transmitting more bytes than what the receiver is willing to receive at one time. As a result, the TCP receiver can prevent overflowing of its receive buffer. The latter mechanism is designed to react to network congestion caused by buffer overflows at bottleneck links in the network. The aim is to prevent further congestion when a loss event is detected.

TCP uses window-based congestion control that support additive increase and multiplicative decrease behaviour known as AIMD. TCP increases its sending rate additively when no loss event is detected, and decreases it multiplicatively when a loss event is detected [30, 12].

TCP's AIMD algorithm is, in short:

- Each ACK [1]increases window w by $\frac{1}{w}$.
- Each loss decreases w by $\frac{w}{2}$.

Additionally, TCP carries out an exponential increase, known as 'Slow-Start', at the beginning of its data delivery and after any retransmission timeout event. TCP is allowed to carry out a Slow-Start increase whenever the congestion window value is

[1]ACK refers to the ACK packet that is acknowledging a sequence number that has not been previously acknowledged.

shorter than the Slow-Start threshold value (*ssthresh*).

The Slow-Start algorithm is, in short:

- Each ACK, when $w < ssthresh$, increases the w by one full sized segment size[2].

2.2.1.1 Tahoe

Tahoe is one of the first versions of TCP proposed by Jacobson in 1988 [22] and solved the well-known problem of TCP congestion control known as TCP congestion collapse. This phenomenon involved TCP flows achieving extremely low throughput due to the lack of built-in congestion control mechanisms in TCP.

The design of the TCP congestion control algorithm proposed by Jacobson is based on a principle of *conservation of packets*: i.e. a packet can be placed into the pipe[3] only when an already sent packet exits the pipe, in order to achieve network stability. The *conservation of packets* principle refers to a condition in which a connection is in a *equilibrium* state, meaning that the connection is in the steady-state condition and running with a full window of data in transit [22]. Jacobson argued that by following this principle the congestion collapse incidents would be eliminated.

TCP Tahoe incorporates a new parameter, the congestion window (*cwnd*), that essentially controls the rate of data transmission and indicates the amount of bytes in transit. This new parameter prohibits a TCP sender from transmitting more than a *cwnd* worth of data even if permitted to do so by the receive window (*rwnd*).

TCP Tahoe also embodies three new algorithms into the original TCP [32]: *Slow-Start*, *Congestion Avoidance* and *Fast Retransmit*.

Slow-Start. A TCP connection enters the Slow-Start phase upon connection establishment or after a retransmission timer is triggered. The core idea of Slow-Starting is that a TCP sender begins its data transmission by probing the network; data transmission starts with one full-size segment and increases exponentially for every round trip time (RTT), i.e. *cwnd* starts with one MSS and is doubled per RTT.

Congestion Avoidance. A sender remains in the Slow-Start phase until its *cwnd*

[2]Maximum Segment Size (MSS) is the maximum size of payload that TCP allows to send. MSS depends on the link layer's Maximum Transmit Unit (MTU). The Ethernet network supports up to 1500 bytes MTU, which allows TCP to send 1460 bytes payload when the TCP option is not used.

[3]The *pipe* refers to a communication path between two endpoints of a TCP connection [31].

reaches a Slow-Start threshold (*ssthresh*). It then enters the congestion avoidance phase during which the data transmission rate is increased linearly, i.e. *cwnd* is increased by one segment per RTT, in order to prevent possible congestion en route.

Fast Retransmit. A sender stays in the congestion avoidance phase until a timeout is fired or after it has received three duplicate acknowledgements (duplicate ACKs) for the same TCP segment. In such cases, the sender infers that a segment has been lost, retransmits the perceived lost segments, halves its *ssthresh* and resets its *cwnd* to one MSS. Thereafter, the sender enters the Slow-Start phase to probe the network again and refills the pipe until it reaches the new value of *ssthresh*.

The whole point of the Fast Retransmit mechanism is that when a TCP sender receives a small number of duplicate ACKs for a lost segment, it retransmits the perceived lost segment promptly without waiting for the retransmission timer, referred to as Retransmission Timeout (RTO), to be triggered. In this way, a TCP connection can achieve higher network utilisation and throughput than the earlier version of TCP [32].

RTO Calculation. Jacobson also proposed a modification to the RTO calculation. In the early version of TCP [32], RTO is calculated based on Smoothed RTT (SRTT) as follows:

$$SRTT = (\alpha \times SRTT) + ((1-\alpha) \times RTT)$$

$$RTO = \beta \times SRTT$$

The parameter α is a smoothing factor with a suggested value of 0.8 or 0.9. The parameter β accounts for RTT variation with a suggested value of 1.3 or 2.0 [32].
Jacobson argued that constant $\beta = 2$ is only adopted to network loads of at most 30%; if a network loads goes beyond 30%, the higher value of β is required in order to estimate RTO accurately and prevent a single delayed packet needlessly triggering RTO (i.e. to prevent spurious retransmission). The argument is thoroughly simple: the queuing delay is increased as the network load is increased and so the constant $\beta = 2$ only supports a small queuing delay variation. Jacobson's proposal to address this issue is to adjust the value of β dynamically as follows:

$$RTO = SRTT + K \times RTTVAR$$

The parameter K is constant with a suggested value of 4 and RTTVAR is the mean deviation of the RTT samples. If there is little difference between the sample RTTs then the mean deviation of RTT samples effectively becomes zero (RTTVAR $\rightarrow$ 0). In

such cases SRTT is a good approximation for calculating RTO. On the other hand, if RTT samples show high variance (e.g. due to sudden surges of traffic), the RTO value is increased accordingly and promptly.

The estimated RTO is capped with another parameter, the so-called 'RTO_{min}', which is set by default to 200ms in Linux Kernel. This is a particular issue in networks with extremely low RTT, such as data centre networks that support RTTs in the order of microseconds. In such networks, a lost packet, which needs to be recovered through a RTO, is recovered after a very long delay (e.g. three orders of magnitude higher than the average RTT [33]). Decreasing the retransmission timer to a very small value may improve the TCP loss recovery process, but it may also create spurious retransmissions due to the queuing delays. Additionally, it is argued that the TCP clock granularity is in the order of milliseconds in current operating systems such as Linux. That is, RTO with a value of less than 5ms is not achievable [34].

2.2.1.2 Reno

TCP Reno, proposed by Jacobson as an extension of TCP Tahoe, introduces the *Fast Recovery* congestion control mechanism. The idea of Fast Recovery is that after Fast Retransmit is triggered, a TCP sender retransmits the perceived lost segments and halves its congestion window (at this condition, TCP Tahoe resets its *cwnd* to one MSS and enters the Slow-Start phase). The sender continues sending data in this phase upon arrival of the new ACK (called a 'recovery ACK'); the new ACK refers to the ACK packet that is acknowledging a data sequence number that has not been previously acknowledged. Thereafter, the sender enters the congestion avoidance phase.

During Fast Recovery, the sender inflates its congestion window for every received duplicate ACK and only places new segments into the pipe when its sending window, referred as a 'usable window', is not constrained by *cwnd* nor *rwnd* (i.e. in effect *cwnd* should be larger than flight bytes and *rcwnd* should not be smaller than *cwnd*). In other words, the sender must wait until half of duplicate ACKs have arrived before sending outgoing packets based on incoming duplicate ACKs. This is because its *cwnd* has already been cut by one half so needs to be re-inflated back to its original size before duplicate ACKs can trigger the placement of new segments into the pipe.

TCP Reno assumes that each duplicate ACK reflects a packet which has left the

network and been stored in the receiver's buffer. Hence, some space is freed in the network; sending new segments according to the arrival of incoming duplicate ACKs does not violate the *conservation of packets* principle. Thus, the sender prevents the pipe from going empty after a congestion signal is received (i.e. after Fast Retransmit is triggered) and maintains its ACK clock[4] in order to improve connection throughput and overall network utilisation.

TCP Reno's Fast Recovery operation is a significant improvement to TCP's loss recovery, but is only efficient when a single packet gets dropped from a single window's worth of data; each lost packet can be recovered during one RTT. In other words, Reno recovers from at most one packet drop per RTT. If multiple packets get dropped from a window's worth of data, Reno is not efficient as its loss recovery mechanism may require several RTTs [31].

2.2.1.3 NewReno

TCP NewReno is an enhancement to TCP Reno's loss recovery, in which multiple packet drops of a single window can be recovered much faster without having to wait for the retransmission timer to be triggered [23]. TCP NewReno modifies only the sender's behaviour during the Fast Recovery phase.

TCP NewReno categorises the new ACKs in the Fast Recovery phase as *partial* and *full* ACK. A *partial* ACK is a new ACK that acknowledges a portion of packets that were outstanding when Fast Retransmit was triggered. A *full* ACK is a new ACK that acknowledges all outstanding segments when Fast Retransmit was triggered. The key idea of *partial* ACK is that the sender should interpret it as an indication of a lost packet and retransmit the next segments in line. Unlike TCP Reno, which exits from Fast Recovery at this point, TCP NewReno stays in this phase until it receives a *full* ACK; i.e. Fast Recovery ends only when all losses from a single window of data are recovered. TCP NewReno exits Fast Recovery by resetting *cwnd* to *ssthresh* and enters the congestion avoidance phase.

This optimisation obeys the *conservation of packets* principle and allows senders to recover from multiple packet losses from a single window of data without waiting

[4]TCP supports the ACK clock congestion control where the data transmission is controlled by ACKs. If a TCP sender loses its ACK clock due to heavy packet drops en route then it cannot place new segments into the network since there is no ACK to trigger new segments.

for a retransmission timeout or re-entering Fast Retransmit. In other words, it prevents multiple reductions in the congestion window size due to multiple packet losses incurred in a single window of data. However, like Reno, it only recovers one lost segment per RTT since the sender does not have any idea about which segments are buffered in the receiver in order to deduce the lost segments precisely. In other words, TCP NewReno becomes aware of multiple losses from a window of data as they are recovered one by one via *partial* ACKs.

2.2.1.4 SACK

Selective Acknowledgement TCP (SACK TCP) is an alternative approach to TCP NewReno. It aims to improve loss recovery when multiple packets are lost from a window of data [24]. SACK does not change the underlying TCP congestion control algorithm but only requires some changes in the Fast Recovery algorithm. It carries its signalling information via a TCP option, the SACK option. SACK TCP provides the senders with precise information about delivered segments, allowing the sender to make correct decisions about which segments to retransmit when a loss signal is received, so that it can effectively retransmit more than one segment per RTT.

The core idea behind SACK is that TCP segments are acknowledged *cumulatively*, as in previous versions of TCP, but also *selectively* when the receiver observes non-contiguous segments in its buffer. To achieve this behaviour, a receiver includes a SACK option in each duplicate ACK it generates. A SACK option includes a number of SACK blocks, where each block represents a non-contiguous set of data stored in the receiver's buffer. The first SACK block must report to the most recently received out-of-order segment, which triggered this duplicate ACK, and the rest of the SACK blocks repeat the most recently reported SACK blocks [31].[5]

During the Fast Recovery phase, a sender initialises a new variable called *pipe*, which stores the estimated number of packets in transit. The main goal of this new variable is to ensure the new data packets are placing into the network while preserving the *conservation of packets* principle. A sender is allowed to send a new or retransmitted segment only when *cwnd* is larger than *pipe*. Whenever it retransmits a segment

[5]Each SACK block occupies eight bytes from the TCP option that has 40 bytes space only, meaning that only *four* SACK blocks can be reported in a single ACK packet, as a SACK option also occupies two bytes for option identification. If a SACK option is carried with the Timestamp option [35], then only *three* SACK blocks can be carried in a single ACK.

or sends a new segment, the *pipe* is incremented by one MSS. Whenever it receives a duplicate ACK with a SACK option, the *pipe* is decremented by one MSS. SACK TCP exits Fast Recovery similarly to TCP NewReno by receiving a *full* ACK.

2.2.1.5 DSACK

Duplicate Selective Acknowledgment TCP (DSACK [25]) is an extension of SACK TCP and behaves in exactly the same way with one exception: duplicate segments are also reported by the receiver in order to allow the sender to detect spurious retransmissions. DSACK uses a SACK option that reports a duplicate segment in its first SACK block. In order for the sender to infer a duplicate segment, the sender should compare the ACK field of the duplicate ACK packet containing a SACK option to the sequence space of the first SACK block [25]. If the cumulative acknowledgement field is higher than the sequence space of the first SACK block, then the sender can safely conclude that the reported segment in the first SACK block has been received more than once by the receiver, and a spurious retransmission has therefore occurred.[6] In such cases, the sender can undo the unnecessary reduction in its *cwnd* by resetting *ssthresh* to the old (before reduction) value of the *cwnd*, and Slow-Starting until *cwnd* reaches to the new value of *ssthresh*.

A simple scenario of detecting a spurious retransmission due to a delayed packet via DSACK is shown in Figure 2.1. In this example, segment 2 is reordered due to a delay in the network, while segments 3 to 5 are received in the receiver. The receiver generates three duplicate ACKs for segment 2 with a SACK option included. When the sender receives those duplicate ACKs, it retransmits segment 2 and halves its *cwnd*. After the last duplicate ACK is sent by the receiver, the delayed segment 2 arrives and causes the receiver to generate a cumulative ACK for segment 5. When the retransmitted segment 2 arrives in the receiver, it generates another duplicate ACK with a DSACK option, signalling the sender that the received segment 2 has already been buffered. The sender detects this signalling since the sequence number of ACK itself is higher than the sequence space of the first SACK block.

[6] DSACK can be used to detect spurious retransmissions due to packet reordering, lost ACKs or early retransmit timeouts [25]. In this thesis we are only concerned with detecting spurious retransmissions.

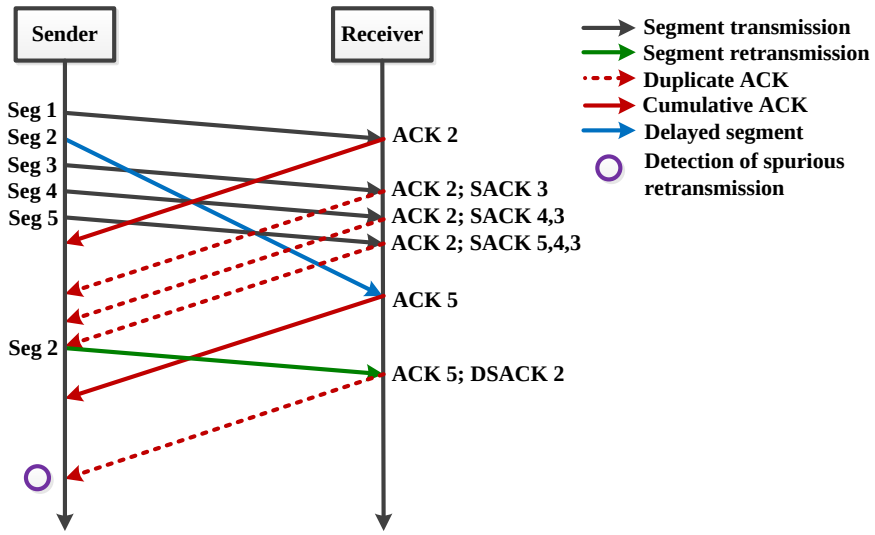


Figure 2.1: A simple scenario for the detection of spurious retransmissions via DSACK

2.2.2 MultiPath TCP

MultiPath TCP (MPTCP) is an extension of TCP [14]. It preserves all TCP semantics such as connection-oriented, reliable and in-order data transmission, but offers the extra feature of parallel data delivery via multiple paths. As with TCP, the key success of the MPTCP protocol arises from its congestion control mechanism, which aims at preserving network fairness among competing flows at bottleneck links [36]. MPTCP is backward compatible with the application layer, allowing applications to remain unaware of the existence of MPTCP in the underlying network stack.

MPTCP has been designed according to the *resource pooling* principle [37]. The goal of resource pooling is to improve resource efficiency by viewing a collection of resources as a single aggregated resource. This design principle allows various traffic demands to be handled more efficiently from a pooled resource than from a number of scattered resources, which would require careful traffic engineering to achieve a similar performance [38]. In other words, any networking design that follows the resource pooling principle significantly helps traffic engineering.

MPTCP follows the resource pooling principle by actively probing the network in order to use any spare capacity in the network. MPTCP achieves this intelligent behaviour via its multipath congestion controller, the so-called *Linked Increases* [29], which effectively shifts traffic from congested to un-congested network paths. This behaviour helps networks to accommodate concentrated surges in traffic and use all

available capacity efficiently. In effect, the MPTCP congestion controller takes on an extra role that is normally associated with routing protocols, i.e. shifting traffic away from congested links and preventing hotspot links in the networks. Furthermore, it strengthens network resilience to failure in individual links/paths by shifting traffic away from those areas, and hence improves overall network throughput significantly.

2.2.2.1 MPTCP Congestion Control

This section explores several multipath congestion control algorithms considered in [36]. The main goal of this section is to obtain a better understanding of the design principles of the MPTCP congestion control algorithm.

2.2.2.2 Uncoupled-TCP

Most prior work which attempts to run TCP over multiple paths use Uncoupled-TCP [26, 39, 27]. The idea is that each subflow of a multipath flow uses TCP congestion control independently from other subflows.

The Uncoupled-TCP algorithm is, in short:

- Each ACK on subflow (s), increases the window w_s by $\frac{1}{w_s}$.
- Each loss, decreases w_s by $\frac{w_s}{2}$.

The only downside to this simple idea is that multipath flows could be excessively aggressive to other competing flows, namely regular TCP flows, so that a multipath flow may get more capacity than other competing flows at a shared bottleneck link. This condition can be clearly imagined when multiple subflows of a multipath flow compete for capacity at a shared bottleneck link. Thus, this solution is not generally fair to competing single-path TCP flows at the bottleneck links.

2.2.2.3 Fully Coupled

Unlike Uncoupled-TCP, this algorithm is fair to competing TCP flows at the bottleneck links. The key idea behind Fully Coupled is that a multipath flow must shift all its traffic onto the least congested path, i.e. the paths with the lowest drop probability. In this way a multipath flow would not get more than its fair share of capacity at a bottleneck link and achieves resource pooling [28, 36, 40].

The Fully Coupled algorithm is, in short:

- Each ACK on subflow s, increases the congestion window w_s by $\frac{1}{w_{total}}$.
- Each loss, decreases w_s by $\frac{w_{total}}{2}$.

Here w_{total} is the summation of all subflows' windows and w_s is bound to be equal to or bigger than one MSS.

A few issues of Fully Coupled is explored in [36, 40]. For example, it keeps little traffic on congested paths so that when these become free of congestion, they cannot be used in a short time frame. Another downside is that the transient fluctuation in congestion tends to make Fully Coupled flap from one path to another. Finally, Fully Coupled may pick an inefficient path when paths have significantly different RTTs, so that it may shift all its traffic to a path with the lowest drop rate even though the RTT of that path is significantly higher than others. Fully Coupled may achieve low throughput in such cases as the TCP throughput is inversely related to RTT.

The intuition is that, the Fully Coupled algorithm can achieve optimal resource pooling when the network is in the steady-state condition, but this can be violated if network condition is highly dynamic, as in the case of the Internet. The issue outlined above may then become apparent [40].

2.2.2.4 Semi-Coupled

This algorithm is an extension of Fully Coupled. The idea is that it is desirable to shift most of the traffic from congested paths to less congested paths, as this behaviour provides network fairness among competing flows at a bottleneck link and equalises loss rate throughout the network [36]. However, keeping a sufficient amount of traffic in congested paths for probing is also vital, as this allows multipath flow to effectively use those perceived congested paths whenever they become free of congestion in short periods of time. This algorithm only changes the increased part of AIMD [36]; the decreased part remaining the same as in the TCP congestion control algorithm. It is argued that network fairness can be completely preserved when both sides of AIMD are coupled [41, 36]. Semi-Coupled is thus more relaxed on network fairness in order to achieve better resource pooling (cutting an individual subflow's window per loss is

more aggressive than cutting the total window per loss).

The Semi-Coupled algorithm is, in short:

- Each ACK on subflow (s), increases the congestion window w_s by $\frac{a}{w_{total}}$.
- Each loss, decreases w_s by $\frac{w_s}{2}$.

Here parameter a is a constant that controls the aggressiveness of multipath flow and tries to maintain a reasonable amount of traffic on each path while shifting some to the least congested paths [36]. The main shortcoming of this modification is that constant a may not be an efficient choice in a wide range of network conditions. For example, if a is very large then the Semi-Coupled performs similarly to the Uncoupled-TCP algorithm. If it is very small, then the behaviour of the Semi-Coupled effectively becomes similar to that of the Fully Coupled algorithm. Thus, dynamic adjustment of a based on an explicit measurement of network conditions seems to be the right approach. In other words, it is desirable to act like Uncoupled-TCP whenever paths are empty in order to use network capacity efficiently.

2.2.2.5 Linked Increases

The MPTCP congestion control proposed in [36], known as Linked Increases or RTT Compensator, tries to find an appropriate value for a as the network condition changes. The value of a is calculated by considering RTTs and the window sizes of all subflows.

The Linked Increases algorithm is, in short:

- Each ACK on subflow (s), increases the congestion window w_s by $min(\frac{a}{w_{total}}, \frac{1}{w_s})$.
- Each loss, decreases w_s by $\frac{w_s}{2}$.

The value for a can be calculated by the following formula:

$$a = w_{total} \frac{max_r \left(w_r / rtt_r^2\right)}{\left(\sum_r (w_r / rtt_r)\right)^2} \tag{2.1}$$

Here max_r means the maximum value for any possible value of r, and $\sum_r$ means the sum for all possible values of r.

The main differences between Linked Increases and the Semi-Coupled algorithm is the Linked Increases has a cap of $1/w_s$ on its aggressiveness. This means that each subflow of MPTCP cannot be more aggressive than TCP congestion control operating on the same subflow, i.e. the window increase would not be more than one segment per RTT in each subflow. This ensures that a MPTCP flow would not unduly harm competing flows at a bottleneck link. The main reason to add this cap is that when a MPTCP flow encounters a path with high RTT and low packet drop, it increases its aggressiveness ($a > 1$) and behaves like Uncoupled-TCP in order to compensate from that path with high RTT so that it can achieve high aggregated throughput. If there is no cap, therefore, other paths, possibly those with low RTTs and packet drop probability, may become extremely aggressive and unfair to other competing flows at shared bottleneck links. Adding this cap thus ensures that a MPTCP flow is not exceedingly harming other competing flows in such scenarios.

2.3 Data Centre Networks

2.3.1 Applications and Services

Data centres typically consist of tens of thousands of networked computers that provide *services* to *cloud applications*. Cloud applications have become an important part of our day-to-day activities. Examples of such cloud applications are Gmail, Google Search, DropBox, Hotmail and Facebook. Examples of such distributed services are MapReduce [42], Hadoop [43], Google File System [44], Google Bigtable [45], Amazon Dynamo [46] and Microsoft Dryad [47].

The main goal of any data centre design has been to provide an infrastructure that is highly available, and offers highly performant computing and high capacity storage while using cheap commodity hardware [17]. It has been claimed that the Google search engine can respond to a single query in fraction of a second (e.g. less than 0.5s) and the result is retrieved by coordinating more than 1000 servers simultaneously [48].

Today, many cloud services are required to exchange information with remote nodes inside data centres to process with their local computations. For example, typical MapReduce computations [42] require an all-to-all communication pattern between servers inside a data centre, i.e. a subset of data centre servers must communicate with one another simultaneously in order to transport the output of the ‘map’ phase before

proceeding with its 'reduce' phase. Another prevalent example of a service requiring such intra-domain communication is an online search query, which essentially requires massive parallel communication with servers storing an inverted index in order to return the most relevant results.

Intra-domain communications, which are rapidly growing, pose a major challenge in today's data centres as they are not designed to handle such communication patterns. Facebook, for example, has stated that its intra-domain traffic is several orders of magnitude larger than what goes out to the Internet [49].

Furthermore, services present different traffic patterns with different bandwidth and communication requirements. Some applications are latency-sensitive, some others are bandwidth hungry and others are both. Servers do not only run a single service; it is common to have multiple services running on the same server. This implies that servers need to handle diverse traffic matrices and unpredictable traffic patterns in which traffic engineering becomes a very complex and challenging task, especially as the size of data centres increases.

2.3.2 Network Topologies

A conventional data centre network topology typically consists of switches and routers in a hierarchical tree structure of two or three levels, referred as a *single-rooted* tree. The 3-tiered network topology has a *core* layer at the top, an *aggregation* layer in the middle and an *edge/access* layer at the bottom of the hierarchical tree structure. A 2-tiered network topology consists of only *core* and *edge* layers. A 2-tiered network topology can support up to 8K servers and a 3-tiered network topology supports more than 10K servers, assuming that the core switches are equipped with more than 120 ports of 10Gbps [5]. The size of these topologies is constrained by the capacity provided in the network elements at the top tier of the topology.

Tree-like network topologies are common in data centres since they can be easily built, administered and expanded [49]. Figure 2.2 shows an example of a 3-tiered design. In this topology, typically 20-40 servers are located in each rack, and are connected to a Top-of-Rack (ToR) switch via 1Gbps links. ToR switches are connected to Aggregation switches via 10Gbps links. Aggregation switches are aggregated further up and connected to core switches via 10 Gbps links.

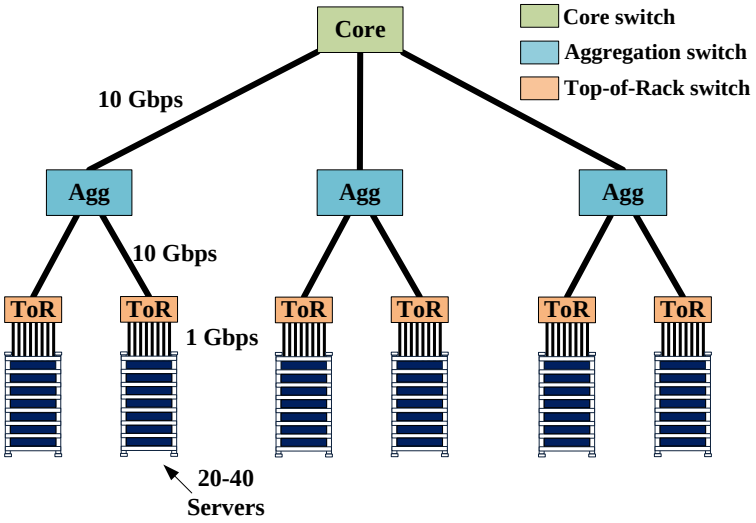


Figure 2.2: Conventional data centre topology

Conventional data centre network topologies do not provide full bisection bandwidth between all servers since core switches do not have a sufficient number of ports to aggregate enough capacity to guarantee that all servers can communicate with one another at their line rate at any point in time. Achieving full bisection bandwidth, although possible, significantly increases deployment costs, even for small data centres. Data centre owners prefer to design their networks with some oversubscription factors, decreasing the total cost of the deployment.[7]

For example, if each rack in Figure 2.2 has five servers then those servers can communicate with one another via any traffic matrix at line rate since the provided capacity in each tier of topology is equal or larger than the aggregated capacity in the host layer. That is, the network provides full bisectional bandwidth among servers and all links in the network have an oversubscription ratio of 1:1. However, in practice each rack typically includes between 20 and 40 servers, and some have more. The links between ToR and Aggregation switches may therefore have an oversubscription ratio of more than 2:1 and between Aggregation and Core switches there may be an oversubscription ratio of more than 8:1. In general by going up the topology, links have a higher oversubscription ratio since the switches support higher statistical multiplexing.

[7] An oversubscription ratio of 1:1 means that the network can provide 100% of host bandwidth for all communication patterns (full bisection bandwidth) and an oversubscription ratio of 5:1 means that the network can only provide 20% of host bandwidth for some communication patterns [50].

2.3.3 Traffic Patterns

Data centre traffic can be categorised into two distinct groups, as follows:

1. **External traffic.** Traffic flowing between computers connected to the Internet and servers inside data centres; or traffic flowing between servers of different data centres.

2. **Internal traffic.** Traffic flowing between servers inside a data centre. Most data centre traffic today originates within data centres and is growing very fast. Intra-data centre traffic can be further categorised into two distinct groups, as follows:

   (a) **Intra-Rack.** Traffic flowing between servers connected to the same ToR switch. Servers may be able to communicate to one another as fast as their maximum line rate's capacity.

   (b) **Inter-Rack.** Traffic flowing between servers located in distinct ToR switches. This type of traffic can be divided into two further categories, as follows:

      i. Traffic circulating between the racks of servers connected via the same Aggregation switch.

      ii. Traffic circulating between the racks of servers connected via a Core switch.

In conventional data centres, traffic engineering can be very complex and difficult to manage, especially when the networks are very large. Each of the internal traffic patterns described above could be derived from different applications with various bandwidth requirements. Thus, the location of applications/services is not typically random; instead, it should be engineered very carefully, based on expected communication patterns [4, 5].

An example of an application that requires careful traffic engineering is web-searching. Many large-scale online applications leverage *partition/aggregate* workflows [17], in which an aggregator node divides computations across many workers, so that computations can be performed in parallel. The result of each computation can be returned back to the aggregator simultaneously. The aggregator node combines these

results and returns the final result to the requester [17]. Figure 2.3 shows the partition/aggregate workflow. The key traffic engineering challenge is that an aggregator node has a deadline in its response time to a search query (e.g. as small as 200ms). This implies that the workers may have to deliver their computation results to the aggregator within a short timescale (e.g. as little as 10ms). Any delivery time higher than the defined time from workers may cause the aggregator to discard the results, since the aggregator has already returned the final result to the requester. This obviously wastes network resources and CPU cycles, potentially resulting in lower quality search results [17, 18, 13]. Thus, it seems essential to place aggregator and worker nodes in the same and/or adjacent racks, thereby preventing congestion on network paths connecting these servers to a significant degree. Even light network congestion can result in an increased queuing delay, increasing RTTs by two orders of magnitude [13]. As a result, some flows may miss their deadlines.

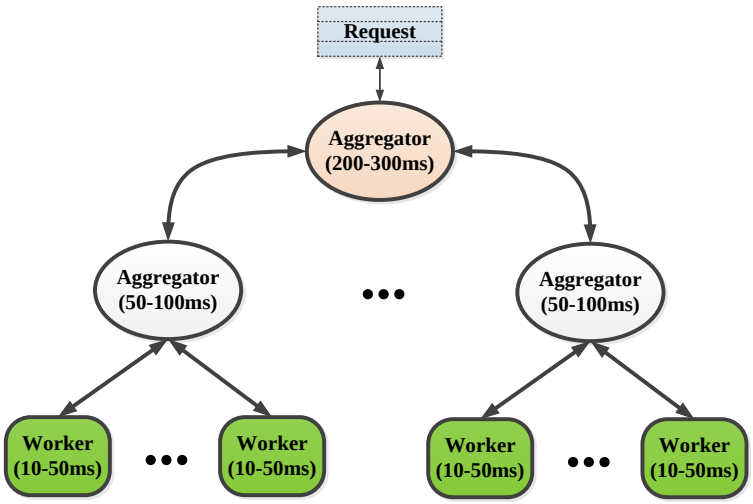


Figure 2.3: The partition/aggregate workflow. Deadlines are showed inside round brackets.

2.3.4 Network Properties

Data centres provide special network environment that are distinct from other networks connected to the Internet [51, 52, 53]. The main notable characteristics are listed below:

- *Short flow dominance.* The majority of flows in data centres are short flows, whereas the minority are long flows. Notably, this minority contributes to the majority of data volume. A recent data centre traffic analysis suggests that 99% of flows are smaller than 100MB but more than 90% of data volume comes from

flows whose size is between 100MB and 1GB [4]. Another traffic analysis shows that the majority of short flows with tight deadlines have a flow size smaller than 1MB (e.g. the query/response flows have a flow size smaller than 50KB [18]), and the long flows have flow size distribution of between 1MB and 50MB. It has also been reported that the majority of data comes from flows with size bigger than 10MB [17].

- *Small and high variance inter-arrival time.* The median inter-arrival time for flows at servers is typically less than 30ms [51]. A detailed traffic analysis suggests that the inter-arrival time of the 99th percentile of the query/short flows are around 500ms and the 75th percentile is around 250ms. The 99th percentile of the long flows is around 20s, the 90th percentile is less than 8s and the 50th percentile is 0ms ([17], Figure 3(b)). These results indicates that the long flows have a very high variance and heavy tail inter-arrival time with embedded spikes.

- *Traffic patterns.* These are diverse, highly volatile and very bursty [4, 5, 17, 13, 54], which features, taken together, may create transient congestion throughout the network, regardless of available network capacity. Preventing transient congestion seems a very challenging task in data centres due to the unpredictability of traffic surges, the lack of any proper traffic diffusion mechanism, and path diversity in some parts of the network topology (e.g. the access layer).[8]

- *Low latency.* Latency between servers is in the order of microseconds. A data centre traffic analysis suggests that the typical latency between servers is between 100 to 250μs in the absence of queuing delay and between servers connected to the intra-rack and inter-rack respectively [17, 18, 13].

- *High Bandwidth.* The link rate between switches is typically 10Gbps and between servers and switches is 1Gbps. However, this trend is changing as 10Gbps Ethernet switch is becoming commodity [55]. This implies that the server-switch links is shifting to 10Gbps and the switch-switch links is shifting to 40Gbps or 100Gbps.

[8] One of our goals in this thesis is to decrease the bursty nature of data centres; a detailed discussion of this can be found in Chapter 3.

- *Single administrative domains.* Servers located inside a data centre can commonly trust each other [18] so there is no need, for example, to deploy any Middlebox or FireWall inside a data centre. Middleboxes can interfere with the operation of a TCP connections by, for example, removing TCP options from packets, if the options are not known to them. This implies that any TCP-based transport protocol, such as MPTCP [3], which uses a TCP option for carrying its signalling messages, could be interrupted or corrupted by Middleboxes.

- *Multipath Communication.* Modern data centres provide dense interconnectivity in the network so that servers can communicate with one another via multiple paths. This feature potentially increases the network availability and robustness in case of the network component failures, which is norm rather than exception in the large-scale data centres.

2.4 Traffic Concentration Problem

As has been said earlier, full bisection bandwidth between any pair of servers is not common because of the high cost and hardware limitations at the top level of the *single-rooted* tree topology. It is argued [3] that conventional data centre architectures are sufficient to handle the traffic in and out of data centres, but insufficient for intra-data centre traffic. Such traffic comes from the increasingly popular bandwidth-hungry applications such as video streaming or online file storage. Thus, some paths might be persistently congested due to a lack of path diversity in the network topology, and a lack of multipath support in routings, e.g. Open Shortest Path First (OSPF) routing, and transport protocols (e.g. TCP), to handle diverse traffic matrices more efficiently.

Network congestion occurs at network links but TCP reacts to it by changing its flow rate end-to-end; TCP halves its congestion window each time a packet is dropped. This behaviour not only introduces a negative effect on the throughput of individual flows, but can also hurt the flow completion times of some latency-sensitive applications. Additionally, TCP's reaction to congestion, which originates from a single link en route, is to decrease its data transmission rate. This may lead to the underutilisation of some other potentially uncongested links.

2.4.1 Localising Traffic into Rack

The simplest approach to controlling traffic concentration is to install services into servers connected to the same ToR switch. For example, cloud services that produce long-lived flows are located in servers that are geographically close to each other. The obvious result of this design is to prevent long-lived flows, which are a major cause of persistent congestion, traversing the core layer. This way, not only can the long-lived flows complete their task faster, but the network core does not get overly congested.

In order to concentrate traffic into racks, a Virtual-LAN (VLAN) is usually defined for each service. This brings some nice features to the network. For example, services can achieve nice traffic segregation from one another and the scope of traffic-flooding in a layer-2 domain can be controlled [55, 56, 51, 57]. This approach is not without limitations. For example, network administrators are required to pre-allocate enough servers for a service to deal with difficult-to-predict traffic spikes. Sometimes the traffic demand of a service can suddenly increase to the point where some servers from adjacent racks or elsewhere in the network need to be allocated to that service. This means that VLAN trunks must be configured manually throughout the network in order to extend a VLAN across it[9]. Furthermore, at times of low demand, reserved servers may be idle, wasting valuable network resources.

Lack of flexibility in traffic management and network configuration prevents the localisation of traffic becoming a comprehensive solution to the traffic concentration problems, especially when a data centre network is very large.

In the next two sections, we explore alternative solutions to the traffic concentration problem, which not only attempt to eliminate it, but also help traffic engineering and allow a very large and resilient data centre network to be constructed cheaply.

2.4.2 Full Bisection Bandwidth Topology

The latest data centre network architectures provide very high aggregate intra-domain bandwidth mainly to solve the traffic concentration problem. The design of VL2 [4] and FatTree [5] is based on the hierarchical tree structure topology, referred as *multi-rooted* topology. A relatively small number of high-end core switches at the top level of the data centre topology are replaced by several lower-capacity commodity switches.

[9]VLAN trunk allows multiple VLANs multiplexes to a single switch port.

BCube [6] and DCell [7] are designed based on hypercubes. Both depart from the hierarchical tree approach.

All these new network topologies provide full bisection bandwidth between all servers in the network. This means that a server should theoretically be able to communicate with any other server located anywhere in the network with the maximum capacity of its network interface device.

2.4.2.1 FatTree

FatTree proposes scaling out the network resources at the top level of the tree structure instead of scaling them up [5]. In other words, it attempts to increase network resources by increasing the amount of hardware instead of increasing the capacity of the currently deployed hardware; the latter is an expensive approach.

Figure 2.4 shows a *k-ary* FatTree data centre topology. Each switch has k ports; there are k pods, each containing k switches organised in two layers of $k/2$ switches. There are $(k/2)^2$ core switches organised in group of $k/2$ switches. Each edge switch is directly connected to $k/2$ hosts and $k/2$ aggregation switches on the top. The remaining $k/2$ ports of each aggregation switch are directly connected to k/2 core switches. Each core switch has one connection to each pod.

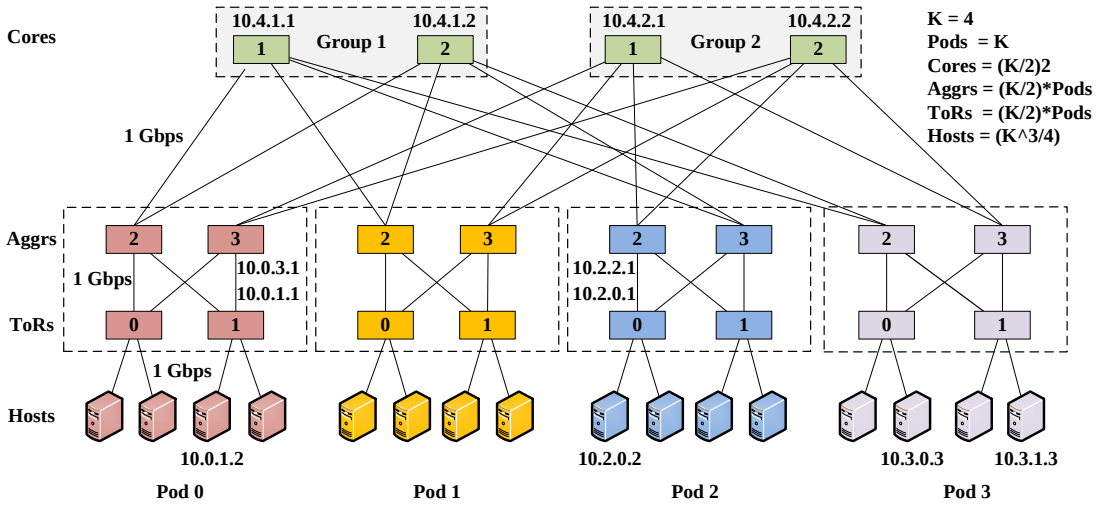


Figure 2.4: A FatTree network topology with 16 nodes

A *k-ary* FatTree provides total hosts of $k^3/4$ and total equal-cost paths of $(k/2)^2$ between all servers communicate via the core layer.[10] For example, in *4-ary* FatTree,

[10]OSPF2 is a common routing protocol in data centre networks and its metric for calculating equal-cost paths to a destination is typically the hop-count [58].

the total number of hosts is 16 and the total number of equal-cost paths is 4 for those servers communicate from distinct pods.

All links in a FatTree topology have identical capacities (1Gbps) and all switching/routing elements are identical. This is the main reason a FatTree topology is much cheaper than conventional data centre topologies that require high-end switches at its top layer.

FatTree designers considered some commodity switches that did not support Equal-Cost Multi-Path (ECMP) [9] technology on top of the OSPF2 routing [58]. Without ECMP, all path redundancies present in a FatTree cannot be used. To solve this issue, a new network-addressing scheme and routing mechanism have been proposed in order to diffuse traffic similarly to ECMP [5].[11] The FatTree addressing for different network elements located at different layers of the topology, are as follows:

- **Core switches.** IP address of switches at the core layer have a pattern of *10.k.j.i*, where k denotes the number of switch port, j denotes the group a core switch belongs to (e.g. *k-ary* has two groups of $k/2$ core switches) and i denotes the location of the core in the group (left to right, starting from 1).

- **Intermediate switches.** The IP address of switches in a *pod* has an addressing pattern of *10.pod.switch.1*, where *pod* indicates the pod number, *switch* indicates the position of that switch in the *pod* (starting from left to right, bottom to top).

- **Hosts.** The IP address of hosts has an addressing pattern of *10.pod.switch.ID*, where *ID* is the host position in the subnet starting left to right.

This IP addressing scheme can be used with any routing mechanism, e.g. OSPF-ECMP routing. The key advantage of this scheme is that each networking element (hosts and switches) can simply infer the location of any other networking element via its IP address.

2.4.2.2 VL2

VL2 topology is very similar to FatTree. VL2 attempts to solve the traffic concentration problem discussed in Section 2.4. In VL2, the links between Aggregation and Core

[11] A detailed discussion of how their new routing mechanism operates can be found in [5].

switches form a complete bipartite graph, i.e. each Aggregation switch has a connection to every Core switch, as shown in Figure 2.5.[12] VL2 supports extremely dense interconnection in the core layer, which makes the topology look like a mesh rather than a tree structure.

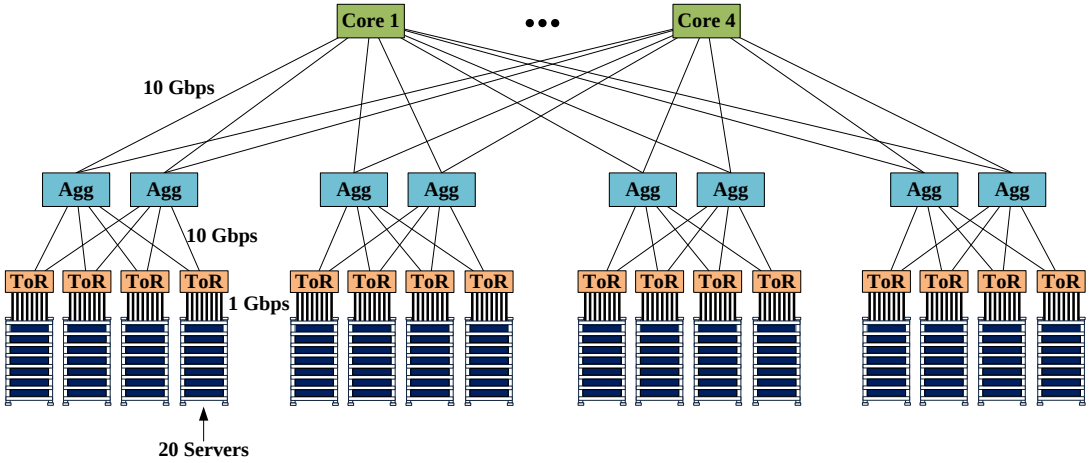


Figure 2.5: A VL2 network topology with 320 nodes

Similar to conventional data centre topologies, VL2 includes racks of 20 servers connected to Top-of-Rack (ToR) switches via a 1Gbps links. ToR switches are connected to two aggregation switches via uplinks of 10Gbps. Finally each aggregation switch is connected to every core switch via a 10 Gbps link.

VL2 permits any cloud service to be installed on any server located anywhere in the network. To achieve this goal, VL2 uses a flat addressing scheme and separates the server IP addresses (the so-called Application Address - AA) from its locations (the so-called Location Address - LA). The LAs are in the same IP subnet and include all switches (e.g. 10.0.0.0/8 subnet). The AAs are in a distinct IP subnet and include all hosts (e.g. 20.0.0.0/8 subnet). The edge switches are the bridge between LAs and AAs. Hosts locate each other via a central directory service, which holds a mapping between AAs and LAs. The routing protocol in LAs is OSPF2-ECMP.

Unlike in FatTree, servers in VL2 are involved in packet forwarding since VL2 uses Valiant Load Balancing (VLB) to route its traffic.[13] The core idea in VLB routing is to route packets via two randomised stages so that a packet can reach its destination via two randomised paths. The first random path is from a source node to an interme-

[12] In FatTree, each Aggregation switch has connections to a group of core switches.
[13] VLB is also referred to as *two-phase routing* or *randomised load-balancing*.

diate node, and the second random path is from an intermediate node to a destination.

VLB is agnostic to traffic matrices since the two-stage per-packet forwarding erases the traffic pattern, so different traffic patterns can be disseminated throughout the network in the same manner [59, 60].

VLB on per-packet basis is widely used in router switch fabrics in order to remove the need for a centralised scheduler [60], and hence improve their scalability. But in VL2, it has been adopted to be performed on per-flow basis as to prevent reordering of the TCP packets. However, the two-stage routing approach with per-flow forwarding still seems to provide a fine-grain randomisation between flows in data centres and is a well-suited approach for dealing with the volatility of traffic patterns in today's data centre networks.

In order to achieve per-flow, two-stage routing, VL2 proposed using multiple IP encapsulations via the hashing of standard five-tuple. That is, each packet has a multiply encapsulated IP header. A packet's first destination is a random core switch. When it arrives, it will be de-capsulated and forwarded towards the next destination address, which is a ToR switch of the final destination. In turn, a packet will be switched from that ToR switch to the destination server.

VLB not only achieves randomisation by utilising two-stage routing but also by using ECMP; the packets will be routed from a server to a random core switch (intermediate node) and from there to destination ToR switches via ECMP routing.

ECMP is an option that can be enabled on router or layer-3 switches, on top of a link-state routing protocol such as OSPF2. The link-state routing protocol provides a set of equal-cost paths in a given subnet and ECMP makes a forwarding decision.[14] ECMP takes this decision as follows: when a packet with multiple candidate paths arrives, it first calculates a hash of the standard five-tuple (src IP, dst IP, src port, dst port, protocol number), then uses a *modulo* operation between the hashing result and the number of candidate paths to the next hob; thus splitting the load to each subnet across all possible paths. If the number of equal-cost paths remains unchanged throughout the data delivery of a flow, ECMP thus forwards all a flow's packets via the same path, thereby maintaining their arrival order.

The key limitation of two-stage routing in VL2 is that all traffic that is not destined

[14] In a link-state routing each layer-3 switch has the knowledge of the entire network topology.

to the same ToR switch bounces the core layer and routes back to the destinations, even though the source and destination are connected to the same aggregation switch. This behaviour may create some unnecessary congestion in the core layer of the network. Furthermore, it has been argued that per-flow VLB becomes effectively equivalent to per-flow ECMP [8].

A key limitation of ECMP is that two or more large and long-lived flows can collide on their hash and end up on the same link [8], creating a preventable network congestion. In other words, ECMP cannot provide disjoint paths between flows.

In VL2, the number of these elephant flows should be more than 10 to create congestion at the core/aggregation layer since the links in those layers are 10 times faster than the links in the edge layer. However, in a topology like FatTree that has 1Gbps capacity for all links, two long-lived flows alone are enough to create heavy congestion in any link anywhere in the network.

VL2 and FatTree offer nice features to the data centre network such as scalability, fault tolerance, agility and full bisection bandwidth. However, the key question remains: does VL2 or FatTree completely solve the traffic concentration problem?

Even though these topologies provide full bisection bandwidth in the physical layer, the networking protocols such as OSPF-ECMP and TCP, which are principally designed to operate on a single path, fail to effectively use these large capacities. The traffic concentration problem might therefore still occur, which not only wastes network resources but also degrades overall network performance in the presence of large aggregated capacity in the network.

In the next section, we discuss some other solutions for solving the traffic concentration problem which takes approaches other than providing a full bisection bandwidth topology. The belief common to these solutions is that providing a full bisection bandwidth seems overkill since not all servers in large-scale data centres require high capacity communication, and hence large aggregated capacity in the core of the network. This implies that a full bisection bandwidth topology is a waste of network resources and money [3, 61, 62].

2.4.3 Dynamic Capacity Allocation

In the previous section, we discussed FatTree and VL2 topologies which attempt to solve the traffic concentration issue by leveraging new network topologies which provide dense connectivity in the backbone of the network, as well as full bisection bandwidth between any pair of servers.

In this section, we explore different approaches for mitigating the traffic concentration problem. The main motivation in these approaches is that the full bisection bandwidth solution is an overkill approach. Given that only a few servers in the entire data centre are responsible for causing the traffic concentrations, there is no need to provide very large aggregated capacity in the network core only for that small subset that runs a traffic matrix which requires high capacity communication [62, 61, 12]. Furthermore, commonly used networking protocols, such as ECMP and TCP, are not capable of efficiently using the available capacity in a full bisection bandwidth topology [8, 12].

Flyways [62] and c-Through [61] attempt to solve the traffic concentration problem by providing on-demand extra capacity between ToR switches, which are handling long-lived flows. A clear intuition for both approaches is that the non-uniform traffic matrices, which originate from a few servers running long-lived flows, are better to be handled by the non-uniform network topology than the full bisection bandwidth topology. In other words, the full bisection bandwidth topology provides unnecessarily high capacity between all pairs of servers because only a few pair of servers require such a high capacity communication.

These solutions do not change the conventional data centre network topology, meaning that other existing problems inherent in a conventional topology, such as scalability, fault tolerance and network agility [4], are still present. Similar to the full bisection bandwidth approach, these solutions partly solve the traffic concentration problem; perhaps with more limitations. We further discuss these matters in the next subsections.

2.4.3.1 Flyways

The design of Flyways [62] is based on the observation that only a few ToR switches typically communicate with one another extensively at any point in time. In other words, a subset of ToR switches is handling long-lived flows at any point in time. If this pattern of communication can be detected, some short-cut links between those ToR

switches can be set up, so that a part of the traffic between those ToRs can be re-routed into these short-cut links. Thus, these short-cut links prevent long-lived flows from traversing several links in the network core.

In order to detect these hotspot ToR switches, Flyways proposes a central controller that monitors the traffic on all ToR switches via Simple Network Management Protocol (SNMP) [63]. When the central controller detects a hotspot ToR switch, it establishes a wireless connection between the hotspot ToR switches and re-routes part of their traffic via this extra capacity.

Flyways proposes installing a wireless radio of 60 GHz on top of each ToR switch, which provides up to 1Gbps capacity. Full capacity between two wireless radios can be achieved only when a path between two wireless antennas is in line of sight and not more than 10 metres away. In conventional data centres, 10 metres cover a significant number of racks/ToR switches, but not all. For example, one ToR switch may need to communicate with another one that is out of its wireless coverage. This limitation is shown in Figure 2.6, the red wireless access point cannot reach to the rightmost wireless access point. Furthermore, detecting hotspot ToR switches and establishing wireless links between them introduces a significant delay, which imposes a limit on the number of traffic matrices that can be supported. Only long-lived and loosely synchronised flows can therefore effectively use these surplus capacities.

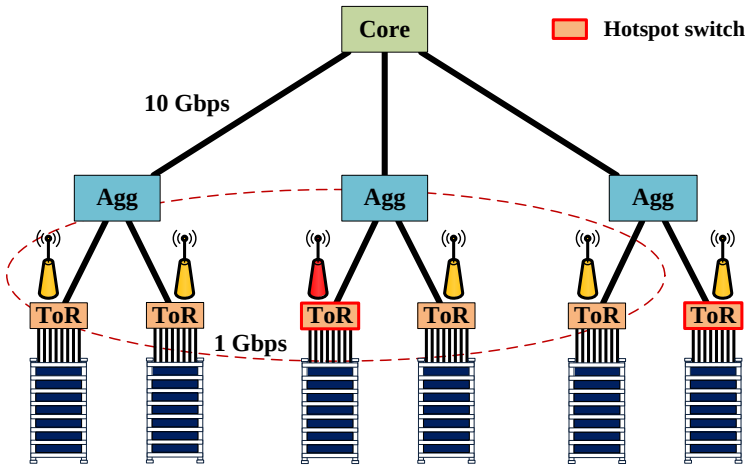


Figure 2.6: Flyways

Flyways can also be implemented via wired links. Flyways proposes connecting ToR switches via commodity switches with a large number of 1Gbps ports. This solution entails less delay than the wireless link scenario but has its own limitations. For

example, the number of ToR switches that can communicate with one another merely depends on the number of switch ports.

2.4.3.2 c-Through

C-Through [61], like Flyways, attempts to overcome the traffic concentration problem via short-cut links in the conventional data centre topology. The c-Through idea is to interconnect a large variety of ToR switches via an optical switch, as shown in Figure 2.7. C-Through is quite similar to the wired version of Flyways.

An optical switch provides a very large dedicated capacity between a pair of ToR switches by establishing an optical circuit. In other words, a pair of ToR switches can be connected via a dedicated communication channel, which provides full bandwidth of the channel over the course of the communication session.

An optical switch establishes the circuits in sequence. At any given point in time, only one pair of ToR switches can communicate; one circuit at a time is established between consecutive pairs of ToR switches. As can be seen in Figure 2.7, the leftmost ToR can only communicate simultaneously with the rightmost ToR. This implies that some traffic matrices which require all-in-all communication patterns, such as MapReduce data processing, are not suited to optical switches since establishing a circuit introduces significant delays that MapReduce and similar traffic patterns cannot tolerate.

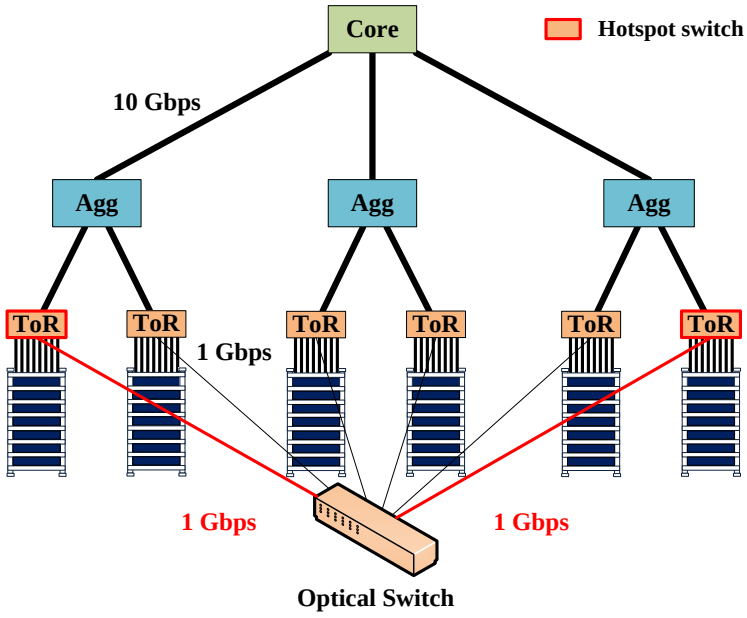


Figure 2.7: c-Through

In c-Through, the optical network is separated from the base network using a sep-

arate VLAN. That is, latency-sensitive and highly-synchronised traffic can be handled from the base network and traffic matrices that are bandwidth hungry, loosely synchronised or bulk data transmission, such as virtual machine migration, can be handled from the optical network and with high data transmission (e.g. 1 Gbps).

There are several limitations to the c-Through approach. For example, optical switches are very expensive, their energy consumption is very high and they can only support a limited number of switch ports (e.g. up to 320 ports of 1Gbps) [61]. Similarly to Flyways, the limitation on the switch port number makes this solution appropriate to a small variety of communication patterns. Finally, the circuit establishments and reconfigurations introduce some overheads and delays, which make this approach well-suited only to a small number of traffic matrices.

2.5 Equal-Cost Multi-Path Routing

Modern data centres, such as VL2 [4] and FatTree [5], shift their architecture from a *single-rooted* to a *multi-rooted* hierarchical tree structure [8]. This provides dense connectivity as well as full bisection bandwidth between all pairs of hosts [55, 5, 4]. However, commonly-used routing protocols in data centres are designed to select a single optimal path for each flow. Even state-of-the-art forwarding mechanisms such as Equal-Cost Multi-Path (ECMP) [9], which stripe flows across available paths, can only partially prevent persistent congestion.

The limitation of ECMP is due to its static flow-base hashing in which traffic is randomised without considering traffic matrices and network conditions. Thus, long-lived flows may be placed in already loaded links although there could be lightly loaded links elsewhere in the network [8, 12]. In other words, ECMP may obliviously place several long-lived flows into the same link, resulting in the overloading of switch buffers and the lowering of overall network utilisation. This limitation is highlighted in Figure 2.8. The first collision is between two long-lived flows (A and B) at an output link of Agg-1. The second collision is between two long flows (C and D) at an output link of Core-3. The result of these collisions is that all four flows achieve a half in their maximum theoretical throughput (i.e. 500MB instead of 1Gbps), even though they could simply achieve 100% throughput; if Flow-A uses Agg-2 and Flow-C uses Agg-7.

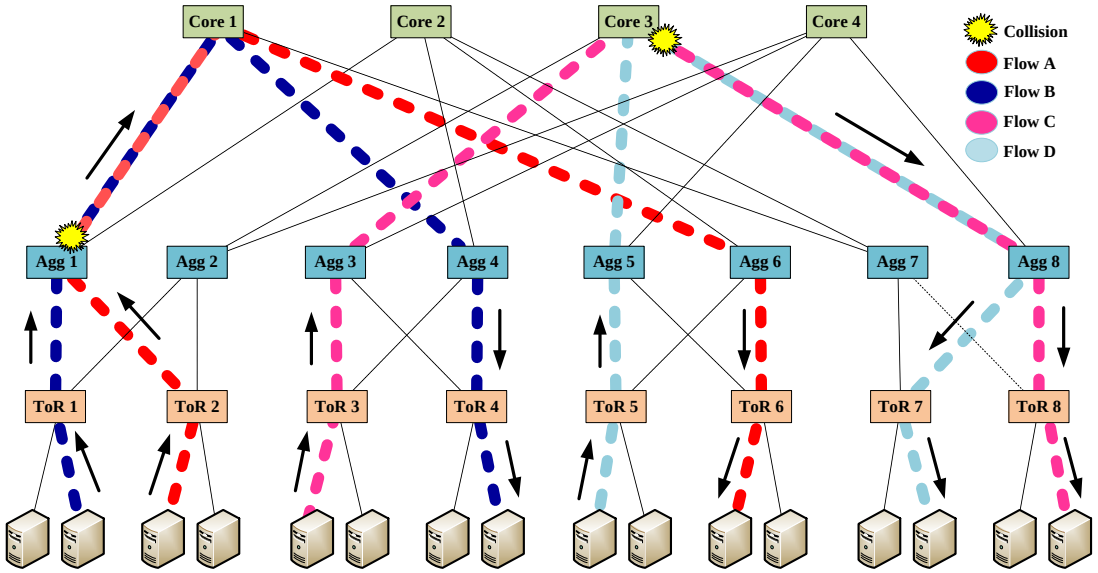


Figure 2.8: ECMP hash collision. Flows A and B are collided on an outgoing link of Agg-1; Flows C and D are collided on an outgoing link of Core-3. Each flow thus gets a half of its maximum connection throughput.

In this section, we discuss Hedera [8] and MPTCP [12], two approches that effectively complement ECMP routing. These approaches compensate for ECMP's limitations, and hence improve overall network utilisation. The common ground of both approaches is to balance traffic dynamically and intelligently by actively monitoring network conditions so that traffic can be removed from highly loaded paths/links and placed in lightly loaded paths/links. The main difference between the two solutions is that MPTCP detects and reacts to congestion on a per-path basis, whereas Hedera reacts per-link.

2.5.1 Central Flow Scheduler

Hedera [8] attempts to improve the overall network throughput by employing a central load-aware scheduler, which dynamically and adaptively balances long-lived flows across *multi-rooted* tree topologies. The main job of the central scheduler is to detect and re-route the long-lived flows from highly-loaded to lightly-loaded links, if possible.

The central scheduler has a control loop consisting of three main steps. First, it collects statistics from all switches, essentially measuring the utilisation of all links in the network, so that the central scheduler has knowledge of all the network link utilisations. That is, it can simply detect the long flows at the ToR switches. It then uses placement algorithms to find alternative paths for those long flows. Finally, it

re-routes the traffic by installing new paths on the programmable switches, such as OpenFlow switches [11]).

Hedera detects long flows when a flow exceeds a pre-defined threshold rate of 100Mbps at a ToR switch (i.e. 10% of each host's link rate), so an alternative path for that flow may be offered. If a placement algorithm offers a new path, the routing table of all switches along that path is modified.

Hedera's fundamental constraint is that the central scheduler is limited in its reaction time, since it has to gather statistics, compute and finally instantiate new paths, all in a very short timescale [12]. The central scheduler's reaction time is intrinsically increased when the total number of long flows is increased; this is particularly crucial in large-scale data centres which need to accommodate more than 100K servers handling a vast number of long flows at any point in time.

Hedera categorises network flows in two groups: *host-limited* and *network-limited* flows. The performance of a *network-limited* flow is only limited by the network congestion in its assigned path; if there is no congestion, then a network-limited flow potentially saturate its network interface device. Performance bottlenecks of *host-limited* flows mostly originate from limitations on hosts in a connection, e.g. those imposed by disk speed, memory or CPU processing.

Hedera's placement algorithms only schedule *network-limited* flows and ignores other flows. Additionally, it assumes that scheduled flows onto lightly loaded links are capable of saturating those new links over time. However, both assumptions can be infringed by *host-limited* flows [12].

To be a *host-limited* flow neither implies a long-lived flow nor the production of the persistent congestion in the network. In other words, *host-limited* flows can be long-lived and create persistent congestion in their paths, although they can be hidden from the central scheduler since they may never exceed the scheduling threshold rate. That is, they may collide with scheduled flows or new long-lived flows, creating persistent congestion on those paths until the scheduler detects this and reacts by rerouting long-lived flows away from those paths, hurting the overall network throughput.

Hedera has shown performance improvement in FatTree data centre topologies under the following assumptions: all flows in the data centre are network-limited, their sizes are exponentially distributed, and their arrivals follow the Poisson process [12, 8].

However, data centre traffic analyses show that flow distributions are not exponentially distributed [4]. In such cases, it has been argued that the scheduler needs to run much faster than Hedera's suggestion of 5 seconds in order to deal with flow arrivals effectively [12]. In other words, the scheduler needs to detect and mitigate possible congestion in a very short timescale in the high bandwidth and low-latency networks, where traffic patterns are diverse, volatile and unpredictable.

It has also been shown that Hedera, with a scheduling circle of 500ms, can only achieve similar performance to randomised load-balancing mechanisms [12].

2.5.2 MultiPath TCP

It has been argued that single-path TCP is not well-suited in *multi-rooted* network topologies that provide dense interconnectivity in the network core [12]. MPTCP essentially offers an alternative solution to the central load-aware approach (Hedera [8]).

The key advantages of MPTCP compared to Hedera are as follows:

- *Reaction Time:* MPTCP responds to network congestion in a few RTTs (i.e. in order of microsecond). Thus, MPTCP seems well-suited for dealing with the dynamic nature of large data centres, which have diverse, volatile and unpredictable traffic patterns [4, 3, 12]. It has been shown that Hedera needs to run every 100ms or less in order to achieve performance close to that of MPTCP [12].

- *Decentralised.* MPTCP inherits the distributed and reactive nature of the TCP protocol. It delivers data on different paths and responds to congestion across those paths end-to-end. This implies that MPTCP does not suffer from scalability limitations nor the existence of a single point failure, which are inherent in any centralised approach. The distributed and reactive nature of MPTCP is vital for data centres since it is almost unfeasible for any central scheduler to have precise information about all flows due to the highly dynamic nature of data centres. This implies that any network scheduling scheme must work with less precise information about the actual network condition at the time of scheduling [64].

- *Intelligent load balancing.* Unlike Hedera, MPTCP detects/reacts to congestion by adjusting/balancing its traffic across its subflows. In other words, MPTCP can

consistently achieve high throughput for *host-limited* and *network-limited* flows as it responds to congestion when they are detected.

It has been shown that the performance improvement of MPTCP also depends on the network topology. For example, MPTCP performs only slightly better than TCP in VL2 topologies, but significantly better in FatTree and BCube ones [12]. MPTCP slightly outperforms TCP in VL2 because links between switches are 10x faster than between hosts and switches. That is, VL2 handles ECMP collisions more gracefully. Additionally, data centres have low statistical multiplexing for the long-lived flows [17, 4], so congesting the network core of VL2 need at least 11 long-lived flows competing at a bottleneck link.

Replacing TCP with MPTCP is not a straightforward process since all applications in the current data centres are designed within a TCP mindset and are unaware of the existence of MPTCP. That is, they cannot dictate whether to use MPTCP or TCP based on their communication requirements. Furthermore, data centre analysis showed that the majority of flows in data centres are short and contain only a few packets [17, 4].

In Section 1, we presented a simulation to highlight that a large MPTCP flow with eight subflows is sufficient to improve the overall network performance significantly in modern data centres [12]. However, it is problematic for running short flows as it damages their flow completion times. The fundamental problem is quite simple: when a MPTCP flow with eight subflows is being used for a short flow with only a few packets then the data is delivered in a few RTTs. This means that the congestion window of each subflow may only increase by as little as a few segments. In such cases, even if a single packet gets dropped from a subflow, the loss recovery may only need to be performed via a retransmission timer since there are not enough packets in flight to generate enough duplicate ACKs to trigger the fast retransmission mechanism.

We find ourselves at an impasse: MPTCP is damaging short flows while a majority of data centre flows are short-lived [4, 3]. However, MPTCP is better-suited to dealing with the dynamic nature of data centres than single-path TCP, which needs to be complemented with a slow central scheduler.

This deadlock becomes one of our incentives to design our novel transport protocol, named MMPTCP, with the intention of complementing MPTCP and allowing it to operate well with any type of flows.

2.6 Short Flow and Deadline

2.6.1 DCTCP

Data Center TCP (DCTCP) is an extension of TCP with a novel congestion control mechanism [17]. The main goal of DCTCP is to keep buffer occupancy of data centres' switches low. That is, latency-sensitive flows can be handled without excessive queuing delay or timeout when competing with long-lived flows at a shared bottleneck link.[15] Long-lived flows are the main reason for a switch's buffer occupancy and hence queuing delay. DCTCP attempts to maintain low-buffer occupancy without excessively hurting the throughput of long-lived flows. As a result, the surge of traffic, especially from synchronised short flows, into a single output port of a switch, can be handled to a great extent. In effect, DCTCP allows a greater fraction of latency-sensitive/mice flows to meet their deadline than legacy TCP, by experiencing fewer queuing delays and costly packet losses, which can lead to timeouts.

DCTCP assumes that data centre switches support a simple Active Queue Management (AQM), such as RED [65], with Explicit Congestion Notification (ECN) capability [66]. The idea of RED-ECN is that network switches monitor their queue sizes (output ports) and mark packets with Congestion Experienced (CE) bits when the queue occupancy exceeds a predefined threshold, but has not yet overflowed. When the receiver receives a data packet with the CE code bits set in the IP packet header, the receiver sets the CE code bits in the corresponding ACK packet and sends it back to the sender. Thus, the sender can react to potential congestion en route, and prevent packet drops and buffer overflows.

DCTCP achieves its goals by employing a novel ECN-based congestion control mechanism that allows the sender to react in proportion to the extent of congestion.[16] In other words, DCTCP's reaction to congestion is based on feedback from congested switches, meaning that the DCTCP sender reduces its congestion window in proportion to the severity of congestion. For example, DCTCP reacts similarly to TCP by halv-

[15]The commodity switches used in data centres typically have shallow buffer and shared-memory.

[16]Random Early Detection (RED) [65] queue discipline with ECN is an example of AQM that it is suggested should be used in conjunction with DCTCP by simple parameter adjustment: setting both the low and high marking thresholds of RED to a single DCTCP making threshold (K). Additionally, the marking of packets needs to be performed based on an instantaneous queue size instead of the average queue size considered in the RED scheme. In this way, DCTCP can signal the congestion state to the sender precisely and promptly.

ing its congestion window if the fraction of marked ACKs is excessive; if the sender receives a few marked ACKs, a small fraction of its congestion window is reduced.

TCP reacts to an ECN signal by halving its congestion window. The consequence of this reaction is a significant degradation of the overall network throughput due to buffer underflows en route: several TCP flows competing at a shared bottleneck link halve their data transmission. However, in such cases DCTCP allows multiple competing flows at a shared bottleneck link to adjust their transmission rates, so that the buffer occupancy of switches can be consistently maintained near the marking threshold. It is claimed that DCTCP can theoretically maintain a throughput of more than 94% with zero buffer occupancy [67, 17].

The DCTCP congestion control algorithm is, in short:

- Each ACK increases the window w to $w + \frac{1}{w}$.
- Each marked ACK decreases w to $w \times (1 - \alpha/2)$.

Parameter α is an estimation of the fraction of marked packets and is calculated for every window of data with the following formula:

$$\alpha = (1 - g) \times \alpha + g \times F$$

Parameter F is the fraction of marked packets in the last window of data, and g is a constant weight factor given to new samples against the past ($0 < g < 1$). The suggested value for g is $1/16$ [17]. When α is close to 0, DCTCP reduces its window w very gently, meaning that DCTCP reacts smoothly to the initial congestion signal. However, like TCP, DCTCP cuts its window w at most once per window of data (i.e. once per RTT) [66, 17]. This implies that when a congestion signal is detected, DCTCP reduces its window w gently and starts collecting the rest of the marked ACKs without any further reduction in its window. In the next window of data, the reaction of DCTCP effectively becomes similar to TCP, if a large fraction of marked packets have been received in the previous window of data ($\alpha \to 1$). In other words, DCTCP probes a congestion link over a few RTTs after the arrival of the initial marked ACK, by updating α per RTT, and adjusts its congestion window proportionally to the congestion level in order to prevent buffer overflow and hence possible timeouts. That is, it maintains the buffer size of the congested link around its marking threshold.

The obvious limitation of DCTCP in comparison with TCP is its convergence time; i.e. the time required for a new DCTCP flow to get its fair share of capacity from competing flows, possibly with large window sizes, in a bottleneck link. In other words, DCTCP takes longer than legacy TCP to get its fair share of capacity from the network (e.g. a factor of 2-3 times more than TCP, if links are 1Gbps/10Gbps and RTTs are ranging from 100-300μs [17]). The main reason is that the DCTCP sender reacts to perceived congestion gently and must reduce its congestion window incrementally over a few RTTs. This may be crucial for some bandwidth-hungry applications whose task completion time is dependent on the slowest flows. This also implies that DCTCP does not preserve fairness between competing flows at a shared bottleneck link, especially when competing flows are non-DCTCP flows.

It is shown that the bursty traffic pattern of data centres does not only come from a partition/aggregate-like workflow, but also from the actual implementation of window-based congestion control protocols in production servers [17].[17] For example, servers with a 10Gbps network interface device tend to send 30-40 packets around the same time whenever the sender's window allows them.[18] It is argued that for 1Gbps links the marking threshold of 20 packets and for 10Gbps links the marking threshold of 65 packets are a conservative adjustment in the presence of the above scenarios [17]. This implies that DCTCP with a single marking threshold at all switches (e.g. $K = 20$) becomes problematic when servers use various interface devices with different line rates. For example, imagine a server support two interface devices of capacity 1Gbps and 10Gbps respectively or a subset of network hosts only support the interface device of 10Gbps instead of 1Gbps.

Another important shortfall of DCTCP is that it assumes that the data centre fabric has low statistical multiplexing for long-lived flows (e.g. four long-lived flows with a size bigger than 1MB at a shared bottleneck link presents the degree of statistical multiplexing at the 99$^{\text{th}}$ percentile [17, 18]); if this is violated then DCTCP cannot provide a headroom for accommodating a localised burst of traffic gracefully.[19] In such scenarios, short flows with deadlines may miss these due to the presence of several

[17]TCP, MPTCP and DCTCP are equipped with window-based congestion control.

[18]The Large Send Offload (LSO) and interrupt moderation optimisations may also induce bursty traffic at a data centre's servers [17].

[19]The dynamic nature of data centre fabrics is changing rapidly so that such assumptions may be simply violated.

long-lived flows, at a shared bottleneck link, racing for share capacity and reacting to congestion signals incrementally.

In Chapter 3, we discuss our novel transport protocol (MMPTCP) that can accommodate any burst of traffic gracefully by utilising path diversity in the data centre fabrics. Additionally, MMPTCP can react to congestion by moving traffic away from congested path; i.e. it can achieve a high overall throughput (e.g. almost doubled that of single-path TCP in FatTree [5]) regardless of any statistical multiplexing present in a data centre fabric. Furthermore, in contrast to DCTCP that is agnostic to deadline flows, MMPTCP can simply handle deadline flows without any knowledge from applications, if data centres support switches with priority ingress and egress queue.

2.6.2 D^3

The core idea behind D^3 [18] is that end-hosts need to be aware of their flow sizes and deadlines so that they can request the desired rates for their flows from the network. In other words, D^3 attempts to apportion the network bandwidth by allocating deadline flows more bandwidth than their fair-share at the bottleneck links. To achieve this, the bandwidth allocation scheme requires tight collaboration and coordination between the network switches and end-hosts. The idea of the explicit rate control to apportion network bandwidth has been explored a decade ago in XCP [68].[20]

A D^3 sender exposes its desired Rate Request (RRQ) to all switches along the path. At flow initiation time, RRQ is carried in the SYN packet. In subsequent requests, RRQ is piggybacked on a data packet once per RTT. Every switch along the forward path records its suggested allocation rate in that packet, which, in turn, returns back to the sender via an ACK packet. This implies that the sender receives a vector of responses from all switches along the forward path. The sender then adjusts its sending rate based on the lowest offered rate in the response vector.

Every RTT, the desired rate request for a flow is changed as the remaining flow size is changed. Additionally, the offer rates by the network may change as network conditions change. For a deadline flow, the initial desired rate (r) is calculated by $r = \frac{s}{d}$, where s is the flow size and d is the deadline. The size of parameter s is decreased as

[20]XCP attempts to decouple network fairness from network utilisation in which any differential bandwidth allocation schemes can be defined without hurting overall network utilisation. D^3 is an example of a new differential bandwidth allocation scheme which essentially cares about flows' deadlines.

data delivery progresses. This implies that a deadline flow requires less capacity as data transmission progresses. At interval (t), the desired request rate (r) for the next RTT is calculated by $r_{t+1} = \frac{remaining\_flow\_sizes - s_t * rtt}{deadline - 2 * rtt}$, where s_t is the current sending rate. The numerator indicates the remaining flow size and the denominator indicates the available time before flow's deadline is reached in the next RTT.

The switches calculate an offer rate (a) for a deadline flow by $a = (r + f_s)$ and a non-deadline flow by $a = f_s$, where f_s is the fair share of spare capacity after allocating all deadline requests. This implies that rate allocation is first performed for all deadline flows. The remaining capacity would then be shared fairly between all existing flows (i.e. deadline and non-deadline flows). This way, a deadline flow may get more than their desired rate in the first few RTTs. The implication here is that future desired request rates would be lower so that the switches can have more spare capacity to share with other deadline flows, and hence accommodate more deadline flows. If switches do not have sufficient capacity to accommodate all deadline flows, they greedily grant rates to deadline flows on a first-come-first-served basis and halt rest of the flows for next RTT, regardless of their flow types.

To achieve such an advanced bandwidth allocation scheme, switches keep three states per each output interface: (1) the total number of flow passing through a interface, (2) the sum of the desired rates of deadline flows, and (3) the sum of the allocated rates.

D^3 increases the overhead in bandwidth slightly as it needs to carry extra information in a data packet and corresponding ACKs. More importantly, it increases the latency of some flows as it pauses their data transmission to begin even when a single switch along the path cannot offer a rate. This might occur due to a false positive estimation of aggregated bandwidth by a switch en route, i.e. a switch falsely assumes its capacity is fully saturated due to traffic bursts for instance. Although this condition may persist quite briefly, some deadline flows may miss their deadlines and valuable network resources are wasted.

The main drawbacks of D^3 are listed below:

- It requires critical modifications to the main networking elements of data centres, such as applications, switches and end-hosts. These requirements essentially make this solution non-viable for already deployed data centres.

- It requires very high-end switches with extra chips throughout the network in order to be able to keep state per each output port and perform bandwidth allocation at line rates.

- It cannot coexist with regular TCP or UDP flows since bandwidth allocation by switches is meaningless to TCP/UDP flows. Thus, as authors clearly admit, incremental deployment cannot be achieved.

2.7 Summary

In this chapter, we explained two related topics to our research: (1) transport protocols; and (2) data centres.

First, we explored TCP variants, focusing on Tahoe, Reno, NewReno, SACK and DSACK. Our aim here was to understand the design principles of each underlying congestion control algorithm used in each TCP version. We then discussed MPTCP and well-known multipath congestion control algorithms, including 'Uncoupled-TCP', 'Fully Coupled', 'Semi-Coupled' and 'Linked Increases'.

Secondly, we reviewed conventional data centre network architectures, focusing on their limitations (e.g. scalability, agility and full bisection bandwidth) and existing solutions to these limitations. We then presented an overview of modern data centre topologies (e.g. FatTree and VL2) and their current issues, and existing solutions.

Chapter 3

Design of the MMPTCP Protocol

3.1 Introduction

This chapter presents the goals, design, challenges, and potential benefits of the MMPTCP protocol in modern data centres. It is organised as follows. Section 3.2 describes the goals of MMPTCP. Section 3.3 reviews our solution (Host-PS) for improving the Packet Scatter (PS) protocol. Host-PS allowed us to think beyond existing solutions and helped us in designing MMPTCP. Section 3.4 analyses MPTCP in the data centre context, focusing on the influence of the number of subflows on short and long flows. The intuitions presented in this section shaped the main design principles of MMPTCP. Section 3.5 presents the design of the MMPTCP protocol, including its benefits within modern data centres. Section 3.6 discusses solutions to packet reordering, which is a key challenge for MMPTCP. We also discuss our ideas for preventing spurious retransmissions. Section 3.7 reviews the potential benefits of MMPTCP for latency-sensitive short flows that contain deadline in their flow completion times. We discuss our ideas about how MMPTCP can potentially overcome several challenges in this context.

3.2 Goals

In this section we set out our main goals in designing the MMPTCP protocol.

1. *High burst tolerance.*

   The network should be able to accommodate sudden bursts of traffic. The main reason of achieving high burst tolerance is to prevent transient congestion in the network. Transient congestion typically occurs due to the bursty traffic pattern of

short flows, which comprise 99% of total flows within data centres [17].

2. *Low Latency for short flows.*

   The network should be able to handle short flows with minimal latency. The reason for aiming at low latency for short flows is to enable specific applications to achieve their results within expected quality and time budget.

3. *High throughput for long flows.*

   Long flows should be able to use network resources efficiently. The reason for aiming at high throughput for long-lived flows, which are typically non-latency-sensitive, is to use network resources efficiently and realise high overall network utilisation by reacting to possible persistent congestion gracefully. Persistent congestion typically originates from collisions between long-lived flows at the bottleneck links. Long-lived flows contribute to 90% of the total bytes in data centres [4]. Additionally, long-lived flows are also prominent in applications with partition/aggregate workflow since they are typically responsible for updating workers with fresh data. Lack of fresh data can impact the quality of responses from workers.

3.3 Packet Scatter

Packet Scatter (PS) is an alternative solution to MPTCP and its applicability to data centres has been explored very briefly [12, 15]. The key idea behind PS is that network *switches* diffuse traffic on a per-packet basis instead of on per-flow, as in Valiant Load Balancing [10]. Traffic can thus be distributed as evenly as possible among all paths between two endpoints. TCP senders must run more robust Fast Retransmit algorithms to deal with out-of-order packets.[1]

It has been argued that if traffic load is equal among servers and a data centre has a uniform network topology, such as FatTree [5] or VL2 [4], then PS achieves perfect load balancing and eliminates congestion from the network core [12].

However, although per-packet traffic diffusion by switches may not create hotspots at the core of the network, per-flow may indeed do so due to potential collisions when flows are distributed through ECMP per-flow routing.[2] In other words, PS naturally

[1] The details discussion of packet reordering is in Section 3.6.
[2] ECMP limitation has been discussed in Section 2.5.

prevents congestion in any region of the network topology with dense path diversity, but congestion may still occur at the network core. Network hardware failures or traffic flowing from the Internet to the data centres, which typically consist of single-path TCP flows, may cause such congestion [12]. Additionally, PS cannot prevent congestion at the access layer of single-homed network topologies, such as FatTree[3], because there is no path diversity at that layer. For example, the high fan-in/fan-out problem, which is common in distributed file storage or web searching applications, typically occurs at the access layer [12].

The intuition here is that PS cannot deal with congestion gracefully, so that even a single packet lost may lead to needless multiplicative reductions of the congestion window (i.e. it exerts a negative effect on the overall network utilisation and connection throughput).

Our initial question was: could we extend PS protocol so that it can detect and better react to network congestion?

Host-PS. Our initial thought was to run PS via the end-hosts instead of switches, as end-hosts can carry out complex operations. In order to randomise traffic in this manner, per-flow ECMP needs to be activated on switches and the end-hosts need randomise source ports of their packets.

By devising random source port per-packet, the hash of standard 5-tuple is also changed per-packet. As a result ECMP routes each packet via a potentially different path. Additionally, end-hosts need to add a flow identifier (Flow Id) per-packet as a connection identifier since a connection can no longer be identified by the standard 5-tuple due to source port randomisation.

In Host-PS, end-hosts randomise their traffic on a per-packet basis. As a result, they can be aware of the congestion state of paths their packets traverse. They achieve this by maintaining a mapping between chosen source port numbers and the corresponding packets (sequence numbers) for each flow. If any packet gets lost, an end-host can recognise the congested path. In this way, end-hosts can actively react to congestion by not assigning those source ports to any future packets.

[3]We refer to FatTree as a single-homed network topology since their servers have only one network interface device.

The main drawbacks of Host-PS are:

- The hash of different port numbers may collide with the same link/path. This means that an end-host may encounter the same congested links several times for different port numbers.

- If the flow ID changes along the path then the connection loses its identity. Unlike public networks such as the Internet, this is not a problem in data centres since there are no middleboxes to re-write the content of a TCP packet header.

- Finding a set of paths that has a good data delivery rate is a matter of trial-and-error; furthermore, as data centre environments are extremely dynamic, the quality of a path may change very quickly.

The above ideas led us to the important realisation that it is possible to randomise traffic at the granularity of packets by using end-hosts even though switches are operating based on per-flow ECMP.

3.4 MultiPath TCP

In Section 2.2.2, we have reviewed the MPTCP protocol and in Section 2.5.2 discussed its advantages in data centres. In short, MPTCP is an extension of TCP, which transfers data through multiple paths simultaneously. MPTCP is capable of actively sensing network congestion in its subflows and shifting traffic from more congested paths to less congested ones. Unlike TCP, MPTCP can deal with the network congestion gracefully by putting fewer packets on the congested subflows. The main requirement to achieve such behaviour is to retain a congestion window for each subflow and link each of them together.

The vital questions here are: how many subflows should MPTCP use in data centres? And which factors influence the number of MPTCP subflows?

The following factors are particularly significant: (1) available capacity on the network or the level of network load, (2) flow size, and (3) number of paths between a pair of hosts. To discuss these factors in modern data centres in more depth and find an answer to the above questions, we review the following scenarios:

- *1st Scenario. Long flows and lightly loaded network.*
  In this case, it makes sense to have a large number of subflows that potentially lead to better network utilisation. However, if there is no path diversity between a pair of communicating hosts then opening several subflows makes no difference.

- *2nd Scenario. Long flows and heavily loaded network.*
  The core idea behind opening several subflows in the data centre context is to improve overall network throughput by actively detecting and utilising unused capacity in the network. If a network is highly loaded and has very low spare capacity to be detected and utilised, then opening several subflows is not effective.

- *3rd Scenario. Short flows and lightly loaded network.*
  The window size of each subflow remains very small since the total number of flow packets is small. In such scenarios, a subflow may lose its ACK clock even after a single packet is lost, especially in the last window of data. The main reason is that there are not enough packets in transit to generate enough duplicate ACKs to trigger the Fast Retransmit mechanism. As a result, the sender should wait until a Retransmission Timeout is triggered as a final resort for loss recovery.

One possible solution to the 3th scenario is to integrate a Forward Error Correction (FEC) mechanism into MPTCP, similar to the one presented in TCP-IR [69]. The core idea of TCP-IR is to integrate a out of band XOR segments into the data stream of a small TCP flows such that the TCP receivers can recover one or two consecutive losses without the need for retransmissions. The XOR segment is formed by applying a XOR-based encoding scheme on a number of segments (e.g. 8 or 16). The number of segments required to form a XOR segment is dependent on a XOR-based encoding scheme. In other words, the sender needs to send a XOR segment for every small number of continuous data segments. The XOR segments are also delivered without any reliability.

There are a few issues with this approach: (1). It can effectively be used only on a very short single-path TCP flows, perhaps, containing only a few packets (e.g. web traffic); it has been shown that this approach is not improving the TCP loss recovery in the long single-path TCP flows [70]; it only adds overhead to the network with such flows. (2). It can only recover up to two consecutive segments; it is ineffective in

scenarios with multiple packet losses.

We therefore believe integrating a FEC mechanism to MPTCP can improve the MPTCP performance to some extent, especially when it is only used in the subflow's last window of data. However, running MPTCP with several subflows for short/query flows is not a right approach even with a FEC mechanism. In the 1th and 2th scenarios, we discussed the purpose of using several MPTCP subflows in the context of data centres, but that purpose is not served when MPTCP is used for short/query flows.

Another approach to the 3th scenario is to use the connection-level sequence space for the purpose of loss detection at the sender. Unlike single-path TCP, MPTCP uses two sequence spaces: one at subflow-level and the other one at connection-level; MPTCP also maintains a separate sequence space per subflow. The main purpose of the subflow-level sequence space is to allow a subflow to perform loss detection and, in turn, congestion control on its path; a receiver generates TCP duplicate ACKs at subflow-level when a segment is lost or misplaced. Each subflow is then performed loss detection and congestion control independently of the other subflows. The main purpose of the connection-level sequence space is to allow the receiver to correctly reassemble data stream, which is striped over multiple subflows and delivered via separate subflow's sequence spaces, before deliver it to the application. The MPTCP receiver also sends the data acknowledgement (data ACK) packet from this sequence space for the purpose of flow control. The data ACK packet can also show a hole in the connection-level buffer.

In this way, the sender may be able to detect a lost packet from a subflow, which lost its ACK clock due to multiple packet losses in a window of data or has a small number of packets in flight, earlier than a retransmission timeout is triggered on that subflow. A key requirement to begin with this idea is that the MPTCP receiver sends data ACK per segment received. The sender can thus detect a hole in the receiver's buffer very fast because all subflow's packets can trigger a duplicate data ACK at the receiver. There are two issues with this approach:

1. It is not clear how many duplicate data ACKs is required to detect a lost packet. Even if a duplicate data ACK threshold is selected, the sender cannot distinguish between a lost and delayed packet. For example, the receiver may send several duplicate data ACKs, during an RTT, for a delayed segment.

2. If multiple packets get drop from multiple subflows simultaneously, then duplicate data ACKs during an RTT can not provide enough information to the sender to detect all those packet losses at the same time and, in turn, prevent timeouts in all of those subflows.

Despite all the above issues, the sender can detect a stalled subflow by observing data ACKs over a few RTTs. In such a condition, the sender can be able to perform congestion control and loss recovery for the subflow in question or resend the unacknowledged segments of the subflow in question via another subflow. However, the detection of a stalled subflow is not beneficial to a flow with only a small number of packets. We therefore believe that enabling data ACKs to detect lost packets of multiple subflows is not effective or it may provide small improvement, especially for short flows. Our approach here is to use a single sequence space operating one a single congestion window, similar to the way TCP operates. In this way, all packets of a short flows can contribute to loss detection and recovery.

The intuition here is that increasing the number of subflows is useful to connection throughput of long flows, in a wide variety of network conditions, but it might be harmful to flow completion of short flows.

Is it possible to adjust the number of MPTCP subflows based on flow sizes?
It has been argued that some applications can provide high-level information, such as flow size [18]. Thus, if each end-host knew the size of its flows, it could decide how many subflows it might be effective to deploy. For example, in the case of short flows, it is better to have a single subflow. Unfortunately, the majority of existing applications do not expose their flow sizes to the end-hosts, (i.e. the network stack is unaware of this high-level application knowledge). *Thus, MPTCP can not have any idea how many subflows to open for a flow; if a predefined number of subflows is used for all types of network flows then MPTCP is likely to significantly damage the flow completion time of short flows.*

3.5 MMPTCP: Combining PS with MPTCP

This section describes the design principles and operation of MMPTCP. The core idea behind MMPTCP is that at the beginning of an MPTCP connection, the data is delivered by PS first to cover short flows, until the total data transmitted reaches a certain threshold (e.g. 1MB since the majority of flows in data centres have a flow size less than 1MB [4]). PS is then switched to MPTCP with multiple subflows to cover long flows. This implies that when a switching threshold is reached, MMPTCP opens new subflows and governs the data transmission via MPTCP congestion control. The initial subflow is only allowed to do per-packet randomisation at the beginning of connection, i.e. until a predefined threshold is reached. Then, it becomes inactive and cannot send any more new segments.

The design principles of MMPTCP are as follows:

1. Prevent MPTCP with several subflows from handling short/mice flows in order to avoid having multiple subflows with a small window size. The small window size of subflows is problematic because MPTCP uses a separate sequence space per subflow. All flight packets of a subflow can only be used in loss detection on that subflow. As a result, if a subflow is responsible to handle a small number of packets in its lifetime then it is very likely that it needs to recover its lost packets via timeouts rather than fast transmissions. With MMPTCP, short flows are handled by PS with a single sequence space (i.e. TCP sequence number) operating on a single congestion window, so that the chance of experiencing timeouts is significantly decreased because all packets of short flows are engaged in loss detection and recovery.

2. Decrease the burstiness of data centre networks, which mainly originates from short flows, by diffusing packets throughout the network. This implies that MMPTCP can significantly prevent transient congestion in the network core of modern data centres.

Switching between PS and MPTCP allows end-hosts to achieve following benefits:

- Handling short flows via single sequence space in order to increase the chance of packets loss recovery via fast retransmissions rather than timeouts when the congestion window is small.

- Handling long flows via multiple congestion windows in order to achieve high connection throughput for long flows and hence high overall network utilisation.

- Diffusing bursty traffic throughout the network.

By randomising the source port in each packet, we effectively combine Host-PS with MPTCP. Network switches only perform per-flow ECMP and end-hosts perform per-packet randomisation only on the initial subflow of a MMPTCP connection. A token is added to each packet of the initial subflow as a connection identifier, so that a randomised packet can be forwarded to a corresponding MPTCP connection correctly (note that the standard 5-tuple is no longer valid during the initial phase of data transmission (PS phase)).[4]

After the sender reaches a switching threshold, it opens further subflows to carry on the data transmission via MPTCP and deactivates the initial subflow but does not close it.[5] The initial subflow is the only subflow presented to the applications and if it was closed, the connection would lose its identity. In other words, after switching to MPTCP no more data is placed on the initial subflow, which is ignored by the MPTCP congestion controller. Furthermore, in the initial handshake of MPTCP, SACK may also be activated if DSACK is used as a part of the packet reordering strategy. MPTCP works with SACK, so there is no difficulty in having SACK activated over the lifetime of a MMPTCP connection. However, using DSACK is only essential in the PS phase.

3.6 MMPTCP and Packet Reordering

A TCP sender receives a duplicate acknowledgement (duplicate ACK) when a packet gets dropped, delayed or reordered. It enters the Fast Retransmit phase upon the arrival of the third duplicate ACK for a missing packet (when the duplicate ACK threshold parameter is set to three). The TCP sender retransmits the perceived lost packet and halves its congestion window as a reaction to the congestion signal.

The problem here is obvious: senders cannot distinguish between a duplicate ACK generated because of a reordered packet, and that generated for a lost one. That is, the sender behaves identically for both events since the signal is the same.

[4] A token is a locally unique identifier assigned to an MPTCP connection upon establishment.

[5] To prevent sudden hold of data transmission, after a switching threshold is reached, the initial subflow becomes deactivated only when at least one new MPTCP subflows are established.

To deal with this problem, a TCP sender waits to receive three duplicate ACKs before enabling the Fast Retransmit mechanism [32]. However, the Fast Retransmit mechanism may be still falsely triggered when a reordered packet reaches the receiver after it has sent a third duplicate ACK. This condition may lead to spurious retransmissions of reordered packets even if no loss has occurred. In other words, the sender interprets the reordered packet as lost and also interprets the respective signal as an indication of congestion. As a results, the sender falsely triggers the Fast Retransmit mechanism and halves its congestion window.

It is clear that the duplicate ACK threshold (*dupthresh*) of three makes sense in the Internet when a TCP connection operates with minimal reordering in its data delivery. In data centres, however, even the *dupthresh* of one can be used since a TCP connection typically operates on a single path with no reordering. With the packet-scattering approach, however, TCP needs to be more robust to packet reordering, since RTTs on different network paths may vary over time due to queuing delays. In such cases, packet reordering becomes the norm instead of the exception.

Setting the right *dupthresh* value is not trivial; if *dupthresh* is too low, spurious retransmissions become the norm. If it is too high, the sender may react to congestion through a retransmission timeout instead of the Fast Retransmit mechanism. Our experimental evaluation in Section 5.3 confirms these observations.

There are three key aspects in making TCP more robust to packet reordering: preventing, detecting and mitigating spurious retransmissions.

One well-known solution for 'detecting'and 'mitigating'spurious retransmissions is DSACK [25], which is an extension of SACK TCP [24].[6] SACK TCP can deal with multiple packet drops much faster than other versions of TCP (e.g. Tahoe, Reno and NewReno [31]). This is a particularly beneficial for latency-sensitive flows. When a spurious retransmission is detected by DSACK, the state of the congestion window can be simply reversed to the state when a loss is detected.[7]

One possible approach for 'preventing' spurious retransmissions is to dynamically adjust the *dupthresh* parameter based on information that can be retrieved from

[6] If PS does not use SACK TCP (e.g. it uses NewReno TCP [23]), then DSACK can not be applicable. In this situation Eifel algorithm [71] might be an alternative solution; it uses timestamp or two bits from the TCP reserved field to disambiguate an original transmission from a retransmission.

[7] Detailed discussions for SACK and DSACK can be found in Section 2.2.1.4 and 2.2.1.5 respectively.

DSACK, SACKs, ACKs, RTOs and Fast Retransmits. RR-TCP [72] follows a similar approach. RR-TCP attempts to adjust the *dupthresh* value dynamically by understanding the maximum distance in packets by which a segments is displaced, based on feedbacks from the networks.

In modern data centre topologies, such as FatTree and VL2, it is possible to adjust the *dupthresh* value according to the maximum number of possible paths to a destination. For example, if there are 10 possible paths for a flow to randomly send their packets then a *dupthresh* of 13 is a reasonable value (our experimental evaluation in Section 5.3 confirm this). Spurious retransmissions could thus be prevented to a significant degree.

We assumed that the routing protocol is able to provide information about maximum possible paths to a destination. This assumption is inspired by VL2 routing design as it uses a central node for finding the ToR switch of the destination node in the network, and FatTree addressing scheme in which a location of a host can be identified from its IP address. The sender can thus choose an appropriate value for the *dupthresh* based on this information. For example, if a source sends its traffic via the core layer then the *dupthresh* value should be much higher compared to when it sends its traffic via the access layer (ToRs).

In this thesis we use the FatTree addressing scheme, as discussed in Section 2.4.2.1, as a basis for adjusting *dupthresh*. That is, each source node can simply infer the layer of network topology that its traffic passes by just comparing its own IP address to its destination IP address. For example when node one with the IP address "10.0.1.1", wants to send data to node two with the IP address 10.0.1.2, it can simply work out that the destination host is located in the same ToR switch as itself and the *dupthresh* should not be changed from the default value of three.

The knowledge of the end-host's location is essential but not sufficient to assign an appropriate value for the *dupthresh*; each end-host also needs to know the size of the network topology. For example, a network topology with 4 cores requires a different value of *dupthresh* than a network topology with 8 cores. Additionally, network switches may also support ECMP with a limited number of paths in each IP subnet (e.g. up to 16 equal-cost paths), therefore knowing these information is also important for deciding a precise value for duplicate ACK threshold.

In [73], several other approaches have been explored, some of which might be suitable for the dynamic environment of data centres; e.g., a simple alternative solution to RR-TCP [72] is to use a short timer (e.g. as small as one RTT) after receiving three duplicate ACKs. This step prevents the risk of increasing the *dupthresh* value that may lead to timeouts, especially when the TCP flow and its congestion window are small.

3.7 MMPTCP and Latency-Sensitive Flows

In this section, we discuss the potential benefits of MMPTCP in other data centre contexts, e.g. traffic management inside data centres (see Section 2.6).

Assumptions. Let us first assume that MMPTCP is capable of dealing with possible network congestion gracefully. That is, the packets of a flow can be forwarded via all possible paths to their destinations without any concern about packet reordering. In this case, we can say perfect load balancing is achieved (i.e. transient congestion can be prevented and persistent congestion can be dealt with gracefully).

Question. Does our hypothetical transport protocol that provides perfect load balancing also decrease the maximum flow completion time of short flows compared to TCP in modern data centres such as FatTree and VL2?

Answer. Preventing network congestion is vital for meeting the deadline of the majority of flows since flows normally miss their deadlines because they encounter network congestion. In other words, MMPTCP certainly helps to decrease the average flow completion time, but it may not be effective in decreasing the maximum flow completion time in a single-homed topology such as FatTree. The main reason is that a large number of short/query flows may not meet their deadlines due to transient congestion at the access layer of the network, in which our hypothetical transport protocol becomes ineffective due to lack of path diversity.

The following two scenarios highlight situations in which MMPTCP cannot achieve perfect load balancing in a single-homed network topology:

- **Scenario 1: The resource-allocation problem:** when deadline flows (typically short flows) and non-deadline flows (typically long flows) are competing for network bandwidth at access links (i.e. there are bottlenecks on the access links of the senders and/or the receivers). This is a common situation in data centres since each server supports different applications, and hence different traffic matrices.

It is likely that deadline flows cannot meet their deadlines when several diverse flows are competing for fair network capacity. In this condition, even queueing delay may cause a flow to lose its deadline [17, 18, 13].

- **Scenario 2: The incast problem:** when a link needs to support a large number of synchronised flows to/from a node. As a result, a heavy transient congestion becomes apparent that leads to timeouts in several of those flows. Incast can commonly occur with applications with partition/aggregate workflow that cause transient congestion on aggregators (typically front-end servers) [17, 18, 13]. In this situation several flows might miss their deadline and the network throughput may significantly drop (i.e. there is a negative impact on the quality of application results, network utilisation and hence revenue). Furthermore, those aggregators might also have some long flows which aggravates the problem.[8]

  Figure 3.1 shows the incast problem due to synchronised responses from the workers to an aggregator.

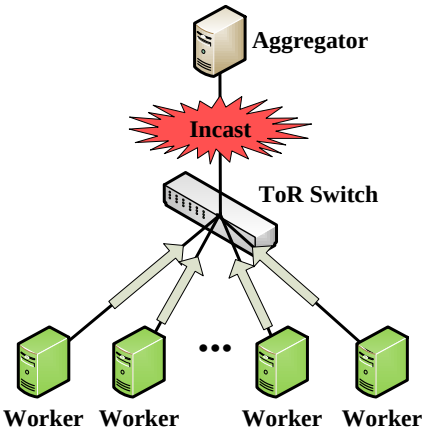


Figure 3.1: The incast problem due to the partition/aggregate workflow.

The intuition here is that MMPTCP cannot solve the possible congestion on the access link of receivers/senders in the FatTree-like network topology since it has no path diversity. In other words, MMPTCP can achieve perfect load balancing in the layers of the the network topologies that provide path diversity. This intuition is also applicable to MPTCP and PS; we compare and analyse the performance of MMPTCP, PS, TCP and MPTCP under various incast scenarios both with short flows and with long flows in Section 5.12.

[8] A similar problem as Scenario-1 might occurs.

The important questions here are: should we modify the networking protocols, such as TCP congestion control, to mitigate these problems [17, 18]? Or could it be easier and more flexible to solve them by providing more path diversity in the access layer? And using multipath transport protocols, such as MMPTCP, which are capable of utilising multipaths anywhere in the network?[9]

Solutions to Scenario-1

First approach. Apportioning the network bandwidth based on an explicit rate control has been proposed [18]. This is achieved by assuming that high-level application knowledge, such as the flow size and its deadline, is provided to the end-hosts. That is, both switches and end-hosts collaborate with one another to divide network bandwidth among competing flows on the bottleneck links so that deadline flows can receive more bandwidth than non-deadline flows.[10] Taking feedback from switches in order to apportion the network bandwidth by end-hosts makes sense when each flow takes a single path. However, it does not seem reasonable when the flows deliver their data via multiple paths.

Second approach. Allocating priority levels among competing flows based on their deadline is a reasonable approach in the today's data centre environments [13]. Thus, it might be possible to solve the problem outlined in Scenario-1 by giving high priority to deadline flows and low priority to non-deadline flows by end-hosts; that is, if switches can have priority ingress and egress queue.

Our approach. High priority is to be given to randomised packets of *MMPTCP* protocol (i.e. the packets of initial subflow). In this way, the packet drops in this phase can be significantly prevented. This implies that the short flows, which are typically latency-sensitive and have a deadline in their flow completion times, can complete their data delivery by experiencing fewer congestion events.

Solutions to Scenario-2

The literature explored two broad approaches to address the incast problem arising as a result of transient congestion originated from query-response flows. The first is to keep the buffer occupancy low on switches, by employing the new congestion control algo-

[9] An example of such multi-homed network topology is the Dual-Homed FatTree (DHFT) [12].

[10] A similar approach has also been explored a decade ago in XCP [68]. The core idea in XCP is to decouple network fairness from network utilisation. That is any differential bandwidth allocation schemes can be simply defined without hurting overall network utilisation.

rithms for TCP that operate based on ECN feedback by switches (DCTCP [17]). The second is to quench some flows by switches to allow other flows to meet their deadline (D^3 [18]). The latter solution damages the quality of final results return back to the applications; the lower number of computations from workers to an aggregator, the lower quality of final results become. The former solution falls short when the statistical multiplexing in an aggregator is high or replies from the workers to an aggregator are quite large in size, i.e. more than few packets. Maintaining low buffer occupancy in such cases becomes infeasible even in modern switches which provide dynamic memory allocation to accommodate traffic surges. Both solutions make sense when a flow takes a single path.

There are other approaches for *mitigating* the TCP incast problem, especially in the context of data centre storage, to a great extent by reducing RTO_{min} [34, 74]. However, these approaches cannot *prevent* transient congestion. The application designer typically introduces the jitters as a prevention mechanism to the TCP incast. This approach may slightly prevent transient congestion, but the problem is not completely eliminated.

Our approach. The incast problem can be mitigated to a great extent via a multi-homed network topology such as DHFT [12], since a burst of traffic can be distributed over multiple interface devices of the servers. The key advantage of the MMPTCP protocol in such topologies is that it can effectively use all available interface devices simultaneously. If a flow is short, then the PS mechanism randomises the traffic evenly in all interfaces and if a flow is long, then the MPTCP mechanism actively balances the traffic between all interfaces. We believe that MMPTCP operating on multi-homed network topologies can effectively solve the incast problem to a great extent.

3.8 Summary

In this chapter, we presented the MMPTCP protocol, focusing on its goals, operations and challenges. We discussed how packet reordering could be addressed in MMPTCP, including our solution for adjusting *dupthresh*. Finally, we overviewed how MMPTCP can be an interesting solution for latency-sensitive short flows.

Chapter 4

MPTCP and MMPTCP Implementation in ns-3

4.1 Introduction

This chapter presents a high-level overview of our implementations in using network simulator-3 (ns-3), a discrete-event driven simulator commonly used in academia [75].

This chapter is organised as follows. Section 4.2 describes the current TCP model in ns-3. Section 4.3 presents our implementation of MPTCP in ns-3. Section 4.4 studies some functionality details of MPTCP main classes and their interactions. Section 4.5 provides an example of how a packet of MPTCP flow walks through the ns-3 networking stack, with the aim of highlighting our contributions at different layers of the ns-3 networking stack (e.g. networking and application layers). Section 4.6 briefly reports on MPTCP signalling and our implementations. Section 4.7 describes our implementation of MMPTCP and Packet Scatter (PS) in ns-3. The last section showcases our MPTCP, ECMP and PS implementations through some simple simulations. This section essentially attempts to examine different key algorithms of MPTCP, such as loss recovery, timeout and congestion control.

4.2 TCP Architecture

Several TCP variants are currently implemented in ns-3. The key ns-3 modules concerning variants of TCP implementations are listed below; their interactions are presented in Figure 4.1.

- **Socket**: This abstract class provides a low-level socket API based on the BSD

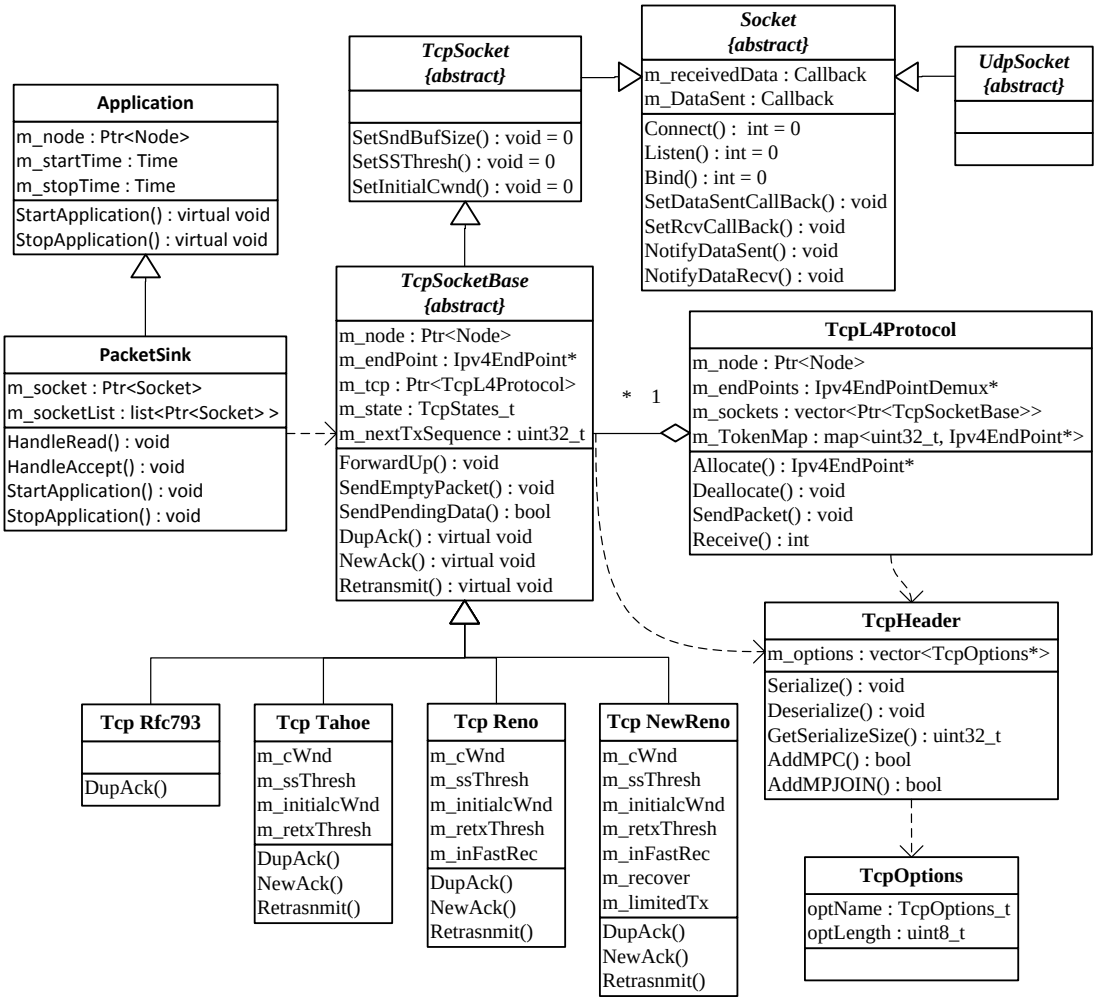


Figure 4.1: TCP class diagram

socket API across all transport sockets, e.g, TCP or UDP socket. Sending and receiving operations to/from the applications uses callbacks provided in this class. The *Socket* class inherits from the *Object* class.

- **TcpSocket**: This pure abstract class inherits from the *Socket* class and includes all essential attributes for a TCP socket, e.g. setting TCP segment size, initial window, and sending buffer size.

- **TcpSocketBase**: This class provides key functionality of TCP and a socket interface for the applications. For example, it performs connection management, packet reordering and congestion control for a TCP connection. It is also responsible for sending and receiving packets to and from the application layer. All variants of TCP inherits from the *TcpSocketBase* class, as shown in Figure 4.1.

- **TcpHeader**: This class provides all functionality needed to form the standard TCP header for a segment [32], e.g. appending a TCP option to the header, serialising and deserializing the packets.

- **TcpL4Protocol**: This class provides an interface between the network layer and the TCP socket. It is responsible for sending and receiving packets to and from the network layer and the TCP socket (*TcpSocketBase*). This class can be viewed as a shim layer between the network layer and transport layer.

The ns-3 simulator heavily uses callback functions for interaction between most of its classes. The goal of callbacks is to permit one chunk of code to call a function without any particular inter-module dependency [76]. As an example, in Figure 4.1 *PacketSink* application, which is a receiver application, can register its *HandleRead()* function in the *TcpSocketBase* class, for receiving data via the *SetRecvCallback()* function of the *Socket* class. Therefore a TCP/UDP socket can pass data to the *PacketSink* application, without any explicit knowledge of the *PacketSink* class, simply by invoking the *NotifyDataRecv()* function of the *Socket* class, which triggers a callback to the *PacketSink::HandleRead()* function.

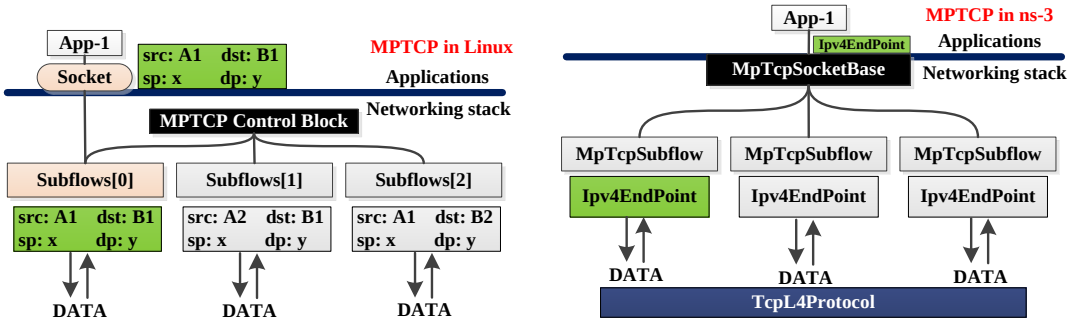
All classes related to variants of TCP inherits from the *TcpSocketBase* class and override the congestion control and loss-recovery related functions in the *TcpSocketBase* class, e.g. *DupAck()*, *NewAck()* and *Retransmit()*, as can be seen in Figure 4.1.

4.3 MPTCP Architecture

To date, a single attempt has been made to implement MPTCP in ns-3 that is in ns-3 version 3.6 [77]. However, this model was never merged with any stable version of ns-3 and also became obsolete after TCP was rewritten in ns-3 version 3.8. In this model, only a single client could connect to an MPTCP server; i.e. a server could not fork new MPTCP connections. This is a problem when dealing with realistic traffic models in data centres. Additionally, the model did not support nodes running TCP and MPTCP in parallel, a feature that is often required in data centre experimentation, when evaluating fairness among competing TCP and MPTCP flows. MPTCP tokens [14], which uniquely identify MPTCP connections in a host and are used to associate new subflows to an existing MPTCP connection, are not supported. The TCP timeout

mechanism was modified, so that if the retransmission timer was triggered on a subflow, the congestion window of that subflow was only halved instead of being reset to one segment and then being entered to the Slow-Start phase. Finally, the existing model incorporated several other simplifications (e.g. the MPTCP connection and its subflows did not follow standard TCP state transitions).

As a part of this thesis, we designed and implemented a new model of MPTCP in ns-3 that closely follows RFC 6828 [14]. Subflow management is carried out similarly to the Linux Kernel implementation and loss recovery is based on TCP NewReno. Our ns-3 model of MPTCP overcomes all limitations described above [21][1].



(a) MPTCP architecture in Linux kernel. (b) Our proposed MPTCP architecture in ns-3.

Figure 4.2: Comparing our MPTCP model with the Linux Kernel model

Figure 4.2(a) depicts the architecture of MPTCP in Linux Kernel [78] and Figure 4.2(b) shows our proposed architecture of MPTCP in ns-3. In our model, the MPTCP control block is created directly by the application and the initial subflow is created by the MPTCP control block. This implies that an application can directly access MPTCP subflows. However, in the Linux model, the MPTCP control block is created by the initial subflow, after which the initial subflow is created by the application at the socket creation. Thus, an application can only see and access the initial subflow, so MPTCP is completely hidden from the application layer. Additionally, the Linux kernel model allows senders to carry on with data transmission via the initial subflow without opening further subflows if the receiver is not MPTCP capable, i.e. the sender can simply fall back to the regular TCP and carry on the data transmission via the initial subflow after which it detects the receiver is not MPTCP capable. The sender can also eliminate the

[1]We have written more than 10K lines of code in this thesis contribution. The source code of our MPTCP implementation can be found by following the link: https://github.com/mkheirkhah.

created MPTCP control block at the connection establishment.

4.4 MPTCP Class Interaction

This section gives a high-level overview of our MPTCP implementation in ns-3, by focusing on its main classes and their interactions. The key ns-3 classes of our MPTCP model are listed below; their interactions are presented in Figure 4.3.

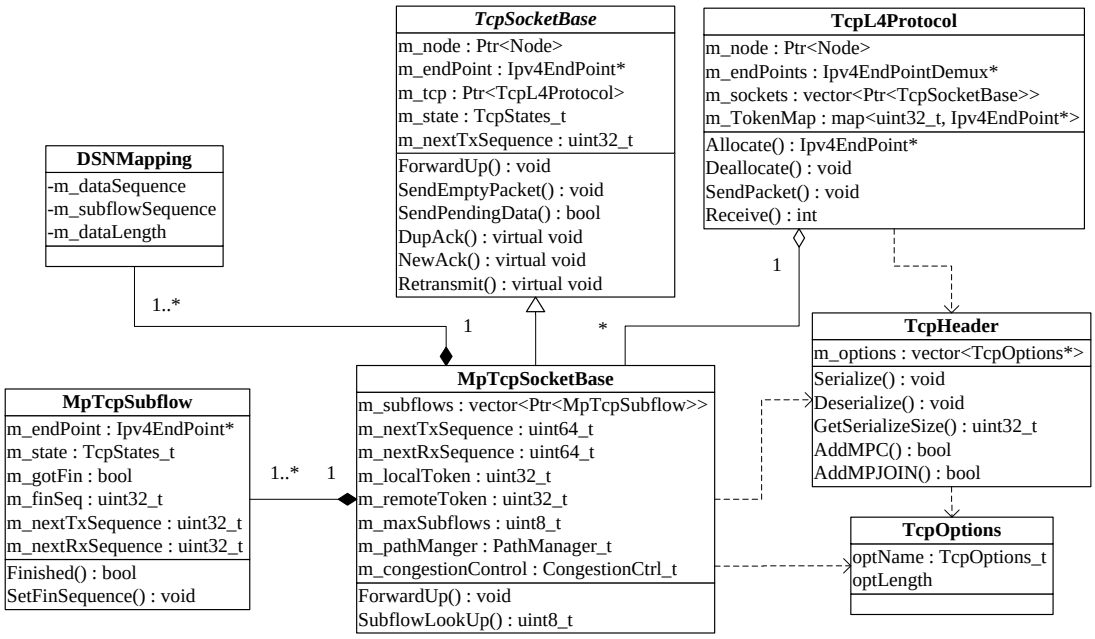


Figure 4.3: MPTCP class diagram

- **MpTcpSocketBase**: This class implements the MPTCP control block and exports the socket API to ns-3 applications. It performs connection management, data scheduling, packet reordering, path management, congestion control and loss recovery for all MPTCP subflows.

  On the server side, an MPTCP connection is represented with one listening *MpTcpSocketBase* object, which forks new *MpTcpSocketBase* objects for each accepted MPTCP connection. On the client side, an *MpTcpSocketBase* object represents an MPTCP connection with a server. *MpTcpSocketBase* is a subclass of *TcpSocketBase*.

- **MpTcpSubflow**: This class represents an MPTCP subflow and essentially holds

all TCP parameters. This class is a subclass of the *Object* class.[2] An *MpTcpSocketBase* object may have multiple *MpTcpSubflow* objects.

- **DSNMapping**: Data Sequence Number Mapping is a standalone class, presenting a mapping between a subflow-level sequence number and connection-level sequence number assigned to each sent segment. A sender *MpTcpSubflow* object holds a *DSNMapping* object for each sent segment until arrivals of an ACK packet that acknowledges the sent segments. The receiver *MpTcpSocketBase* also holds an object of this class for each received unordered segment in its out-of-order buffer in order to perform per-connection reordering. In other words, MPTCP makes use of the DSNMapping, linking segments sent on different subflows to a 64-bit connection-level sequence numbering, permitting segments sent on different subflows to be reconstructed in an orderly manner at the receiver.

- **TcpL4Protocol**: We changed this class so that MPTCP connections can be handled, without disrupting any existing ns-3 TCP functionality. As with single-path TCP models, this class is an interface between the transport and network layers and is responsible for sending and receiving packets to and from the network layer, respectively. When a packet is received, *TcpL4Protocol* looks up an endpoint (*Ipv4EndPoint*) based on the TCP header's four-tuple, as shown in Figure 4.4(d). In our MPTCP model, several *Ipv4EndPoint* objects, representing endpoints for respective MPTCP subflows, can be associated to one *MpTcpSocketBase* object, as shown in Figure 4.4 (b) and (c). To achieve this, we implemented the MPTCP token support in this class. A token is a locally unique identifier assigned to an MPTCP connection upon establishment. When a sender initiates a new subflow, the receiver looks up an *Ipv4EndPoint* based on the token passed in the MP-JOIN option and forwards the request to the respective *MpTcpSocketBase* object, as shown in Figure 4.4 (c). Without the MPTCP token lookup mechanism, the SYN packet in Figure 4.4 (c) would be incorrectly forwarded to the MPTCP listening socket. Requests for new MPTCP connections are resolved using the four-tuple and forwarded to the listening *MpTcpSocketBase* object, as

[2]Essentially this class should be extended from TcpSocketBase, the main reason that it is currently extended from *Object* class is to prevent any modification to the core implementation of TCP in ns-3.

illustrated in Figure 4.4 (a). Adding the token lookup mechanism in this layer allows a node to receive TCP and MPTCP traffic simultaneously.

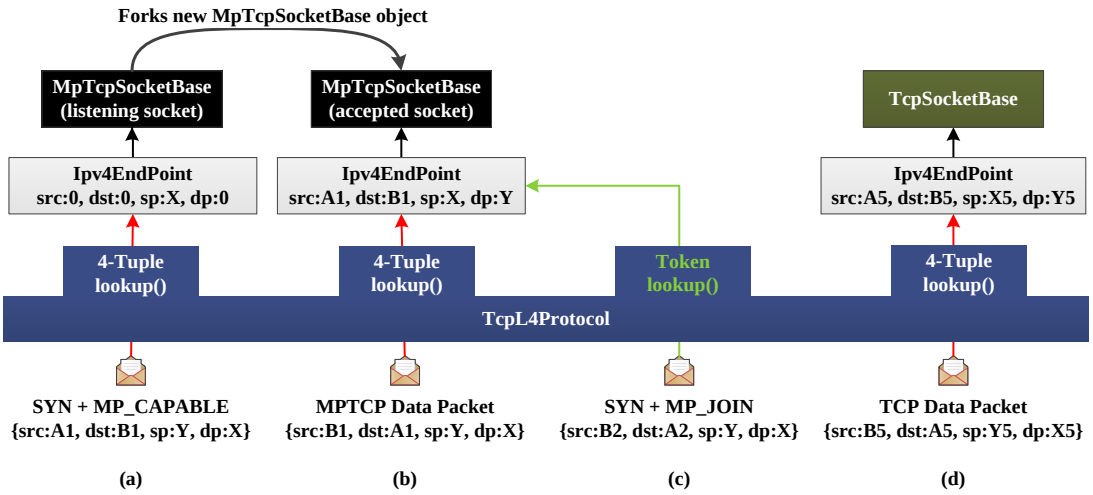


Figure 4.4: An example of the token and 4-tuple lookup mechanisms in *TcpL4Protocol* class

4.5 Networking Stack Trace

This section aims to overview the ns-3 networking stack and highlights our contributions at different layers of the stack. Figure 4.5 depicts an example of how packets flow through the ns-3 networking stack. MPTCP is used as a case study in this example. The key operations and function calls in each layer will be explored in detail.

1. Let us assume an application, in this case *MpTcpBulkSendApplication*, has already created a MPTCP socket, so it can directly call the *SendPendingData()* function of *MpTcpSocketBase* class in order to transfer its data from the application buffer to the MPTCP socket sending buffer. The TCP socket layer notifies the application layer after each sending process, so that the sending socket buffer can be refilled again by the application layer.

2. *MpTcpSocketBase* chooses the subflows by using a round-robin algorithm, so that subflows can be selected sequentially and in a circular manner in order to distribute the segments equally in subflows. If a subflow has a window size of less than one maximum segment size (MSS), the next subflow is selected.[3] After a subflow is selected, *MpTcpSocketBase* creates the TCP header with the

[3]This condition is not applied for the last segment as it may be less than one MSS.

attached Data Sequence Signal (DSS) option. The MPTCP sender signals the mapping between the subflow-level to the connection-level sequence number via the DSS signal. Thereafter, *MpTcpSocketBase* finds the output interface object associated with the subflow's source IP address, and the packet, TCP header, source IP address, destination IP address and output interface are in turn passed to the next layer via the *TcpL4Protocol::SendPacket()* function.

3. *TcpL4Protocol* is where the socket-independent protocol logic for TCP and MPTCP is implemented. The *SendPacket()* function adds the TCP header, initialises the checksum (if enabled), finds a route entry object by querying the routing protocol, and finally sends the packets to the Ipv4 layer (Internet Layer).

   The routing table in ns-3 includes several routing entry objects. Each routing entry object, i.e. *Ipv4Route* class, holds information related to the source and destination IP address, gateway and output interface device. The output interface device of the found route entry object should have an IP address identical to the source IP address of sending packets.

   We enforced the bounding between a subflow and specific interface device in order to make sure that segments sent from a subflow leave the end-host through an associated output interface device. This is because it is possible in ns-3, if this constraint is not enforced, that all segments from all subflows exit via a single output interface device since that output interface device has the shortest path to the destination. The *TcpL4Protocol* class does not send packets directly to the network layer but via a callback called *m_downtarget*. In this case, *m_downtarget* is linked to the *Ipv4L3Protocol* class.

4. *Ipv4L3Protocol* includes key network layer functionality, adds the IP header and sends the packet to *IPv4Interface* class. If address resolution (ARP) is required, the *Lookup()* function of *ArpL3Protocol* class is called directly. Thereafter, the *Send()* function in the *NetDevice* class is called.

5. The *NetDevice* class calls the *Node::ReceiveFromDevice()* function on the destination node (or the next hop) by triggering a callback called *NetDevice::m_rxCallback* on the destination *NetDevice* object.

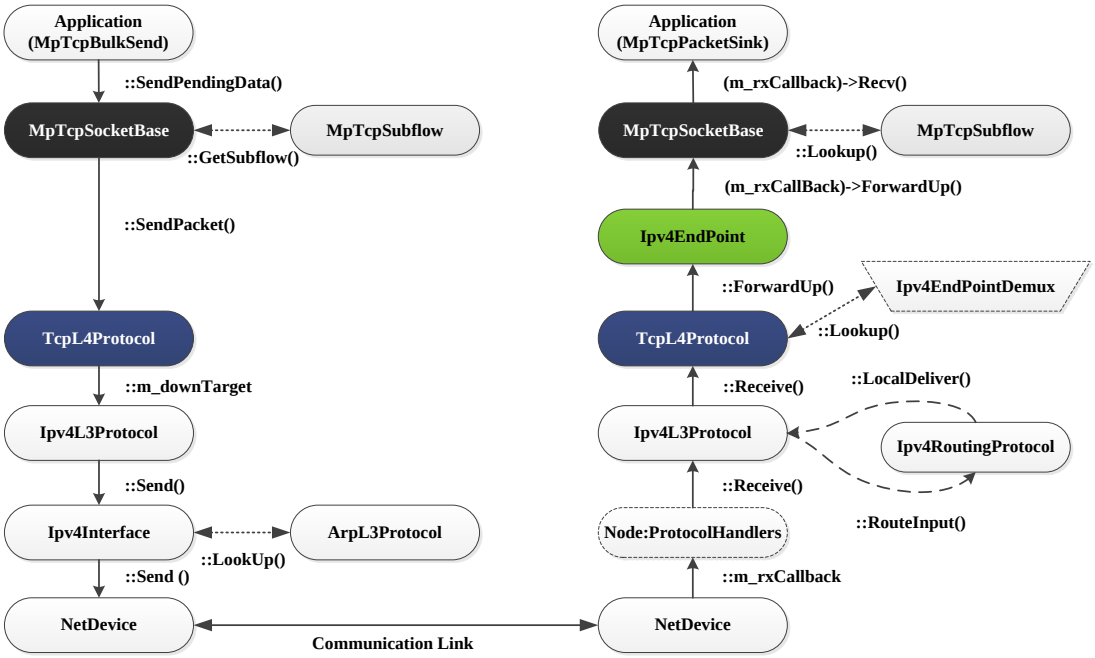


Figure 4.5: An example of the ns-3 networking stack trace of how packets flow through the ns-3 node objects

6. The destination node calls the *Node::ReceiveFromDevice()* function to lookup for a callback from a stored set of callbacks (*ProtocolHandlers*) to forward the packets up the stack. The lookup mechanism is based on the protocol number and the network device of the received packet. In this example, the result of the lookup is a callback to the *Ipv4L3Protocol::Receive()* function as the packet belongs to a MPTCP subflow that holds the TCP protocol number.

7. Ipv4L3Protocol decapsulates the TCP header from the packet, and passes the packet, IP header and device objects to the *RouteInput()* function of *Ipv4RoutingProtocol* class.

   The function *RouteInput()* is an abstract function in the *Ipv4RoutingProtocol* class, so its implementation depends on which routing protocol is employed. In our data centre model, we use the existing ns-3 global routing mechanism (*Ipv4GlobalRouting* class), so a routing table is formed based on the Open Shortest Path Routing (OSPF) algorithm, which is the common routing protocol used in today's data centres [4, 5].

   The *RouteInput()* function decides whether a packet is for a local delivery or should be forwarded to a another node. If the packet is for lo-

cal delivery, the call return back to the *Ipv4L3Protocol* class by calling the *Ipv4L3Protocol::LocalDelivery()* function. This function essentially forwards the packet to the appropriate shim layer based on the protocol number, e.g. for TCP and MPTCP the packets are forwarded to the *TcpL4Protocol* class. However, if a packet is required to be forwarded to another node, *RouteInput()* looks up the routing table and finds a route entry object to forward the packet to the next hop. After a next hop is selected, a callback is triggered and the call returns back to *Ipv4L3Protocol* by calling the *IpForward()* function directly, and then the packet is forwarded down the stack.

We implemented ECMP flow-based routing, based on hashing the standard five tuple, in the *LookupGlobal()* function that is called from the *RouteInput()* function. The *LookupGlobal()* function is responsible for finding the possible routes to a destination IP address by searching the global routing table. If ECMP is activated and *LookupGlobal()* finds more than one equal-cost path to a destination then ECMP per-flow routing will select the next hop, based on the hashing of the standard five tuple of the packet header (i.e. source address, destination address, source port, destination port, protocol number).

8. The *TcpL4Protocol::Receive()* function removes the TCP header from the packet and looks up the per-flow context state, which is one or more *IPv4EndPoint* objects stored in an *Ipv4EndPointDemux* object. The *Ipv4EndPointDemux* class is responsible for demultiplexing packet to the appropriate transport protocol. It acts as a lookup table matching packets' four-tuple (i.e. source IP address, destination IP address, source port and destination port) to an *Ipv4EndPoint* object. If a match is found, it then calls the *Ipv4EndPoint::Forwardup()* function.

   We implemented the MPTCP token lookup mechanism in the *Receive()* function of *TcpL4Protocol*, so that a match for an *Ipv4EndPoint* object can be found based on a 32-bit token instead of 4-tuple. *Receive()* extracts the TCP options (if available) from the TCP header and searches for specific MPTCP options that carry a token, e.g. MP-JOIN. If a match is not found via the token lookup function, *Receive()* continues its search for an endpoint via the *Ipv4EndPointDemux::Lookup()* function. If a match is still not found, the packet

is discarded and the RST packet is sent to the sender address of the packet.

9. The *Ipv4EndPoint* class has a callback which allows a socket object to register its *Receive()* function. In this case study, this callback calls to the *ForwardUp()* function of *MpTcpSocketBase*. *Forwardup()* is a polymorphic function to allow other extensions of the MPTCP protocol such as MMPTCP and PS to receive data via the *MpTcpSocketBase* class.

10. The *MpTcpSocketBase* class informs the application when a data is ready to be read, by triggering a callback (*Socket::m_receivedData*) that is registered by the application on a socket. In this example, *MpTcpSocketBase* calls to *HandleRead()* of the *MpTcpPacketSink* class via the *m_receivedData* callback.

4.6 MPTCP Signalling Operation

All MPTCP operations are signalled via the option field of TCP header. Our MPTCP signalling model closely follows the specifications set in RFC 6824 [14]. Figure 4.6 shows the key MPTCP operations over the course of a connection. These operations are discussed in the next subsections.

4.6.0.1 Connection Establishment

MPTCP connection establishment follows the standard TCP three-way handshake. The client attaches an MP-CAPABLE option in the SYN packet to denote its MPTCP support. The SYN packet is sent to the MPTCP server and the sender's subflow state is changed from CLOSED to SYN_SENT, as shown in Figure 4.6. The MP-CAPABLE option is only carried at connection setup via initial subflow and it includes the sender's local token in SYN and the receiver's local token in SYN-ACK packets.

In our model, when the MPTCP server receives the SYN packet, it is forwarded to the listening *MpTcpSocketBase* object, based on the destination port number indicated in the TCP header of the received packet, which, in turn, forks a new *MpTcpSocketBase* object to handle communication with the client.

The newly forked *MpTcpSocketBase* object creates its initial subflow with an associated endpoint, which has complete 4-tuple information, i.e, src IP (server), dst IP (client), src port (server) and dst port (client).

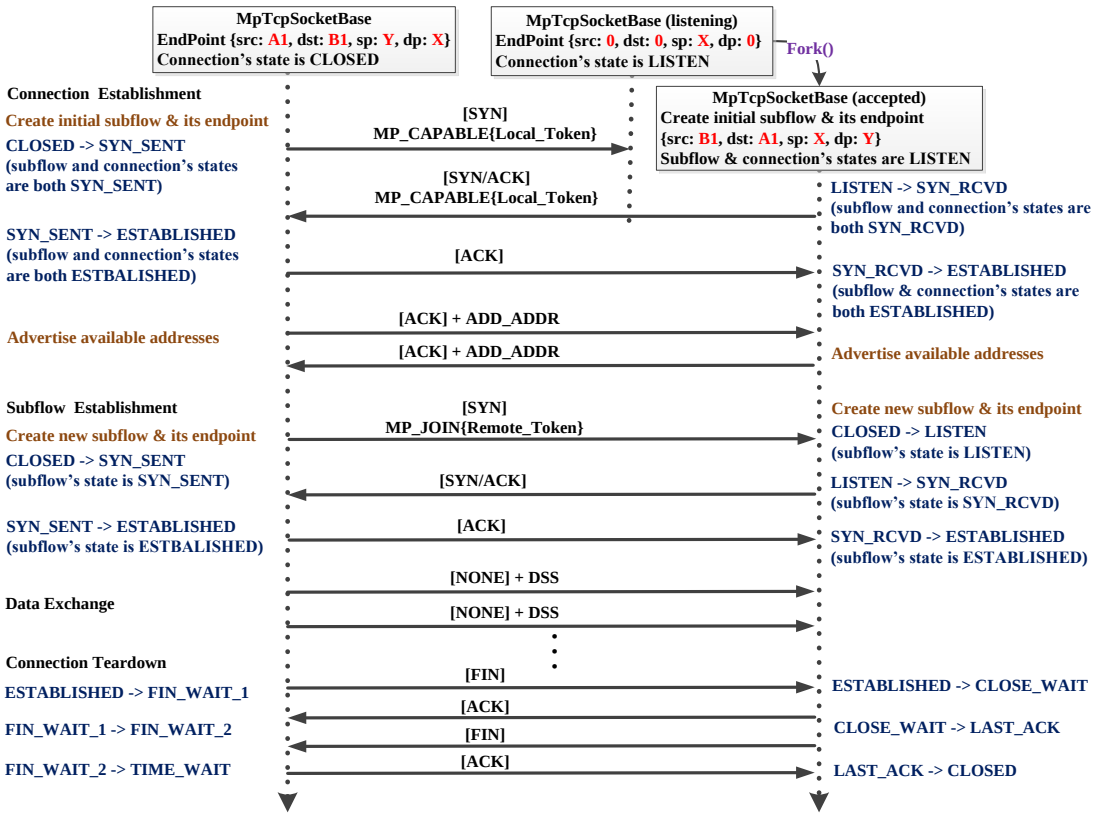


Figure 4.6: MPTCP signalling from the beginning to the end of a connection

For example, as illustrated in Figure 4.6, the client's initial subflow has an endpoint information of src: A1, dst: B1, sport: Y, dport: X sends the SYN packet to destination port number X. The server's *TcpL4Protocol* class searches for a match based on the packet's 4-tuple information, and so the packet is forwarded to a corresponding *MpTcpSocketBase* if a match is found. In this example a match is associated with a listening *MpTcpSocketBase*. The listening socket cloned itself to a new *MpTcpSocketBase*. The cloned object creates its initial subflow with an endpoint with full information of the 4-tuple: src: B1, dst: A1, sport: X, dport: Y. The 4-tuple information corresponds to the initial subflow of both communicating nodes in an MPTCP connection.

In response to the SYN packet, if the server supports MPTCP, an MP-CAPABLE option is attached to the SYN-ACK packet. The server sets the acknowledgment filed of the the SYN-ACK to one more than the sequence number of the SYN packet received from the client. The initial subflow state is also changed from LISTEN to SYN_RCVD.

A 32-bit token randomly generated by the client and server is carried in the MP-CAPABLE option in the SYN and SYN-ACK packet respectively. The token is mapped to the respective *MpTcpSocketBase* via an *Ipv4EndPoint* pointer; this mapping is stored

in the *TcpL4Protocol* object, which is used to associate new subflows to a currently established MPTCP connection.

4.6.0.2 Subflow Establishment

After an MPTCP connection has been established, the communicating endpoints advertise their available IP addresses to each other using the ADD-ADDR option. The new IP addresses, for advertising, could be from single or multiple network devices. Multiple ADD-ADDR options can be accommodated in a single packet. Address advertisement is the responsibility of the Path Manager component.

In our model, the Path Manager component is implemented similarly to the Linux Kernel implementation [78], and provides the following modes of operation:

- **Default:** MPTCP does not advertise its addresses; it operates similarly to the TCP connection. If this mode is selected, data transmission is started immediately after the connection establishment.
- **FullMesh:** MPTCP advertises all IP addresses and attempts to create a full mesh-like topology between the IP addresses of client and server, i.e. each IP address of the client attempts to establish a connection to all IP addresses of the server.
- **NdiffPorts:** The client node initiates subflows based on random source ports, immediately after connection establishment. If this mode is selected, the handshake of the address advertisement is skipped and the data transmission is started immediately on the initial subflow. NdiffPorts effectively allows the endpoints to use port-based load balancing with MPTCP, if the network routes different ports over different paths, which is the case with ECMP routing [9].

Each new subflow can be later established using a TCP three-way handshake and by attaching the MP-JOIN option in the SYN packet. The token is also attached so that the receiving side (server) can resolve the subflow establishment request to an existing MPTCP connection, as described in the previous section.

4.6.0.3 Data Exchange

MPTCP uses two separate sequence number spaces, one per-connection (64-bit) and one per-subflow (32-bit) [14]. The former is used for packet reordering and loss recovery at connection level and it is signalled by the Data Sequence Signal (DSS) option,

which is placed in the option field of TCP header. The latter is used for the same reasons at a subflow level and is carried in the sequence number field of the TCP header. DSS option is carried in each data segment sent from the sender. The receiver sends an ACK packet per data segment received.

4.6.0.4 Connection Teardown

As with standard TCP, each subflow terminates after four-way FIN handshake. An MPTCP connection is also terminated at a connection level by signalling a DATA-FIN option. In our current implementation of MPTCP, the connection teardown is performed at a subflow level only. The FIN packets to all subflows are issued when the last data segment leaves the host. After all subflows have been closed, MPTCP endpoints deallocate all resources. The token mapping and *MpTcpSocketBase* object are also deleted from the *TcpL4Protocol* object.

4.7 MMPTCP and Packet Scatter

We implemented the MMPTCP and Packet Scatter (PS) protocols by inheriting them from MPTCP and overwriting key functions related to congestion control and data management. The key ns-3 classes concerning MMPTCP and PS implementations and their interactions are presented in Figure 4.7.

When a packet is received, the *ForwardUp()* function of the *MpTcpSocketBase* class is called from lower layer (*TcpL4Protocol*). This function then redirects the call to another function called *DoForwardup()* for processing the received packet. The *DoForwardup()* is a polymorphic function so that it simply redirects the call from *MpTcpSocketBase* to *MMpTcpSocketBase* or *PacketScatterSocketBase*. In this way, we only re-implement a few functions of the *MpTcpSocketBase* class in the *MMpTcpSocketBase* and *PacketScatterSocketBase* classes.

The *SendingPendingData()* function in the *MMpTcpSocketBase* class disseminates each packet via a random source port number. We defined a new parameter called *m_packetScatter* to keep the state of a connection, indicating whether a connection is in PS phase or MPTCP phase. We also added *m_remoteToken* parameters to the DSS option, so that the receiver node can understand whether the incoming packets are from PS mode or MPTCP mode. If a PS packet is received it can be forwarded to a correct *MMpTcpSocketBase* object by a MPTCP token provided in the DSS option. The

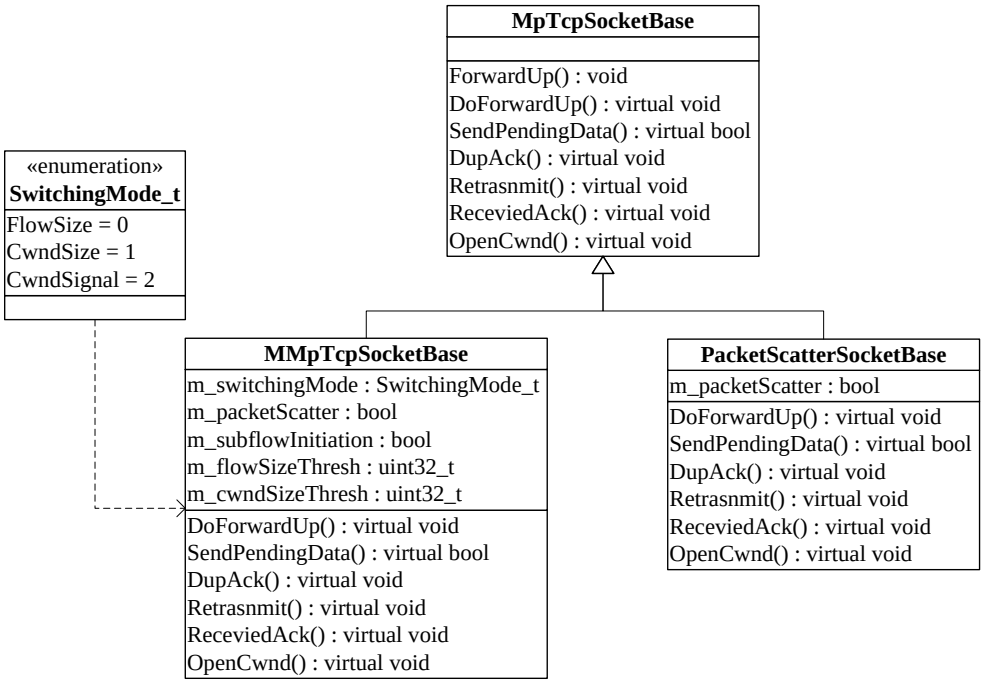


Figure 4.7: MMPTCP and PS class diagram

MMPTCP sender runs TCP congestion control in the PS mode and MPTCP congestion control in MPTCP mode.

MMPTCP switches to MPTCP mode as follows: when the predefined switching threshold is reached, MMPTCP initiates new MPTCP subflows and changes the state of the parameter *m_subflowInitiation* in order to prevent the initiation of further subflows in future. Meanwhile, the initial subflow of MMPTCP continues its data transmission through packet-scattering until at least one of the new initiated subflows is established. Then, MMPTCP deactivates the initial subflow, changes the state of the *m_packetScatter* parameter to false to prevent further packet-scattering, and continues its data transmission via MPTCP mode. However, after the initial subflow is deactivated, MMPTCP continues its loss recovery process on the initial subflow until all of the outstanding packets of that subflow have been acknowledged.

If SYN and SYN-ACK packets of new initiated subflows are not lost, then it is expected that all new initiated subflows are established at the time of switching. In this way, the pipe does not suddenly become empty. Additionally, the new established subflows can fully use the sender's access link capacity very quickly (in a few RTTs). Otherwise, if the initial subflow is deactivated when the switching threshold is reached then either there would be a gap or stall of data transmission due to the delay or loss of

SYN packets of the new initiated subflows respectively.

The *PacketScatterSocketBase* class disseminates all packets via random source ports and it only runs TCP congestion control. It also includes the MPTCP token as a connection identifier in each data segment.

4.8 Showcasing MPTCP, ECMP and PS

4.8.1 MPTCP with Single Subflow

Figure 4.8 illustrates the evolution of the congestion window when an MPTCP connection with a single subflow is opened between two nodes connected via a point-to-point link. The value of the key parameters of this simulation are as follows: the link rate is 100Mbps; the RTT is 2ms; the flow size is 10MB and the queue size is 65pkts. It is expected that an MPTCP flow with a single subflow will operate exactly like a single-path TCP flow. The result of this simulation is shown in Figure 4.8. It is identical to the result achieved from running the same simulation configuration with the existing TCP model in ns-3.

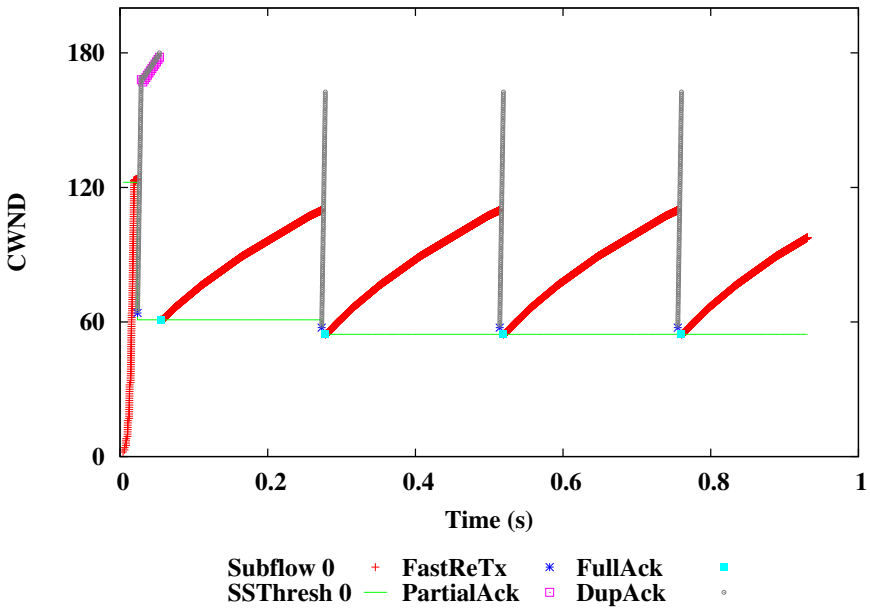


Figure 4.8: Simulation of MPTCP with single subflow running TCP NewReno.

This simple simulation setup illustrates the TCP NewReno[4] loss recovery algorithm in detail. For example, the first packet drop after TCP entered the congestion

[4]A detail study of TCP NewReno is in Section 2.2.1.3.

avoidance phase, can be seen between 0.2 and 0.4 seconds on the x-axis. More than 65 packets were in the flight so the sender's drop-tail queue filled up and a single packet was dropped. The receiver therefore generated duplicate ACKs for all segments received after the lost segment, acknowledging the latter's sequence number. When the TCP sender received the first three duplicate ACKs, the fast retransmission mechanism was triggered, the perceived lost segment was retransmitted and the Fast Recovery phase was started. These retransmissions are indicated by blue star in Figure 4.8.

If any additional duplicate ACKs are received while the sender is in the Fast Recovery phase, the congestion window is increased by one segment and a new segment is sent if possible [79]. The main reason for this small inflation of the congestion window is to prevent a TCP NewReno sender from losing its ACK clock [30], especially when multiple segments were lost in sequence from an entire window's worth of data. The light grey circles key in Figure 4.8 shows this behaviour.

TCP NewReno stays in the Fast Recovery phase until it receives a *full* ACK packet. This special ACK packet acknowledges all the outstanding segments before the sender enters the Fast Recovery phase. After the *full* ACK is received the sender enters the congestion avoidance phase. The blue filled boxes in Figure 4.8 illustrate this behaviour.

Any other new ACK packets received before *full* ACK are called *partial* ACK. TCP NewReno treats this especial ACK as an indication of loss. It thus assumes that the next segment after *partial* ACK has been lost. This is the way that the TCP NewReno recovers from multiple packet drops from a single window's worth of data [80]. Partial ACK attempts to prevent the TCP sender from triggering the retransmission timer on a perceived lost segment by recovering it in the Fast Recovery phase. This behaviour can be clearly observed in the pink boxes in Figure 4.8.

4.8.2 MPTCP Loss Recovery

In order to validate our implementation of TCP NewReno's loss recovery algorithms in the MPTCP subflows, we set up a simulation with a configuration similar to the one presented in Section 6.3 of the well-known research paper "Simulation-based Comparisons of Tahoe, Reno, and SACK TCP" by Kevin Fall and Sally Floyd [81]. The simulation was conducted with MPTCP with a single subflow. The value of the key parameters of this simulation are as follows: the link rate is 800Kbps; the RTT is 100ms;

the flow size is 100KB and the queue size is 100pkts. The result is shown in Figure 4.9 and is very similar to [81](Figure 3).

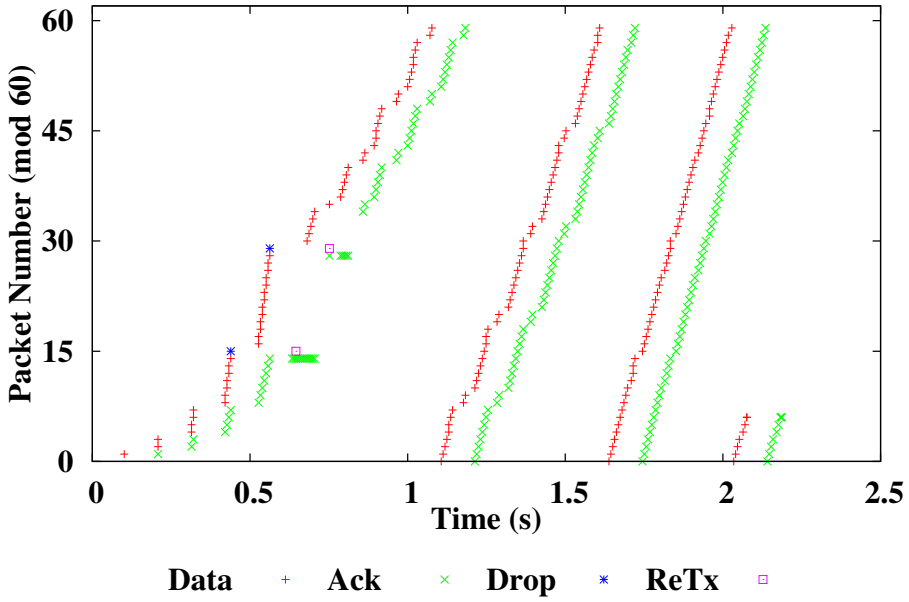


Figure 4.9: Simulation of MPTCP with single subflow and two packet dropped.

Analysis. The TCP Slow-Start behaviour can be clearly seen in the first five RTTs: as the sender received new ACK packets (green points), it sent the new data packets (red points). After packet 15 was dropped, the sender received ACK packets up to packet 14 (i.e. 7 ACK packets), and so it sent packets number 16-29. However, packet 29 was also dropped. Therefore, the TCP receiver was waiting for packet 15 to arrive, but it received packets 16-28, which generated 13 duplicate ACKs to the sender to signal that packet 15 was missing. When the sender received the first three duplicate ACKs, it retransmitted packet 15 and entered the Fast Recovery phase. These 13 duplicate ACKs allowed the sender to send new packets 29-33. When the sender entered Fast Retransmit the congestion window decreased from 14 to 7 and then followed by further increase of 3 segments (i.e. *cwnd* = 10). Thereafter, the sender received 10 more duplicate ACKs, which in turn caused the further inflation of the congestion window (i.e. increase of one full-size segment per duplicate ACK). As the sender was holding 15 unacknowledged packets in its sending buffer (i.e. packets 15-29), it was only allowed to send 5 new packets (i.e. packets 29-33). At this point, the next ACK received by the sender is the partial ACK since it is acknowledging packet 28.[5] The sender interpreted the partial

[5]This ACK has been generated by the receiver when it received the retransmitted packet 15.

ACKs as an indication that the next packet after this has been lost, so rightly the sender retransmitted packet 29 immediately. Following this retransmission, the sender sent packet 35. The reason for this further new packet transmission is to allow the TCP sender to maintain its ACK clock. Inflating the congestion window by one segment whenever *partial* ACK is received would follow the *conservation of packets* principle. These behaviours can be seen clearly in Figure 4.9.

4.8.3 MPTCP Timeout Mechanism

In order to examine the TCP timeout behaviour in the subflows of MPTCP, we ran the previous simulation but with packet drops targeting an entire window's worth of data, thus forcing the triggering of the timeout mechanism. Figure 4.10 shows how the loss-recovery mechanism operates when the entire window worth of data has been dropped (i.e. when TCP lost its ACK clock). After the entire window's worth of data had been dropped, the TCP sender started to recover the lost segments via the retransmission timeout mechanism, which was triggered after 200ms of not receiving any acknowledgement packet. When timeout was triggered the congestion window was reset to one segment and the Slow-Start Threshold (*ssthreshold*) value was adjusted to halve the number of packets in the flight[6] so that the TCP sender entered the Slow-Start phase and stayed in this phase as long as the *cwnd* was less than the *ssthreshold*.

The Slow-Start phase of TCP congestion control can be seen at the first two RTTs at the beginning of the data transmission and at the first RTT after the retransmission timeout. TCP entered at the congestion avoidance phase at the second RTT after the retransmission timeout since the $cwnd \geq threshold$ in such a condition. During congestion avoidance, TCP increased the *cwnd* by one full-sized segment per RTT.

4.8.4 Multipath Congestion Control

We discussed several multipath congestion control algorithms including MPTCP congestion control, known as Linked Increases [29], in Chapter 2. In this section we aim to showcase them in a simple simulation scenario.

Simulations conducted in a FatTree network topology with 128 nodes providing full bisection bandwidth as shown in Figure 4.11. The traffic matrix used is Permutation in which each node sends continuous traffic to one randomly selected node via a

[6]In this simulation, the *ssthreshold* was reset to 2 segments as only 4 packets were in the flight.

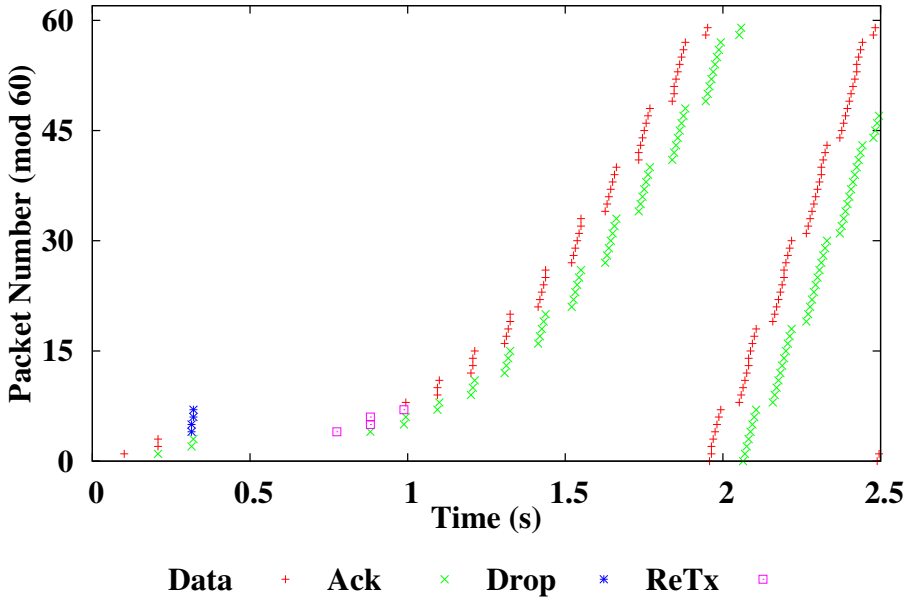


Figure 4.10: Simulation of MPTCP with single subflow and an entire window dropped

MPTCP flow with two subflows. The simulation parameters are as follows: the link speed is 100Mbps; the link delay is 20μs; and the queueing discipline is a drop-tail queue with size of 100pkts. Figures 4.12, 4.13 and 4.14 show the results of MPTCP with two subflows, running the Uncoupled-TCP, Fully Coupled and Linked Increases congestion control algorithm respectively, from a randomly selected node.

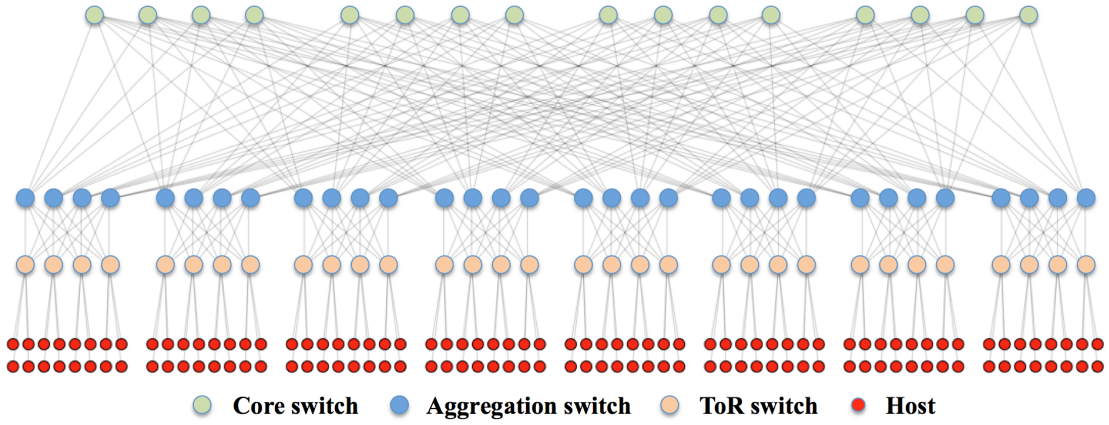


Figure 4.11: FatTree 128 nodes providing full bisection bandwidth.

As expected, Uncoupled-TCP is shown to be more aggressive than Fully Coupled and Linked Increases since each subflow is running TCP congestion control independently. The aggressiveness of each subflow of Uncoupled-TCP is irrelevant to the loss rate. The Fully Coupled algorithm keeps most of its traffic on subflow 1 (SF 1) as it has

a lower loss rate (SF 1 only has one window reduction in first 2 seconds). In Section 2.2.2.1, we discussed that the Fully Coupled algorithm decreases the congestion window of subflow (s) by $\frac{w_{total}}{2}$ when a loss event is detected. If the subflow s has a small window size then the result of this reduction would be a negative number so that the congestion window is then reset to two full-sized segments. In fact the Fully Coupled algorithm removes its traffic from perceived congested paths in this manner. This behaviour can be observed at any loss event of SF 0 in Figure 4.13. The Linked Increases keeps more traffic on SF 0, but also maintains a significant amount of its traffic on SF 1 since both paths in use seem to have similar network conditions. In fact the Linked Increases algorithm actively measures network conditions, i.e. RTT and drop probability, by calculating the a parameter.

It is worth mentioning that Linked Increases puts more traffic on subflows with lower RTTs, if all subflows have the same loss rates. To justify this behaviour we extracted the RTT estimation at the time of each congestion window change, as depicted in Figure 4.15. The result is as expected, as both subflows seem to have the same loss rate but the SF 0 (RTT 0) is slightly lower than SF 1 (RTT 1), so SF 0 has a higher window size, i.e. higher throughput.

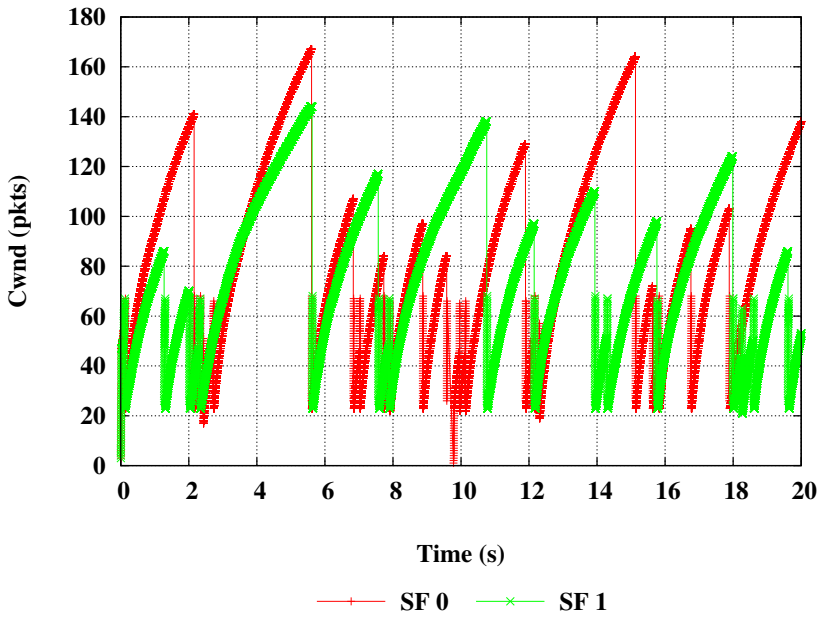


Figure 4.12: Congestion window changes with Uncoupled-TCP

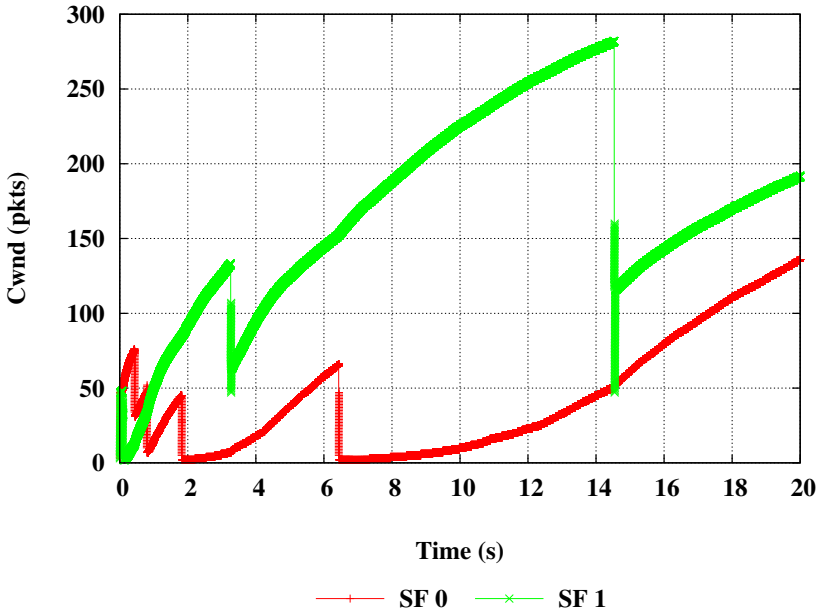


Figure 4.13: Congestion window changes with Fully Coupled

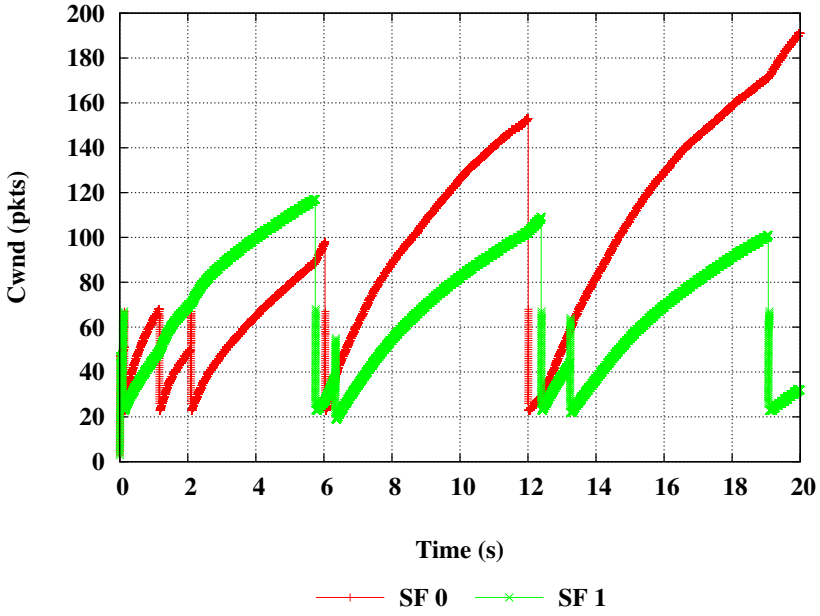


Figure 4.14: Congestion window changes with Linked Increases

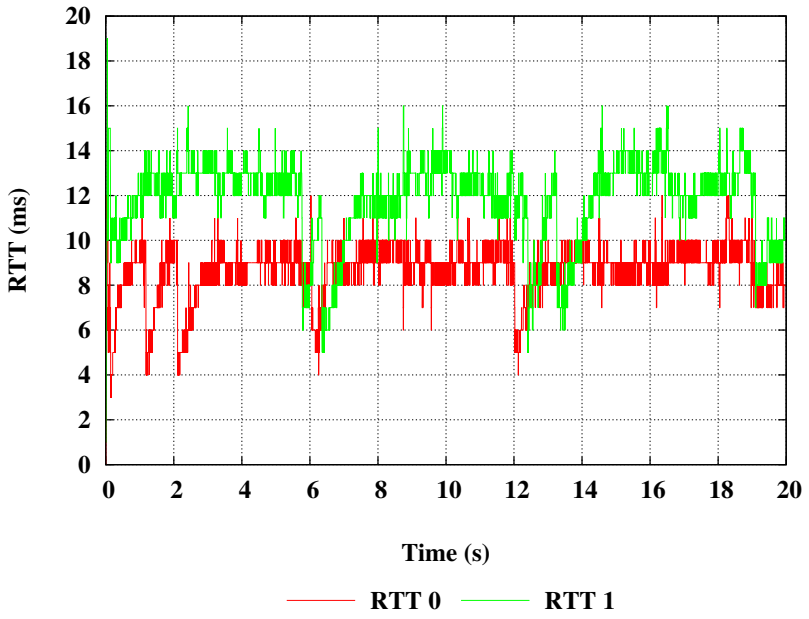


Figure 4.15: RTT estimations as congestion window changes with Linked Increases

4.8.5 ECMP and PS

We implemented ECMP per-flow in ns-3 with the Murmur3 64-bit hash function [82]. To showcase our implementation, we conducted a simulation in a FatTree topology with 128 nodes providing full bisection bandwidth (Figure 4.11). The traffic matrix used is Stride in which each node only has one incoming and one outgoing connection and all network flows should traverse via the network core (See Section 5.2.2 for a detailed discussion of the Stride traffic matrix). The transport protocol used is PS running only long flows. With this setup, equal utilisation of all links in the network core is expected. Additionally, as the network load is equal between end-hosts due to the Stride traffic matrix and the network provides full bisection bandwidth then the network core should be fully utilised if ECMP diffuses the network traffic evenly. Figure 4.16 shows the results. All 16 core switches of the FatTree topology are almost fully utilised and also all links in the network core are equally utilised. The Jain's fairness index [83] of flow goodputs is also 0.999097, indicating that all PS flows achieved their fair share. The value of Jain's fairness index is between 0 and 1, where 1 implies perfect fairness between contending flows (i.e. all flows share bandwidth equally).[7] The result is therefore convincing: that our implementation of ECMP per-flow works as expected.

[7]We calculated the Jain's fairness index with the following formula: $F(x) = \frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}$. The fairness index $F(x)$ considers n flows where the goodput of flow i is x_i. The value of n is 128 in this experiment.

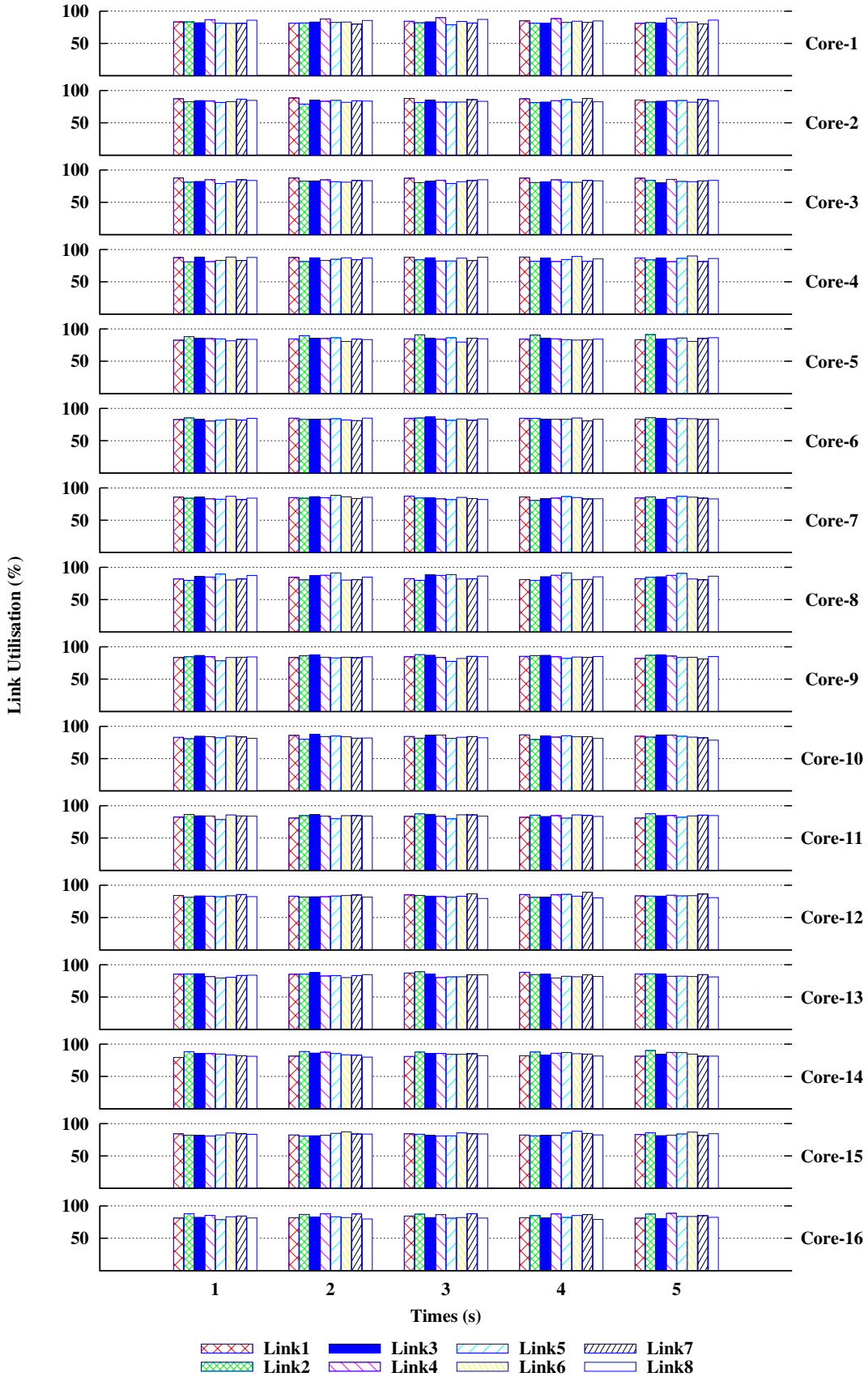


Figure 4.16: ECMP, PS and the Stride traffic matrix in a FatTree topology with 128 nodes. Each plot shows the link utilisation of all eight links of a core switch per second.

4.9 Summary

This chapter first provided an overview of the TCP architecture in ns-3. It then explained our implementation of MPTCP in ns-3. This was followed by an overview of our proposed MPTCP architecture; a detailed explanation of key MPTCP modules and their interactions; an overview of the ns-3 networking stack with the aim of highlighting our contributions (e.g. implementation of per-flow ECMP); a brief review of MMPTCP and PS implementation, and, finally a showcasing of the algorithms used in MPTCP via simple simulation setups.

Chapter 5

Evaluation and Results

5.1 Introduction

This chapter studies the performance of MMPTCP in various network conditions. It also compares MMPTCP with other existing transports, such as TCP, MPTCP and Packet Scatter (PS).

In this thesis, we conducted all of our simulations in a FatTree and VL2 topology with various oversubscription ratios ranging from 1:1 to 4:1.[1] We found that all of the transport protocols under consideration (i.e. MMPTCP, TCP and PS) perform well and with small performance differences in a VL2 topology, even with a 3:1 oversubscription ratio VL2 topology with 960 nodes.[2] However, we observed a larger performance differences between the transport protocols in question when we compared them in a FatTree topology, even with a 2:1 oversubscription ratio FatTree topology with 256 nodes. The main reason is that VL2 has 10 times faster link capacity between its switches than its hosts and switches. As a result, VL2 can handle higher statistical multiplexing of flows in its core and aggregation layers than FatTree. For example, a link in the core of a 1:1 oversubscribed VL2 topology requires at least 5 times more long flows than a 1:1 oversubscribed FatTree topology to become slightly congested. Furthermore, as a VL2 topology has faster link rate between its switches, it decreases queueing delays compared to a FatTree topology. As a result, a PS-based protocol such as MMPTCP experiences less packet reordering due to the imbalanced network queuing delays in a VL2 topology compared to a FatTree topology.

[1] See Section 2.3.2 for a detailed explanation of the FatTree and VL2 topology.

[2] In our VL2 model, we used the link rate of 1Gbps between switches and 100Mbps between hosts and switches.

We have also observed that MMPTCP performed better under a FatTree topology with the link rate of 1Gbps instead of the 100Mbps like rate. The reason is that the 1Gbps like rate can handle traffic surges more gracefully than the 100Mbps link rate so that the network queueing delays will be less imbalanced.

We therefore present all of our results from a FatTree topology in this thesis to highlight the influence of packet reordering more distinctly. Our simulations were also run on the High Performance Computing (HPC) network at the University of Sussex. By default each simulation was run for 20 simulated seconds with 20 repetitions using different seeds. The main reason for running each simulation for 20 simulated seconds is to provide enough data transfer and number of short flows to accurately calculate the goodput of long flows and the flow completion time of short flows respectively. All results are reproducible.

We proceed in the remainder of this chapter as follows. Section 5.2 describes our simulation configurations, traffic matrices and simulation templates. We discuss the adjustment of duplicate ACK threshold during the initial phase of MMPTCP in Section 5.3. Section 5.4 compares MMPTCP with MPTCP with eight subflows. We then compare MMPTCP (running both large and short flows) with MPTCP (running long flows with eight subflows and short flows with a single subflow) in Section 5.5. Section 5.6 compares MMPTCP with TCP and PS. Thereafter we examine the performance of MMPTCP, TCP, PS and MPTCP under different hotspot levels and network loads in Section 5.7 and 5.8 respectively. We explore the performance of MMPTCP with different multipath congestion control algorithms in Section 5.9. Section 5.7 examines MMPTCP performance when the Limited Transmit mechanism is enabled. Section 5.11 studies of the MMPTCP switching mechanism. We then investigate the performance of MMPTCP, TCP, MPTCP and PS under incast scenarios with both short and long flows separately in Section 5.12.

5.2 Simulation Setup

5.2.1 Network Topology

One hurdle in modelling data centre networks is that their sizes are very large (e.g. typically more than 100K servers). The modelling of such network sizes via *packet-level* simulations is a very challenging task as it requires an extensive amount of resources,

especially in terms of memory and CPU cycles.

In the following section, we explore the feasibility of simulation as the network size increases. The ns-3 simulator is a single threaded simulator therefore no parallelisation is possible within a single simulation. Another problem is the memory constraint: as the network size increases, more flows and therefore more packets are generated. As a result memory requirements increase.

To study feasible network sizes, we have conducted simulations in a FatTree topology with a link rate of 100Mbps and an oversubscription ratio of 4:1 with varying network sizes, ranging from 64 to 1728 hosts. One third of the nodes run long flows. The remainder run short flows (70KBs each), which are scheduled by a central scheduler according to a Poisson arrival ($\lambda = 256$). A detailed explanation of our traffic matrices and simulation templates are in subsection 5.2.2 and 5.2.3 respectively. Figure 5.1 depicts the results. It is clear that the simulation duration is directly related to the network size. It turned out that FatTree with a switch port of eight can give us a suitable simulation completion time, being around 15 hours per simulation scenario.[3] Typically, new simulations are designed and run after analysis of the results of previous simulations. Thus, allocating 15 hours to each simulation permitted us to design and execute new simulations on a daily basis.

Network size is not the only factor that impacts the simulation time; the link rates are also very important. The faster these are, the more packets are generated and processed. As a result, simulation becomes slower.[4] To explore this matter, we ran simulations in a FatTree topology with 512 nodes and an oversubscription ratio of 4:1 with varying link rates, ranging from 100Mbps to 1Gbps. Figure 5.2 depicts the results. Link rates higher than 100Mbps would require at least 24 hours to complete a single simulation. We therefore decided to use a FatTree topology with the switch port of eight ($K = 8$) and the link rate of 100Mbps.

[3]We have seen a small impact on the results (e.g. the average core utilisation, overall network throughput or short flow completion times) when increasing the network size in this simple experiment.

[4]Our HPC is equipped with high memory servers (e.g. 250GB), so we were not concerned about the memory consumption in our simulations. However, the average memory consumption of our typical simulations was approximately 700MB.

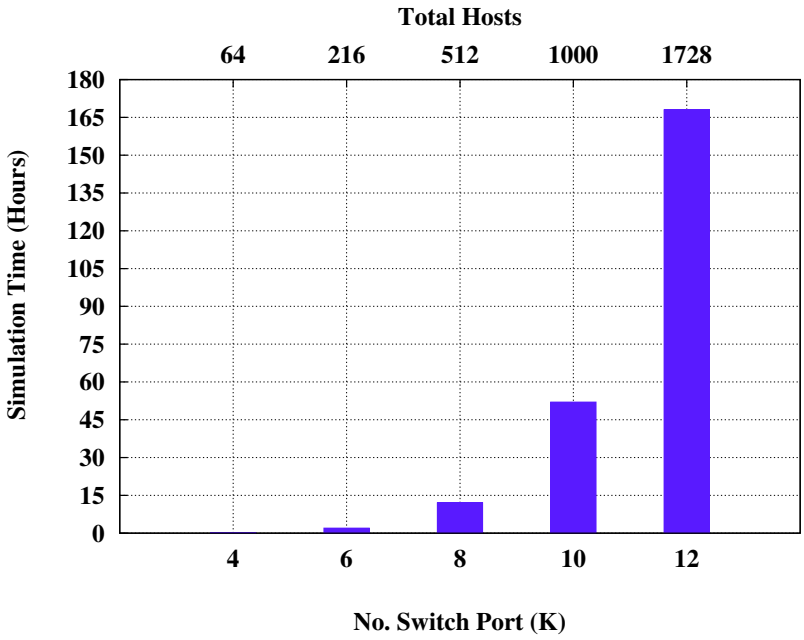


Figure 5.1: Network size has a direct impact on simulation completion time. A FatTree topology with 16 cores (K = 8) provides a suitable simulation completion time.

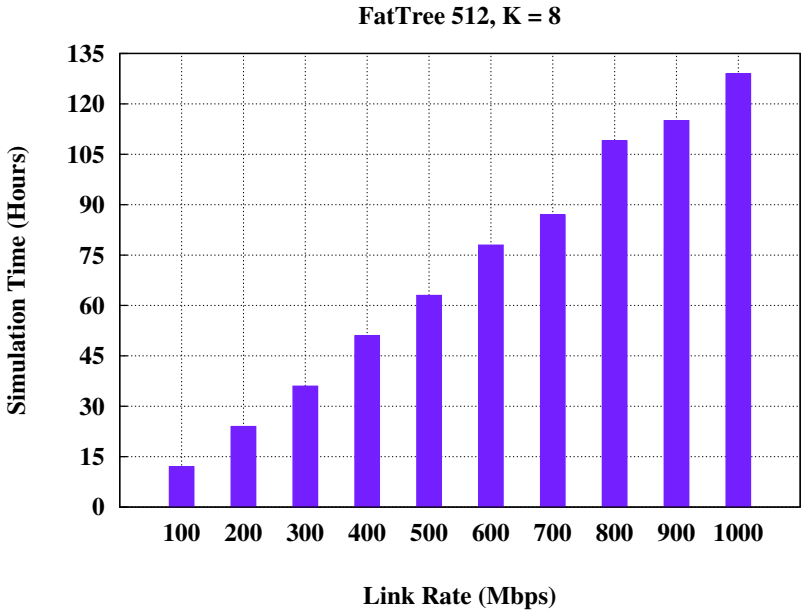


Figure 5.2: Link rate has a direct impact on simulation completion time. A simulation takes more than 120 hours to be completed in a FatTree topology with 1Gbps link rate.

5.2.2 Traffic Matrices

It is certainly very challenging to model realistic data centre traffic patterns, as most data centre owners are not willing to publicly release their workload models. Furthermore, traffic patterns in data centres have been shown to be very volatile and unpredictable [4, 5, 17].

This section contains detailed descriptions of various traffic matrices which have been incorporated into the ns-3 simulator. We employed traffic matrices widely used in the related literature [8, 12].

Stride Matrix: this traffic matrix connects hosts to one another with a specific pattern. For example, a source host with id X finds its destination host with id Y by a Stride matrix with constant I by the following formulae: $Y = (X + I)\ mod\ (total\_host)$.

Figure 5.3 shows an example of a Stride traffic matrix in a FatTree topology with 16 nodes ($total\_host = 16$). Each node connects to another node with $I = 4$. For example, node with id 0 ($X = 0$) connects to node with id 4 ($Y = 4$). If traffic is distributed evenly amount nodes in a Stride traffic matrix, then each node is expected to transmit data at its line rate.

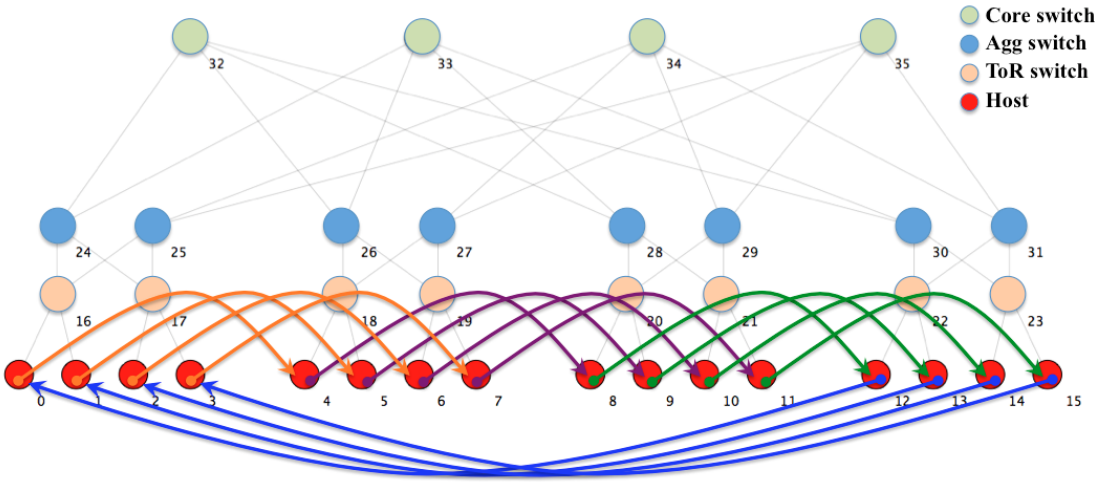


Figure 5.3: Stride traffic matrix in a FatTree topology with 16 nodes

In this model, each node only has one incoming and one outgoing connection and all flows will be routed through the core layer. The Stride traffic matrix provides a good presentation of how network resources are shared among flows. For example, if the network flows are distributed as evenly as possible throughout the core layer, the aggregated throughput of all flows should be in line with the aggregated capacity in

the core layer. The performance of different routing or transport layer protocols can thus be compared. The Stride traffic matrix can be a useful model to examine network protocols in full bisection bandwidth topologies since the model can simply saturate the network core.

Permutation Matrix: in this matrix, each node randomly selects another node that does not have any incoming connections. Furthermore, each node only has one incoming and outgoing connection and there is no guarantee that a flow crosses the core layer. In other words, a pair of nodes can be connected via a core, aggregation (Agg) or even Top-of-Rack (ToR) switch. Figure 5.4 shows a simple example of this traffic matrix in a FatTree topology with 16 nodes. Node 13 is connected to node 12 via a ToR switch and the rest of the nodes are connected to one another in such a way that their flows traverse the core layer.

The Permutation model is similar to the Stride model; the difference is that the network core may not be fully saturated. For example, if the Permutation traffic matrix generates 10 connections that traverse the network core of a FatTree topology with 16 nodes, then it is inferable that at most 10Gbps of network core is used in theory and the remaining of 6Gbps are unused.[5] This information allows us to compare various transport protocols and infer how each protocol finds and uses unused resources throughout the network.

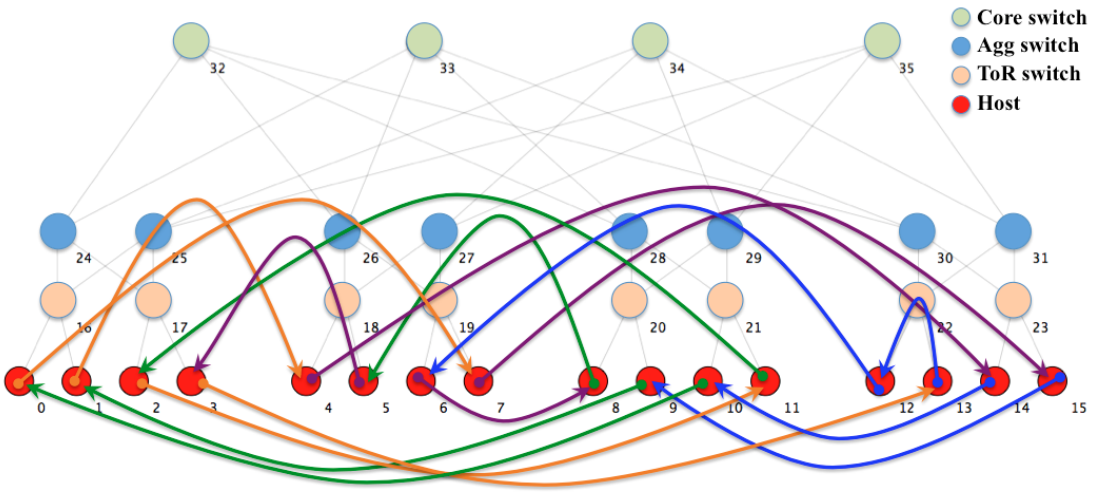


Figure 5.4: Permutation traffic matrix in a FatTree topology with 16 nodes

[5]The core layer of FatTree with 16 nodes provides a total capacity of 16Gbps (16 servers × 1Gbps).

Random Matrix: in this matrix, the source and destination node of a connection is selected randomly. In other words, a pair of nodes can be connected via a core, an aggregation or even a ToR switch. Additionally, each node can have multiple incoming and outgoing connections, unlike the Permutation and Stride traffic matrices. This implies that traffic concentration at the access layer is likely to occur. For example, in Figure 5.5, nodes 3 and 8 have three incoming connections and one outgoing connection, and node 9 has only two outgoing connections.

The Random traffic matrix may provide a more realistic model of data centre traffic, compared to the Permutation or Stride traffic matrices, but the expected resource allocation to network flows is harder to predict.

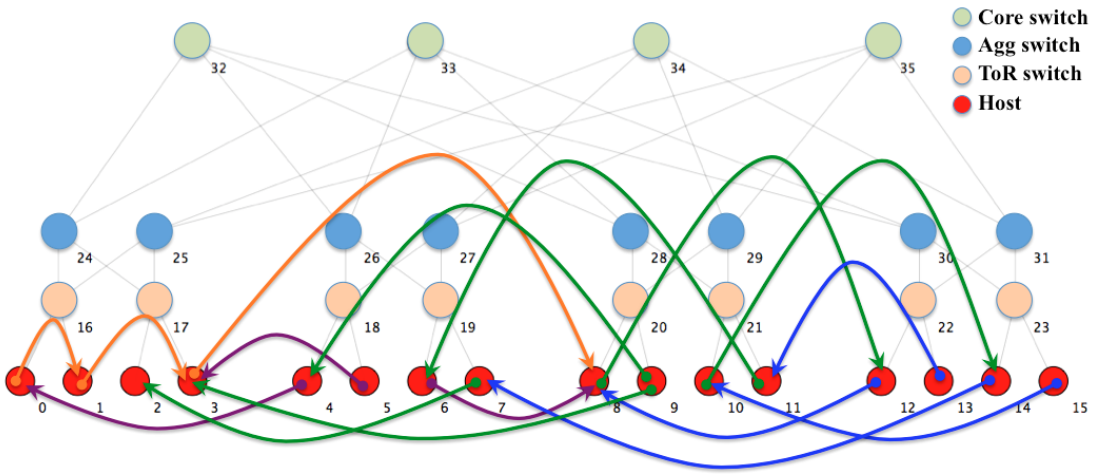


Figure 5.5: Random traffic matrix in a FatTree topology with 16 nodes

5.2.3 Simulation Templates

Our simulations can be broadly categorised into two categories, as follows:

- **Sim_{Mix}. A mixture of large and short flows**. In this simulation template, a fraction of nodes send a continuous flow over the course of the simulation in order to provide enough traffic to congest the network core. The number of nodes in this subset is determined by the amount of traffic necessary to saturate the network core. For example, in order to saturate the network core of a FatTree topology with 512 nodes, an oversubscription ratio of 4:1 and with the Stride traffic matrix, 25% of nodes need to run long flows. For the Permutation traffic matrix this percentage value is higher (e.g. 33%) as not all flows traverse the core

layer. The remainder of the nodes are involved in data exchange only through short flows (e.g. 67% of nodes in the example above).

We also assigned a name to each Sim_{Mix} based on the transport protocol used for short and long flows. For example, if a simulation setup uses MPTCP with eight subflows for both large and short flows, we refer to it as $\text{MPTCP}_{\text{Pure}}$. If a simulation uses MPTCP with eight subflows for long flows and TCP for short flows, we refer to it as $\text{MPTCP}_{\text{SFTCP}}$ (SFTCP is an abbreviation of Short Flow TCP). Table 5.1 depicts these simulation names and corresponding transport protocols.

A simulation can be conducted with different settings. For example, PS_{Pure} can be conducted via a FatTree_{128} topology (i.e. a FatTree topology with 128 nodes providing full bisection bandwidth[6]) or a FatTree_{512} topology with an oversubscription ratio of 4:1. Note that a FatTree_{512} topology with an oversubscription ratio 4:1 has four times more hosts than a FatTree_{128} topology, and the number of core switches is the same in both topologies.

Sim_{Mix} Simulation Name	Short Flow Transport Protocol	Long Flow Transport Protocol
PS_{Pure}	PS	PS
PS_{SFTCP}	TCP	PS
$\text{MPTCP}_{\text{Pure}}$	MPTCP	MPTCP
$\text{MPTCP}_{\text{SFTCP}}$	TCP	MPTCP
TCP_{Pure}	TCP	TCP
MMPTCP	MMPTCP	MMPTCP

Table 5.1: Various Sim_{Mix} simulation names based on employed transport protocols in short and long flows. SFTCP indicates that short flows are handled by the TCP protocol.

Sim_{Mix} is heavily used in this thesis with various settings, for example with different traffic matrices, transport protocols, arrival rates and network sizes. All of our key evaluation metrics can be considered after execution of a Sim_{Mix} simulation. The following are examples of such metrics:

- Average flow completion time of short flows (ms).
- Average goodput of long flows (Mbps).
- Average network utilisation on each layer of a network topology (%).
- Average loss rates on each layer of a network topology (%).

[6] We also refer to the full bisection bandwidth by stating an oversubscription ratio of 1:1.

Central flow scheduler with Poisson arrival. In Sim_{Mix}, short flows are scheduled by a central flow scheduler based on the Poisson arrival with rate λ (arrivals/second).[7] In each arrival, the central scheduler picks a random source node from a set of nodes that are responsible for sending short flows. The selected source node establishes a connection to its destination and sends a flow with a size of 70Kbytes.[8] The central flow scheduler schedules the next arrival based on the Poisson arrival rate (λ). We conducted our simulations with either $\lambda = 256$, or $\lambda = 2560$. The latter generates 10 times more short flows than the former and hence can produce a more bursty traffic pattern but it takes longer to complete a simulation.

The central flow scheduler is designed to provide a high randomness between nodes running short flows. In this way, short flows can randomly compete with long flows in the bottleneck links in a random setting so that the congestion control algorithms of the long flows can be evaluated, e.g. their reaction time to congestion.

Note should be taken that in order to select a size for short flow, we studied various flow sizes, ranging from 10KBs to 1MB. It turned out that the flow size of 70KBs is suitable for analysing the influence of packet ordering on a random packet spraying based transport protocol with a window based congestion control such as the MMPTCP and PS protocols. The reason is that the flow size of 70KBs is large enough to be recovered from multiple packet reordering events or losses and is short enough to reveal the effect of a single packet reordering event on flow completion time, in a wide range of network scenarios. Furthermore, it prevents a large simulation completion time when a large mean short flow arrival rate is used (e.g. 2560 per seconds). Finally, changing short flow sizes would not change our overall results, conclusions and discussions presented in this thesis.

- **Sim_{Long}. Long flows only.** This simulation template involves long flows which run for the duration of the simulation. This configuration allows us to evaluate

[7] The Poisson arrival is a good approximation for modelling the short flow arrival in data centres [51, 13, 52, 84]. The Poisson arrival can be modelled via exponential distribution with rate λ.

[8] The connection between nodes is determined at simulation configuration time based on the traffic matrix used. This implies that a connection's source node, but not its destination node, can be determined randomly by a central flow scheduler.

the performance of different networking protocols under various network loads. For example, we can use the Stride matrix with a Sim_{Long} simulation to evaluate the overall network throughput of TCP versus MPTCP under heavy load in the network core of a full bisection bandwidth topology.

5.3 MMPTCP and Duplicate ACK Threshold

In this section, we review the adjustment of the duplicate ACK threshold value (*dupthresh*) during the initial phase of MMPTCP. We then examine our proposed solution for *preventing* spurious retransmissions due to packet reordering, as described in Section 3.6.

MMPTCP randomises its traffic via all possible paths during its first phase, and so receivers may receive out-of-order packets, and generate duplicate acknowledgements (duplicate ACKs) to the sender, for a segment that is expected to be received in-order. TCP NewReno cannot distinguish between the signal received for packet reordering and packet drops. This may cause the fast retransmission mechanism to be triggered unnecessarily; we refer to this as *spurious retransmission*.

As discussed in Section 3.6, one solution for *preventing* spurious retransmissions as a result of packet reordering is to increase the *dupthresh* value so that the TCP sender can react to loss signals with more tolerance. To better understand packet reordering in our model of data centre, we conducted a series of simulations with a varying *dupthresh* value ranging from 3 to 23. Simulations were conducted in a FatTree_{128} topology running short and large MMPTCP flows.[9]

The result is shown in Figure 5.6, which depicts short flow completion times as a function of *dupthresh* value. It is clear that the default TCP duplicate ACK threshold of three achieved the worst average flow completion time (158ms). However, by increasing the value of the *dupthresh*, the average flow completion time decreases significantly to a *dupthresh* of eight. After that the result remains unchanged.

The main obstacle for adjusting the *dupthresh* is that if its value is very large then TCP fast retransmission might not be triggered, especially when the congestion window

[9]FatTree with 128 nodes has 16 core switches which in turn provide 16 equal-cost paths to any pair of nodes that need to send their data via the core layer. The number 16 may be a good indicator of the maximum distance that a packet may be reordered in the stream of packets between two communicating nodes in this topology.

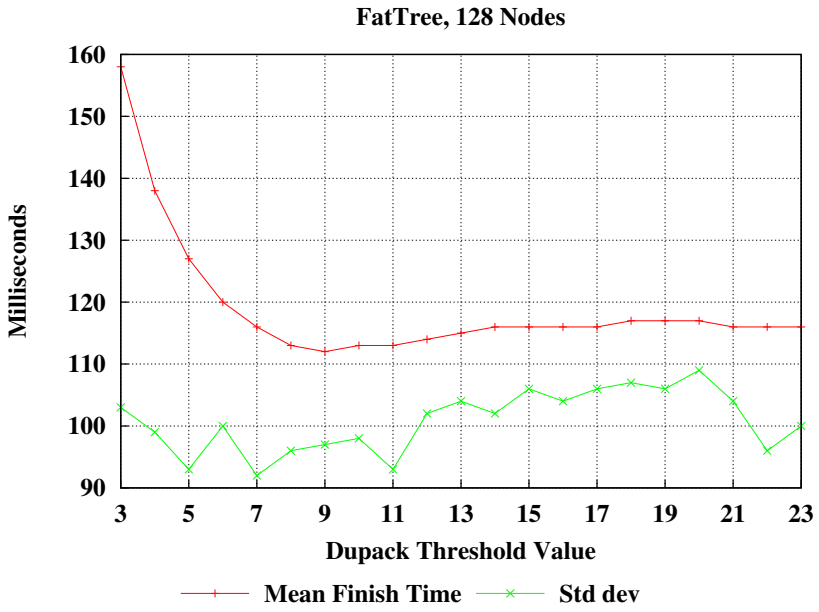


Figure 5.6: Duplicate ACK threshold value effect on short flow completion time

is smaller than the *dupthresh* or when multiple packets get dropped in sequence. In these scenarios, TCP needs to wait until the retransmission timer is triggered because it cannot place any new data packets into the pipe due to the lack of ACK packets in transit. On the other hand, if the *dupthresh* value is very small then fast retransmission might be triggered unnecessarily and frequently, which may cause the TCP sender to needlessly halve its rate and retransmit spuriously.

To get a better grasp of the problems, we present simulation results related to the number of fast retransmissions and timeouts experienced by each short flow. Figures 5.7, 5.8 and 5.9 show the simulation result for a *dupthresh* of 3, 23 and 9 (two extremes and one best case scenario from Figure 5.6). At one extreme, which is related to a *dupthresh* of three, we observed the highest fast retransmission hits and lowest timeout hits. At another extreme, which is related to a *dupthresh* of 23, we observed no fast retransmission hits and high timeout hits. The best performance is observed when the threshold is set to nine (Figure 5.9). The majority of flows were completed without any fast retransmission or with only one fast retransmission; a few flows experienced two fast retransmissions. The density of timeout at line one is also slightly lower than the *dupthresh* of 23 (Figure 5.8).

The results of this experiment do not lead to any concrete value for the *dupthresh* since they are only valid for this particular network setup. This implies that by altering

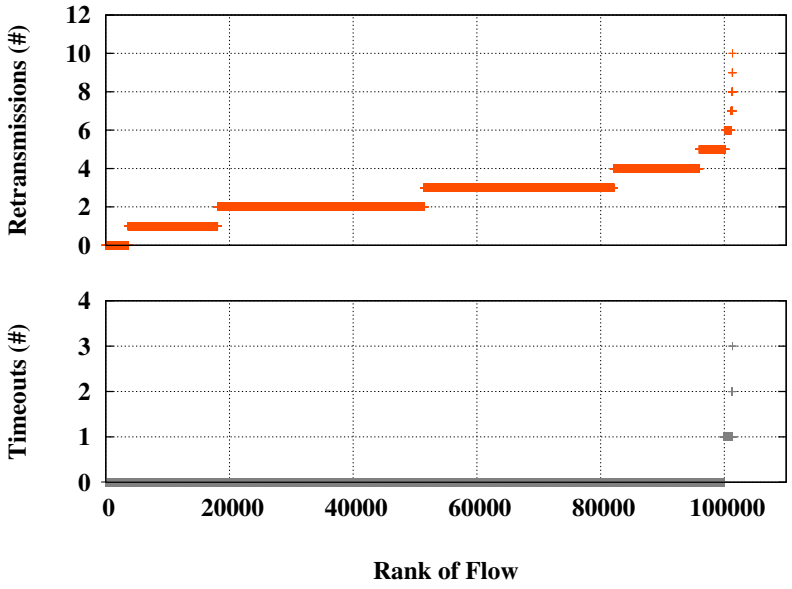


Figure 5.7: A *dupthresh* of 3. High fast retransmission and low timeout hits

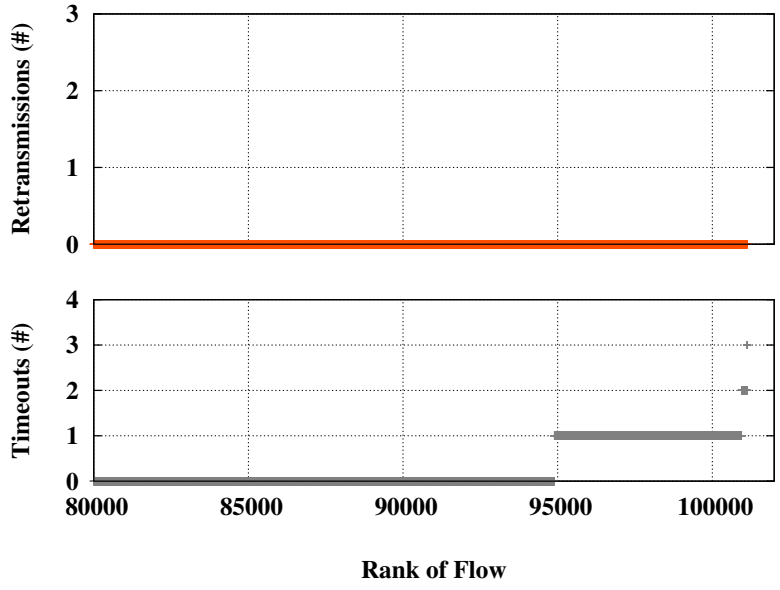


Figure 5.8: A *dupthresh* of 23. No fast retransmission and high timeout hits

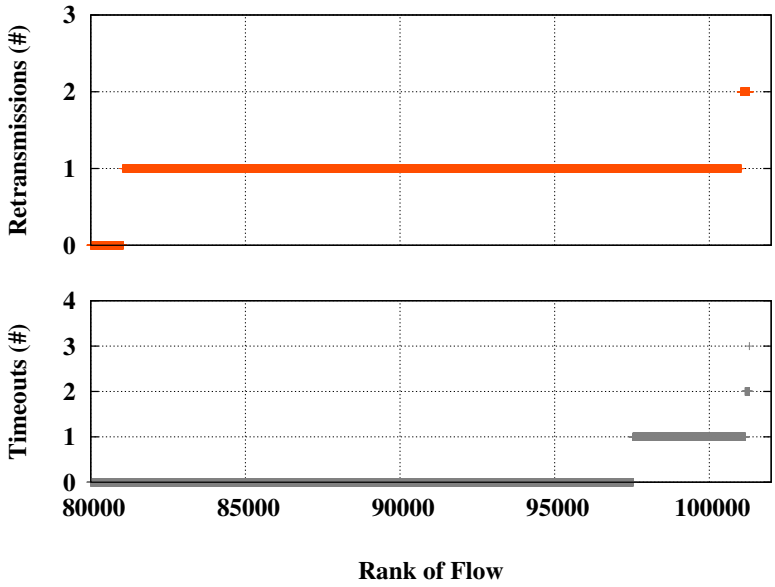


Figure 5.9: A *dupthresh* of 9. Ideal outcome

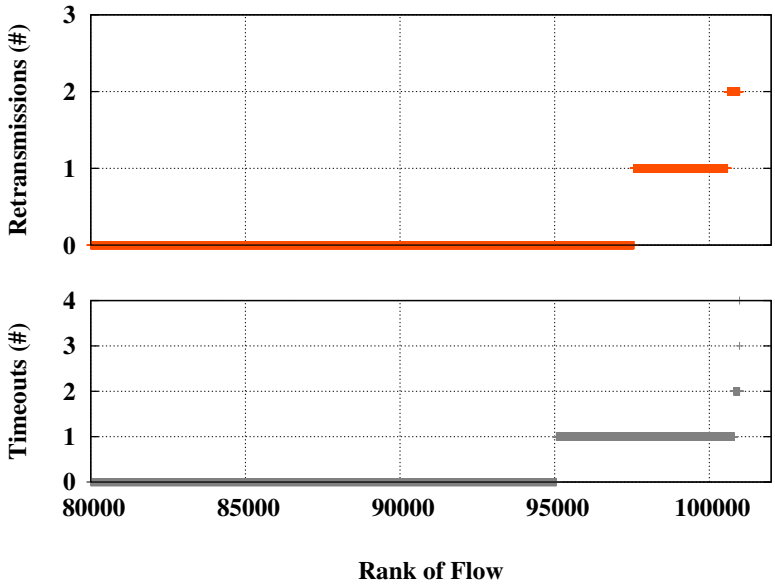


Figure 5.10: Our solution for adjusting a *dupthresh* value based on the FatTree IP addressing scheme. The achieved results are similar to a *dupthresh* value of 9 in Figure 5.6.

the network topology, e.g. its size or traffic matrices, the ideal value selected from the above experiment may not be ideal at all. Another issue with selecting a single ideal value for the *dupthresh* is that most of the traffic may be localised in ToR switches in which they do need to have a *dupthresh* larger than the default value of three.

Figure 5.10 shows the result of simulating an auto *dupthresh* solution (i.e. our solution based on the FatTree IP addressing scheme) with the same simulation setup as Figure 5.6. Auto *dupthresh* significantly decreases the number of spurious retransmission due to packet reordering by adjusting the value of *dupthresh* based on topology-specific information. This can be observed by comparing the proportion of nodes with a single fast retransmission at the Retransmissions plots in Figure 5.9 and Figure 5.10. However, by comparing the proportion of nodes with a single retransmission timeout at the Timeout plots one can observe that the auto *dupthresh* slightly increases the number of timeouts compared to *dupthresh* of 9. The main reason is that a TCP NewReno sender may force to trigger a retransmission timer by a single packet lost when its congestion window is smaller than its duplicate ACK threshold; increasing the value of *dupthresh* to a large number can thus slightly increase a chance of experiencing timeouts. In order to improve the performance of auto *dupthresh* and hence MMPTCP in such conditions, we proposed to integrate the TCP limited Transmit mechanism in the initial phase of MMPTCP. We explore the influence of a large *dupthresh* value on short flow completion times in Section 5.5 and examine the performance of MMPTCP with Limited Transmit in Section 5.7.

5.4 Comparing MMPTCP to MPTCP$_{Pure}$

MPTCP is a well-suited approach for long flows in which they can achieve a high network throughput [12]. However, we showed that MPTCP is not appropriate for handling short flows as it can hurt their flow completion times (see Chapter 1).

The important question is: why is MPTCP not good for handling short flows? An answer to this question seems trivial since it is not sensible to open eight subflows for a flow consisting of only few packets, but this may not be the only answer.

In Chapter 2, we studied the design principle of MPTCP congestion control in detail. In short, the MPTCP congestion control algorithm does not completely remove its traffic from the most congested paths. In fact, MPTCP removes its traffic from congested paths exponentially and then places a small amount of new data on these paths until their network conditions are improved. This means that MPTCP keeps some traffic on its subflows in order to actively probe their network conditions. If the traffic on a subflow is very small then even experiencing a single packet drop may lead to the loss of the TCP ACK clock (i.e. no data packet can be sent since no ACK is received). For example, let us assume an application uses a MPTCP with two subflows and intends to transmit a flow worth of 10 packets. Each subflow can therefore transmit five packets via round-robin data scheduling. The trouble here is that if any packet gets dropped at any point during the data delivery, the subflow experiencing this loss event would not be able to recover the lost packet via the Fast Retransmit mechanism, and hence the retransmission timer needs to be triggered as a last resort of the loss recovery process. In other words, a single packet drop from a subflow does hold up the entire MPTCP connection until that lost packet is recovered. In this example, if TCP runs that flow instead, the chance of recovering that lost packet via Fast Retransmit increases significantly.

Keeping some traffic on the subflows that are experiencing congestion seems to be a better approach than removing almost all of the traffic from those subflows when a flow is large. For example, Fully Coupled resets its congestion window to two segments in such a case. That is, MPTCP subflows can maintain their ACK clocks, and hence experience fewer timeouts [40]. However, neither approach is well suited to short flows.

To expand further on the above discussion, we designed a Sim$_{\text{Mix}}$ simulation in a FatTree$_{512}$ topology with 4:1 oversubscription ratio as presented in Figure 5.11. The

transport protocol used is MPTCP with eight subflows running both large and short flows (MPTCP$_{Pure}$). The arrival rate of short flows is 256 arrivals per second and 33% of nodes send long flows.

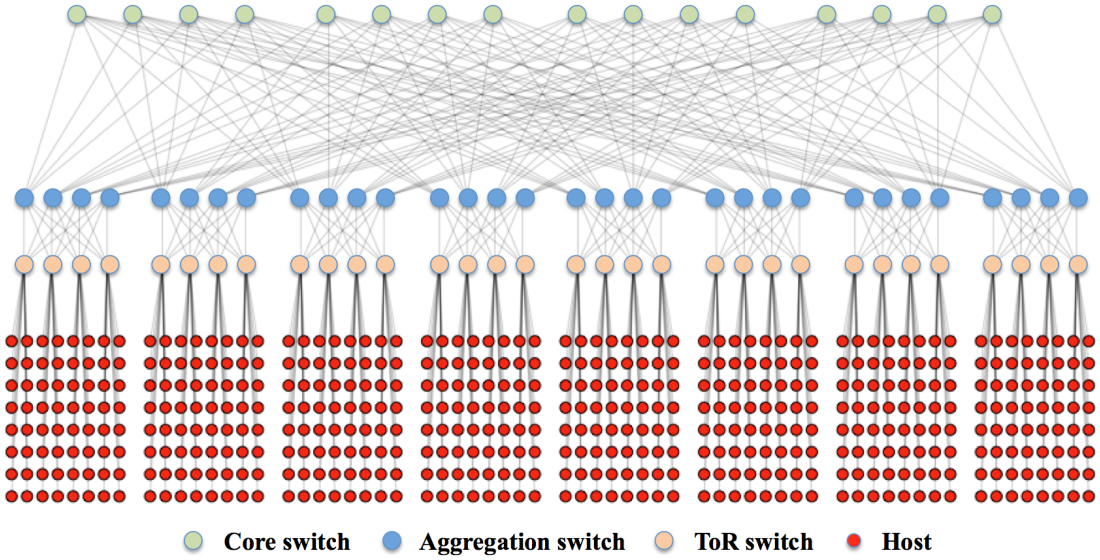
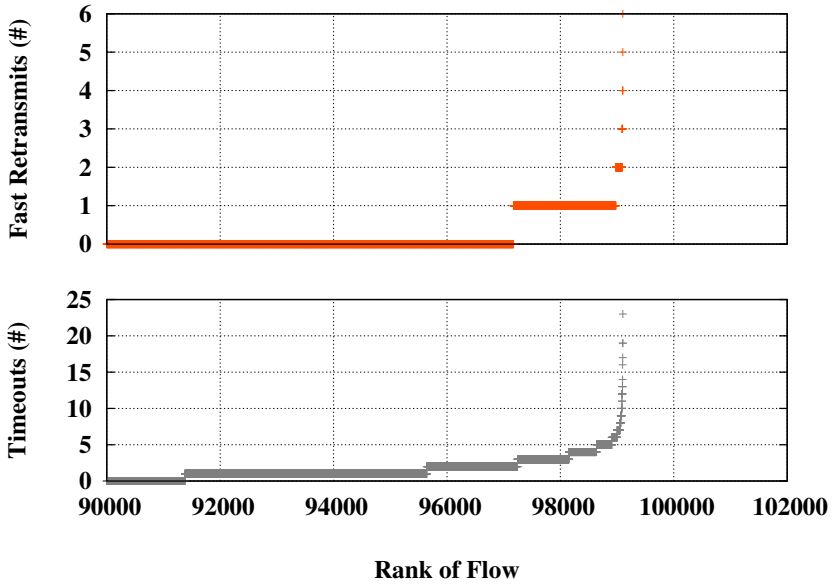


Figure 5.11: A 4:1 oversubscribed FatTree$_{512}$ topology

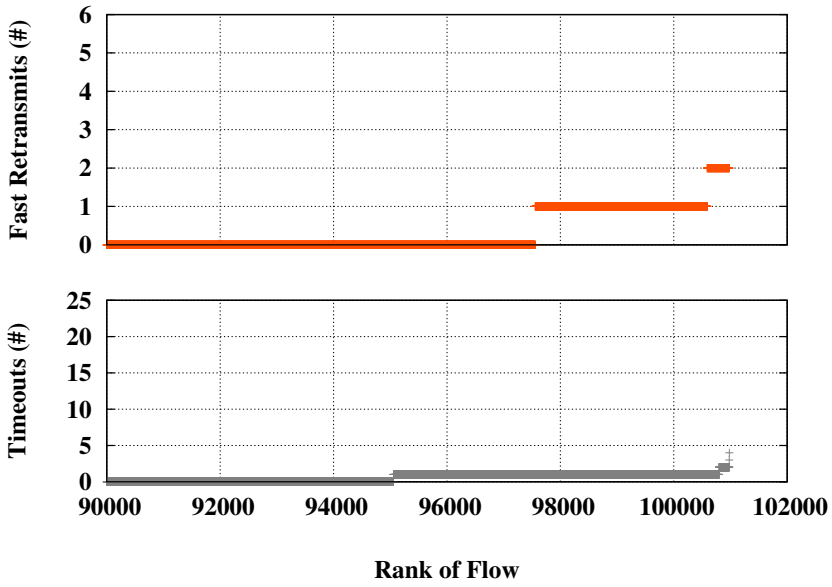
MPTCP$_{Pure}$ achieved an average short flow completion time of 126ms with the standard deviation of 425ms for a total of 99103 completed short flows. The high standard deviation indicates that there are some cases in which MPTCP performs far worse than the average. We repeated the above simulation but this time with the MMPTCP protocol. The results are promising: MMPTCP achieved the average flow completion time of 116ms with the standard deviation of 101ms for a total of completed short flows of 100980. This implies that MMPTCP short flows maintain their ACK clock better than MPTCP with eight subflows when they experience loss events.

We also extracted and plotted the flow completion times, total fast retransmissions and timeouts of each individual short flow. Figure 5.12 shows the results achieved for timeouts and fast retransmits. As is evident from Figure 5.12(a), MPTCP suffers from excessive timeouts. For example, some short flows experienced 25 timeouts during their data deliveries, which is very high for 20 seconds simulation. Figure 5.12(b) shows that MMPTCP performed far better than MPTCP with eight subflows: it decreased the maximum timeouts and fast retransmissions from 25 to 4 and 6 to 2 respectively and a majority of flows experienced fewer than two timeouts.

Figure 5.13 shows the results of short flow completion times. As clearly shown



(a) MPTCP with eight subflows

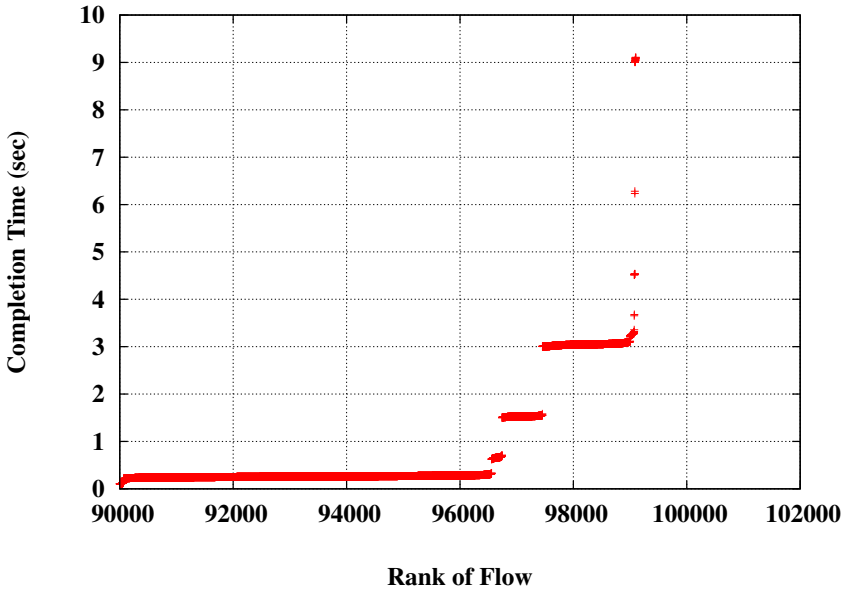


(b) MMPTCP

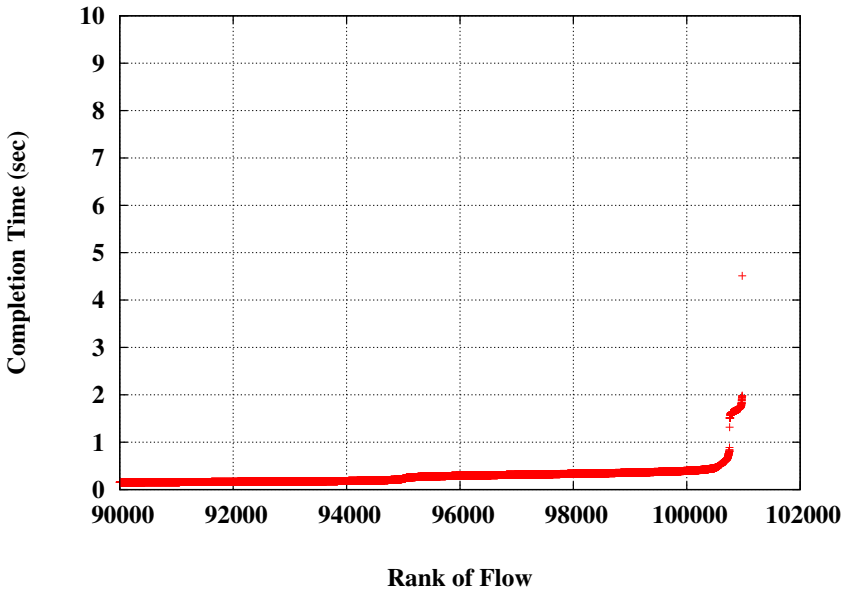
Figure 5.12: Timeouts and fast retransmissions (MMPTCP against MPTCP$_{Pure}$)

in Figure 5.13(a), a majority of MPTCP short flows completed their data deliveries without experiencing any congestion event. That is, the mean flow completion time of MPTCP (126ms) is not unduly high compared to MMPTCP. However, the high standard deviation of MPTCP (425ms) is certainly related to a fraction of short flows that have completed their data deliveries with a long delay due to excessive timeouts and/or fast retransmissions; the red dots around 3 seconds in Figure 5.13(a) are a good example of such completion times.

In this section, we have shown that MMPTCP decreases average flow completion time and standard deviation of short flows compared to MPTCP with eight subflows. This implies that, unlike MPTCP with eight subflows, MMPTCP does not produce a heavy tail in the flow completion time of short flows while it achieves high overall network utilisation. It allows the MPTCP protocol to be deployed in existing data centres and to be used with all existing applications without any reliance on higher layer information. In other words, an application does not need to know about its flow size in order to make a proper decision about the number of subflows to use. This is particularly important for data centre application designers who prefer not to consider underlying networking protocols when they design their applications. Another important implication here is that MMPTCP is incrementally deployable in existing data centres as it could coexist with legacy TCP.



(a) MPTCP with eight subflows



(b) MMPTCP

Figure 5.13: Short flow completion times (MMPTCP against MPTCP$_{Pure}$)

5.5 Comparing MMPTCP to MPTCP$_{SFTCP}$

One of the MMPTCP design goals is to allow MPTCP to run all network flows without the application layer dictating the number of subflows. In other words, MMPTCP attempts to run MPTCP for any type of flows when applications are not aware of MPTCP. In the previous section, we have shown that MMPTCP achieves this goal.

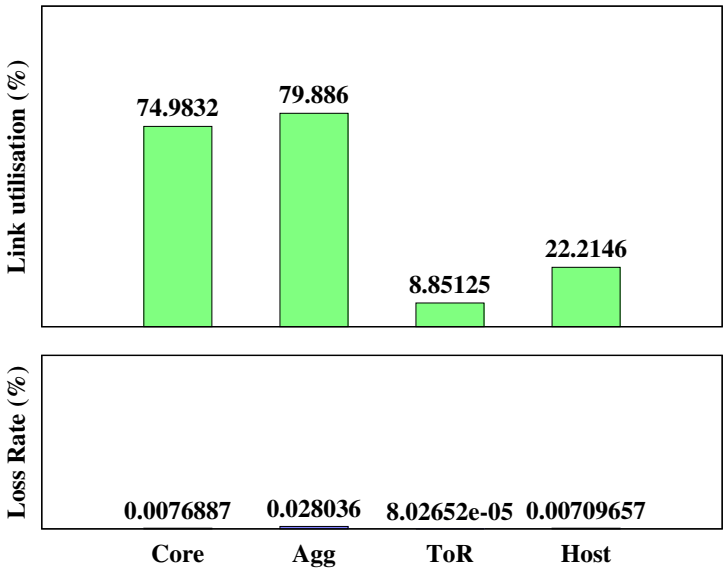
A striking question here is: what if applications are aware of MPTCP and their flow sizes, so that they run short flows via a single subflow (i.e. single-path TCP) and long flows via eight subflows?

To answer this question we designed a Sim$_{\text{Mix}}$ simulation, where short flows run on top of TCP and long flows on top of MPTCP with eight subflows (we refer to this Sim$_{\text{Mix}}$ as MPTCP$_{SFTCP}$). We conducted this simulation with the same configuration as the simulation presented in Section 5.4. In short, a FatTree$_{512}$ topology with an oversubscription ratio of 4:1, the Permutation traffic matrix and $\lambda = 256$. The result is depicted in Table 5.2. Short flows of MMPTCP achieve a higher overall flow completion time than MPTCP$_{SFTCP}$, but the average network core utilisation is almost the same in both simulations.

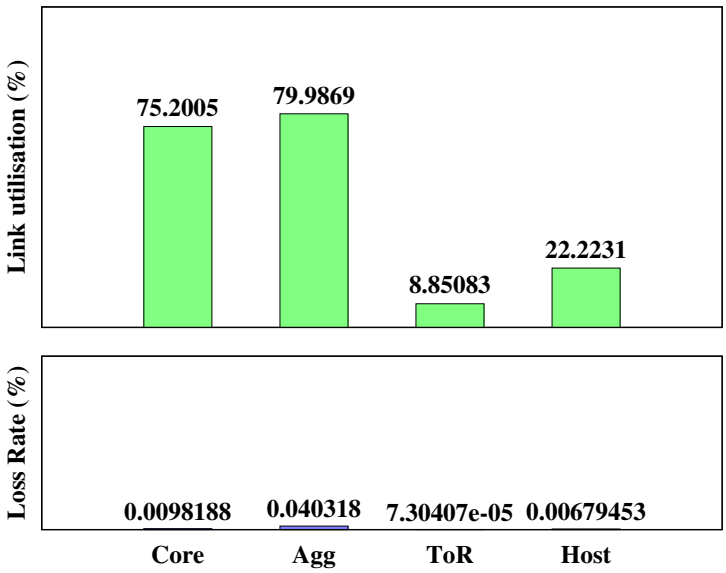
Sim$_{\text{Mix}}$ Simulation Name	Short Flow Finish Time (mean/stdev)	Core Layer Utilisation (mean)
MMPTCP	116± 101	74.9%
MPTCP$_{SFTCP}$	89.2 ± 108.9	75.2%

Table 5.2: MMPTCP compared to MMPTCP$_{SFTCP}$

To explore why MMPTCP achieves a higher average short flow completion time than MPTCP$_{SFTCP}$, we extracted the overall link utilisation and mean loss rate for each layer of the FatTree topology. Figure 5.14 depicts the results. *MMPTCP decreases the average loss rate significantly compared to MPTCP$_{SFTCP}$ at the core, aggregation and access layers (i.e. layers with multipaths).* This is completely as expected, as MMPTCP delivers the packets of its short flows via all parallel paths to their destinations, so that the chance of contention between short and long flows decreases significantly. In addition, the overall link utilisations are almost the same in both simulations. This is also as expected, as both simulations handle long flows by MPTCP (MMPTCP switches to MPTCP after a switching threshold of 100KB is reached).



(a) MMPTCP



(b) MPTCP$_{SFTCP}$

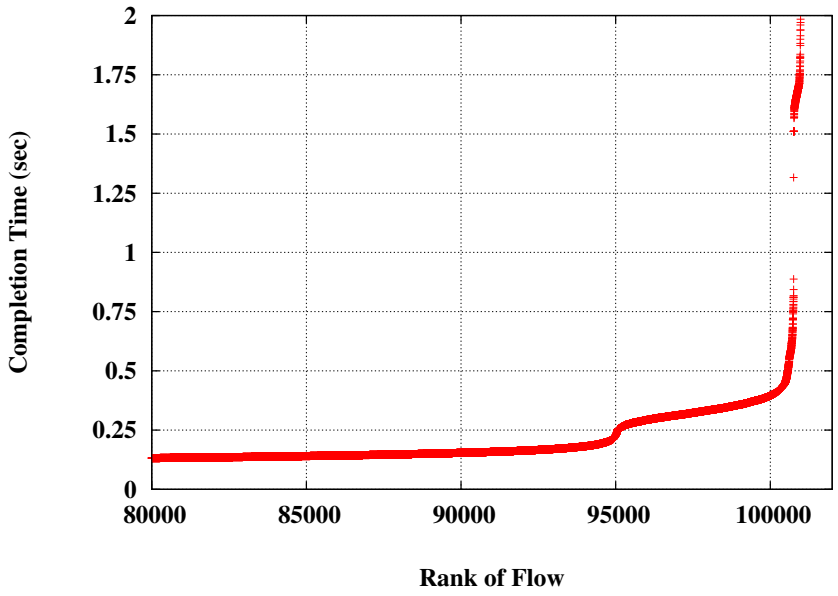
Figure 5.14: Overall link utilisation and loss rate in different layers of the network topology. MMPTCP decreases the average loss rate at core, aggregation and access layers of the FatTree network.

The important question here is: if MMPTCP decreases the average loss rates, especially at the core and aggregation layers, why it does increase the average short flow completion time compared to MPTCP$_{SFTCP}$?

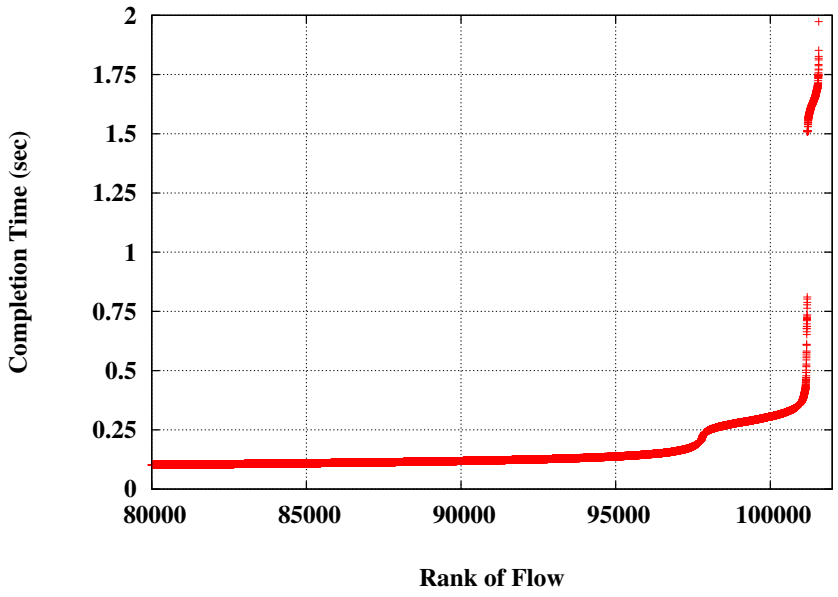
To answer this question, we extracted and plotted the flow completion times, fast retransmissions and timeouts in each individual short flow. Figure 5.15 and 5.16 depict the results. It can be observed by comparing Figure 5.15(a) and 5.15(b) that MMPTCP has a higher flow completion times for a small fraction of short flows (e.g. those between 95K and 100K) than MPTCP$_{\text{SFTCP}}$. Comparing Figure 5.16(a) and 5.16(b) also reveals that short flows of MMPTCP (only a small fraction) experience more timeouts than MPTCP$_{\text{SFTCP}}$. It may be suggested that MMPTCP achieves the higher overall short flow completion time because of the large value of the duplicate ACK threshold (*dupthresh*).

Increasing or adjusting the *dupthresh* value is a tricky task as the TCP NewReno sender could lose its ACK clock, especially when the value of the *dupthresh* is larger than the congestion window. Thus, if any packet gets dropped in such scenarios, TCP needs to wait for a retransmission timer to be triggered. For example, in the above simulations 85% of network flows traverse the network core. This means that a majority of flows set their *dupthresh* value to 19. If any segment gets dropped at the first five RTTs, either at the beginning of data transmission or after any timeout event, then the corresponding subflow should wait until its retransmit timer is triggered.

We believe therefore that the large *dupthresh* value is the main reason why short MMPTCP flows experience a slightly higher timeout compared to TCP ones. In order to improve the performance of MMPTCP in such scenarios, we propose to use the TCP Limited Transmit mechanism in the PS phase of MMPTCP. The detailed discussion of Limited Transmit is in Section 5.7.

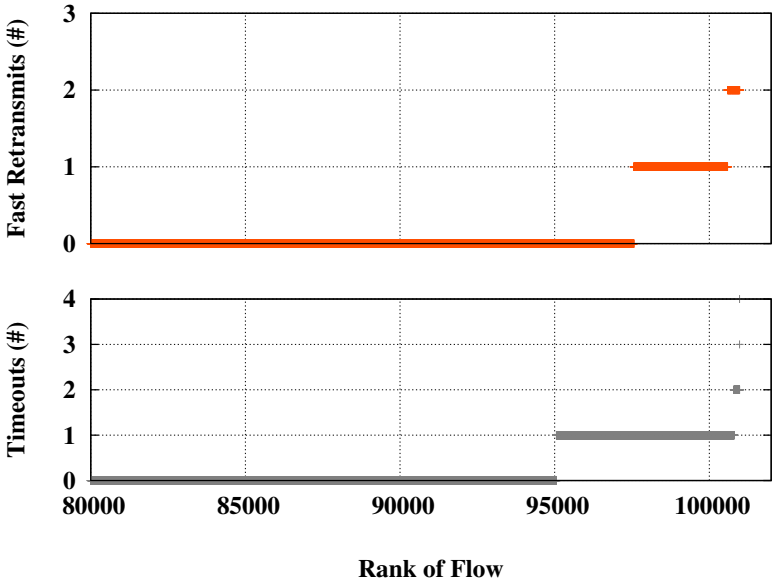


(a) MMPTCP

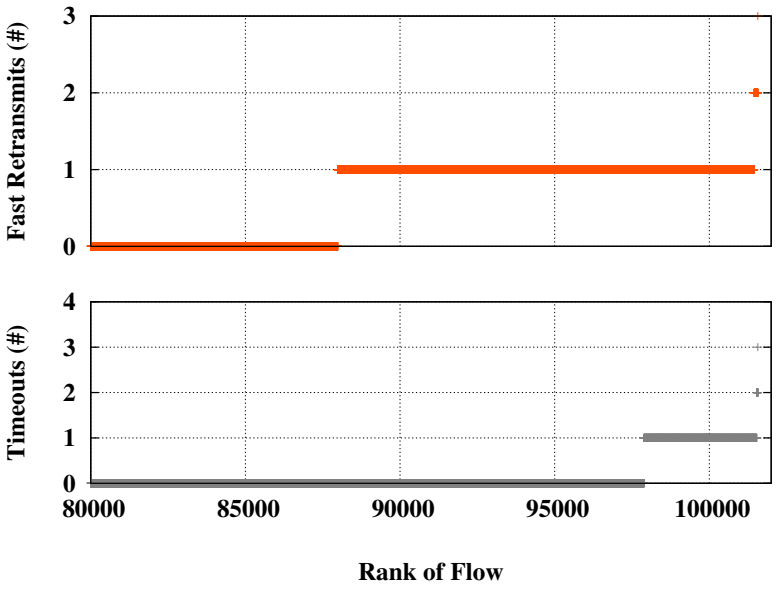


(b) MPTCP$_{SFTCP}$

Figure 5.15: Flow completion times (MMPTCP against MPTCP$_{SFTCP}$)



(a) MMPTCP



(b) MPTCP$_{SFTCP}$

Figure 5.16: Timeouts and fast retransmissions (MMPTCP against MPTCP$_{SFTCP}$)

5.6 Comparing MMPTCP to TCP_{Pure} and PS_{Pure}

This section compares and analyses the performance of TCP, PS, MPTCP and MMPTCP. In order to evaluate TCP and PS protocol, we set up two Sim_{Mix} simulations, first with TCP and second with PS running both large and short flows. We refer to these simulations as TCP_{Pure} and PS_{Pure} respectively. The simulation setup is the same as ones presented in Section 5.4 and 5.5. The results are depicted in Table 5.3. In order to clarify the comparisons of these simulations, we also included the summary of previous results from MMPTCP, $MPTCP_{SFTCP}$ and $MPTCP_{Pure}$.

Sim_{Mix} Simulation Name	Short Flow Transport Protocol	Long Flow Transport Protocol	Short Flow Finish Time (mean/stdev)	Long Flow Goodput (mean/stdev)	Core Layer Utilisation (mean)	Core Layer Loss Rate (mean)
PS_{Pure}	PS	PS	36.9/±38.2 ms	58.6/±18.2 Mbps	75.1%	0.0001%
TCP_{Pure}	TCP	TCP	64.3/±118.2 ms	38.5/±19.8 Mbps	44.7%	0.0259%
MMPTCP	MMPTCP	MMPTCP	116/±101 ms	61.9/±20.0 Mbps	74.9%	0.0076%
$MPTCP_{SFTCP}$	TCP	MPTCP	89.2/±108.9 ms	61.9/±19.6 Mbps	75.2%	0.0098%
$MPTCP_{Pure}$	MPTCP	MPTCP	126/±425 ms	62.1/±19.7 Mbps	75.5%	0.0077%

Table 5.3: All Sim_{Mix} simulations with $\lambda = 256$

The analyses of the results in Table 5.3 are listed below:

- PS_{Pure} achieves the lowest average short flow completion time and overall core loss rate. PS_{Pure} also achieves a high average goodput for long flows.
- TCP_{Pure} achieves the worst overall core utilisation and highest mean core loss rate. However, it achieves better mean flow completion time than the MMPTCP, $MPTCP_{Pure}$ and $MPTCP_{SFTCP}$ simulations.
- $MPTCP_{Pure}$ achieves the worst flow completion time with the highest standard deviation although it performs well in utilising the network core.
- MMPTCP and $MPTCP_{SFTCP}$ achieve higher overall network core utilisation and lower average short flow completion time than $MPTCP_{Pure}$.

PS_{Pure} performs very well in all the above comparisons since it prevents the creation of any hotspots in the network by randomising all packets via all possible paths.

TCP_{Pure} performs worse in utilising network resources because it only uses a single path throughout its data delivery and it therefore unable to find and shift its traffic to least congested paths. TCP gets trapped in a congested path and damages itself and

other competing flows at bottleneck links along the path. As a result, network resources are not efficiently used. This is also the main reason that TCP_{Pure} achieves a lower overall short flow completion time than $MPTCP_{SFTCP}$ or MMPTCP, since a lot of unused capacity in the network is used by a majority of short TCP flows to complete their data deliveries in a short timeframe. In other words, the inability of large TCP flows to utilise network resources provides headroom for short flows to be completed faster.

After this analysis, one might question what is the advantage of running MPTCP in today's data centres if the PS protocol can achieve a perfect load balancing.
The real question here is why PS protocol did not achieve the highest average goodput for long flows even though it had a very low loss rate in the network core.

To our knowledge PS has not been extensively discussed or analysed in the data centre context. However, a recent study argued that PS can deliver short flows much faster compared to TCP, in a oversubscribed FatTree topology (e.g. 4:1), because it can utilise all network capacity [15]. Another study argued that PS throughput is very sensitive to network congestion because PS only holds one congestion window; when a loss event is detected the rate of data transmission is halved. Unlike MPTCP, PS has no way to shift traffic to the least congested paths [12]. Therefore, it is expected that if PS coexists with other transport protocols, such as single-path TCP, its performance may not be as good as what we have seen so far.

In order to gain a deeper insight into PS performance, we have conducted the above simulations with a short flow arrival rate of 2560 per second, which is generating 10 times more short flows per second than $\lambda = 256$. In this way, it is possible to understand the overall performance of each transport protocol under more traffic, and perhaps under a more bursty traffic pattern. This simulation setup not only explains how the congestion control of each transport protocol operates but also explains how the congestion is actually produced by each transport protocol in the network.

Furthermore, we designed another Sim_{Mix}, referred to as PS_{SFTCP}, which uses the TCP protocol for running short flows and the PS protocol for running long flows. PS_{SFTCP} helps to evaluate the performance of PS when it competes with non-PS flows such as TCP flows. Table 5.4 presents the results achieved in this experiment.

PS_{Pure} performs well in all evaluation metrics, especially its overall loss rate in the network core, which is distinctively low in comparison with other setups.

Sim_{Mix} Simulation Name	Short Flow Transport Protocol	Long Flow Transport Protocol	Short Flow Finish Time (mean/stdev)	Long Flow Goodput (mean/stdev)	Core Layer Utilisation (mean)	Core Layer Loss Rate (mean)
PS_{Pure}	PS	PS	40.5/±44.3 ms	52.9/±16.7 Mbps	76.8%	0.0001%
PS_{SFTCP}	TCP	PS	29.7/±31.1 ms	42.5/±11.3 Mbps	61.9%	0.0014%
TCP_{Pure}	TCP	TCP	66.5/±150 ms	34.2/±18.1 Mbps	48.8%	0.0576%
MMPTCP	MMPTCP	MMPTCP	111/±127 ms	55.9/±18.7 Mbps	76.7%	0.0105%
$MPTCP_{SFTCP}$	TCP	MPTCP	83.9/±148 ms	53.3/±17.4 Mbps	73.6%	0.0258%
$MPTCP_{Pure}$	MPTCP	MPTCP	148/±502 ms	55.0/±18.2 Mbps	75.9%	0.0100%

Table 5.4: All Sim_{Mix} simulations with $\lambda = 2560$

PS_{SFTCP} achieves the lowest flow completion time, which entails the degradation of almost 10Mbps in the overall goodput and 15% less in the overall core utilisation. PS_{SFTCP} also increases the overall loss rate of the core layer by 14 times more than PS_{Pure}. The main reason that PS_{SFTCP} achieves a better overall flow completion time than PS_{Pure} is that long flows in PS_{SFTCP} are more susceptible to random packet drops, and hence they reduce their rates more frequently. When a buffer filled up their packets most likely are in the tail of the queue since long flows randomly spread their packets via all possible paths. The consequence of such rate reductions is to decrease some traffic throughout the network (from all queues). This helps many short flows to complete their data delivery without experiencing any collision and with less queuing delay. In other words, one collision between short and long flows helps several other short flows complete their data deliveries with fewer congestion events. This might be a desirable goal per se, if short flow completion is more important than overall network utilisation.

The above experiment justifies that PS is very sensitive to network congestion, so that when it is used for handling long flows it hurts their connection throughputs, and consequently the overall network utilisation. However, we believe that using PS for long flows could be an excellent approach for a data centre that wants to use PS for short flows as well. It does not, however, seem the best approach if required to integrate with existing data centres and compete with other transport protocols.

TCP obtains the worst results in almost all comparisons except the mean flow completion time. As expected, it has the highest loss rates in the network core of all the setup compared.

$MPTCP_{SFTCP}$ doubles the mean core loss rate and achieves a lower mean goodput for long flows and core utilisation than MMPTCP. $MPTCP_{SFTCP}$ also achieves a lower mean flow completion time with a higher standard deviation compared to MMPTCP.

An observation may suggest that any network flow using PS will enable other flows, i.e. non-PS flows, to perform better. A comparison of PS_{Pure} with PS_{SFTCP} and MMPTCP with $MPTCP_{SFTCP}$ bears out this observation.

5.7 Effects of Hotspot

This section studies the performance of various transport protocols under varied number of hotspot core switches in a FatTree topology. The main goal of this study is to understand how each transport protocol reacts to these hotspots. These hotspots may occur for several reasons in modern data centres, including:

- Contention between traffic flowing from the Internet to data centres.
- Hardware failures or cable faults.
- Uneven load in some servers.

In order to model hotspots in the core layer, we modified the drop-tail queue size of hotspot links from 100 to 10 packets.[10] To select links under the hotspot, we devised two approaches: selecting all the links of some randomly selected core switches; or selecting some links of each core switch randomly or arbitrarily. The former approach was used in this experiment as it allows the monitoring of hotspot areas by simply monitoring each core switch under the hotspot. Figure 5.17 shows an example of a FatTree topology with hotspot links under two core switches. In other words, there are two hotspot core switches that produce 16 hotspot links in this example.

Creating hotspots by reducing queue sizes, rather than link rates, helps us to induce packet drops without increasing the network load, so that the long queuing delays is prevented. A long queueing delay due to rate reduction of a link in the core layer of a FatTree topology may produce packet reordering for all PS-based flows, which is not the aim of this experiment. We aimed to investigate that: what type of flows (short or long flows) is the source of collisions in those limited size queues and how each transport protocol reacts to those bottlenecks.

Simulations in this section were conducted in various number of hotspot core switches, ranging from 20% to 60% of total core switches (we refer to the percentage

[10]To select a size for the drop-tail queue, we examined various queue sizes, ranging from 10 to 50 packets, and it turned out that 10 packets can best show the distinctions between the behaviour of the different transport protocols.

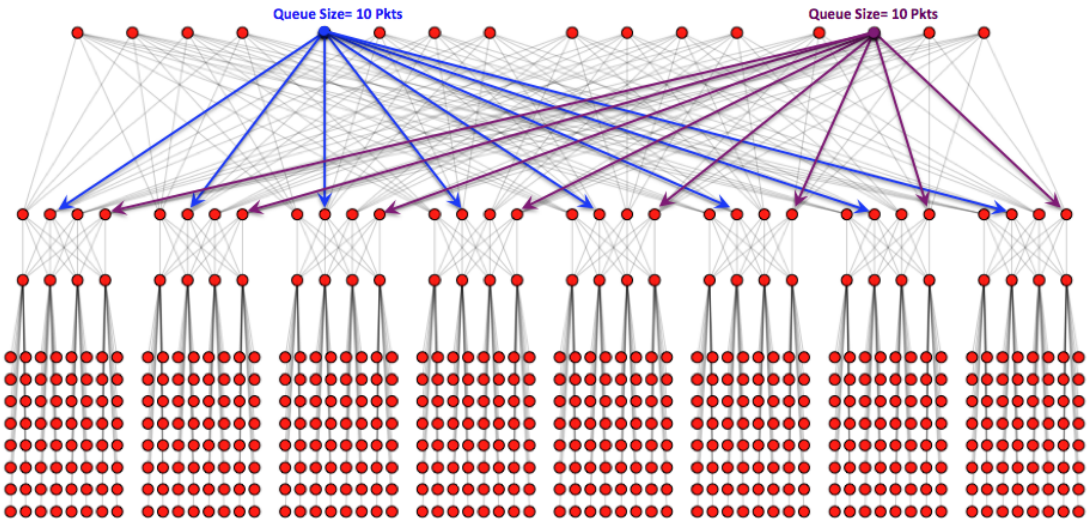


Figure 5.17: All links of two core switches are in hotspots

of cores under hotspot as "hotspot degree"). The network topology used is a FatTree_{512} topology with an oversubscription ratio of 4:1 in links at core and aggregation layers. The traffic matrix used is Permutation and the value of λ is 2560.

It is expected that by increasing the number of hotspot core switches, the overall network utilisation will decrease, the overall short flow completion time will increase and the mean loss rate will increase in all transport protocols. It is interesting to see how each transport protocol follows this trend.

Figure 5.18(c) shows the mean goodput achieved by long flows of each simulation setup under varied hotspot core switches. It is noticeable that PS_{SFTCP} achieved the worst and MMPTCP achieved the best overall goodput compared to other Sim_{Mix} setups. This is another highlighted experiment to show the weakness of PS and the strength of MPTCP in handling congestion. In other words, PS is extremely sensitive and MPTCP is extremely insensitive to network congestion.

Figure 5.18(b) shows that as the hotspot core switches increase, MMPTCP behaves consistently and achieves the highest mean core utilisation at all hotspot degrees. TCP_{Pure} also behaves consistently and with little changes to overall core utilisations. However, it does not behave this way when it comes to the short flow completion time (Figure 5.18(d)). TCP_{Pure} achieves the highest standard deviation compared to other simulation setups, which is not actually surprising (Table 5.5). For example,

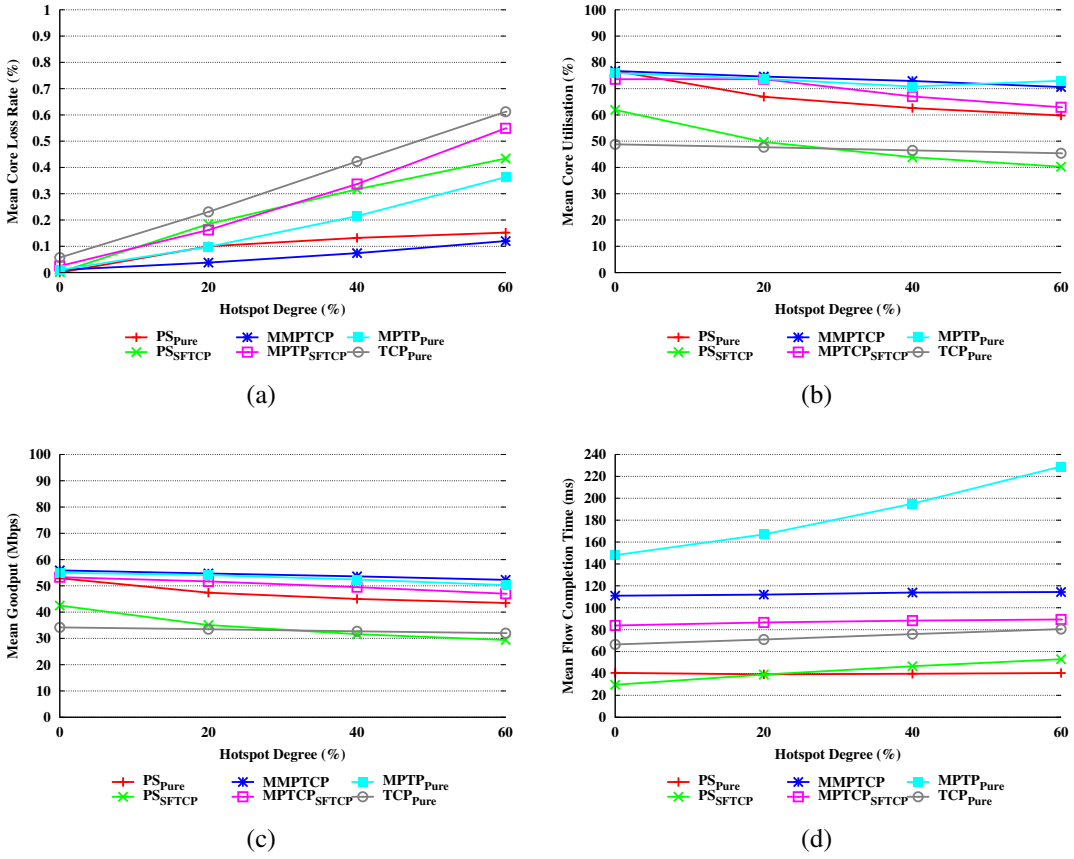


Figure 5.18: All Sim_{Mix} setups under varied hotspot core switches. MMPTCP achieves the lowest mean core loss rate, the highest mean long goodput and the highest mean core utilisation at all hotspot degrees.

at the hotspot degree of 60%[11] short flows of TCP_{Pure} achieved the mean finish time of 80.5ms with a standard deviation of 214ms. This implies that hotspots in TCP_{Pure} mostly exert influence on short TCP flows because large TCP flows have already shown their inability to use network resources efficiently, even without any hotspot link, due to ECMP hash collisions.

Figure 5.18(a) illustrates the mean core loss rate achieved by each simulation setup. MMPTCP achieves the lowest mean loss rate at all hotspot degrees. By increasing the percentage of hotspot cores, the mean loss rate of PS_{SFTCP} and $\text{MPTCP}_{\text{SFTCP}}$ increases significantly as both simulation setups use the TCP protocol for running short flows. The completely opposite result is achieved with MMPTCP and PS_{Pure} because both simulations use the PS protocol for handling short flows.

[11]The 60% of total core switches in this experiment corresponded to 9 core switches, i.e. 72 links from the total of 128 links in this layer.

Simulation Setup	Hotspot Degree (%)	Short Flow Finish Time (mean/stdev)	Long Flow Goodput (mean/stdev)	Core Layer Utilisation (mean)	Core Layer Loss Rate (mean)
$MPTCP_{Pure}$	0%	148/±502 ms	55.0/±18.0 Mbps	75.9%	0.010 %
$MPTCP_{Pure}$	20%	167/±536 ms	54.1/±18.0 Mbps	73.7%	0.098%
$MPTCP_{Pure}$	40%	195/±596 ms	52.4/±17.9 Mbps	70.8%	0.215%
$MPTCP_{Pure}$	60%	229/±662 ms	50.3/±18.9 Mbps	73.0%	0.363%
$MPTCP_{SFTCP}$	0%	83.8/±148 ms	53.3/±17.4 Mbps	73.6%	0.025%
$MPTCP_{SFTCP}$	20%	86.6/±162 ms	51.7/±17.7 Mbps	70.8%	0.162%
$MPTCP_{SFTCP}$	40%	88.3/±177 ms	49.5/±18.2 Mbps	67.0%	0.337%
$MPTCP_{SFTCP}$	60%	89.3/±194 ms	47.0/±18.7 Mbps	62.9%	0.549%
MMPTCP	0%	111.0/±127 ms	55.9/±18.7 Mbps	76.7%	0.010%
MMPTCP	20%	112.0/±130 ms	54.7/±20.0 Mbps	74.6%	0.038%
MMPTCP	40%	113.9/±137 ms	53.6/±21.0 Mbps	72.9%	0.074%
MMPTCP	60%	114.4/±147 ms	52.3/±21.7 Mbps	70.6%	0.120%
TCP_{Pure}	0%	66.5/±150 ms	34.2/±18.1 Mbps	48.8%	0.057%
TCP_{Pure}	20%	71.0/±175 ms	33.5/±18.0 Mbps	47.7%	0.231%
TCP_{Pure}	40%	75.9/±197 ms	32.7/±18.0 Mbps	46.5%	0.423%
TCP_{Pure}	60%	80.5/±214 ms	32.0/±18.0 Mbps	45.4%	0.612%
PS_{Pure}	0%	40.5/±44.3 ms	52.9/±16.7 Mbps	76.8%	0.0001%
PS_{Pure}	20%	39.2/±84.1 ms	47.4/±15.2 Mbps	66.9%	0.100%
PS_{Pure}	40%	39.7/±94.3 ms	45.0/±14.9 Mbps	62.6%	0.132%
PS_{Pure}	60%	40.4/±100 ms	43.5/±15.1 Mbps	59.8%	0.152%
PS_{SFTCP}	0%	29.7/±31.1 ms	42.5/±11.3 Mbps	61.9%	0.0014%
PS_{SFTCP}	20%	39.0/±95.1 ms	35.1/±12.7 Mbps	49.7%	0.185%
PS_{SFTCP}	40%	46.6/±112.7 ms	31.6/±14.4 Mbps	43.9%	0.317%
PS_{SFTCP}	60%	53.0/±125 ms	29.4/±15.7 Mbps	40.3%	0.434%

Table 5.5: The raw results of hotspot simulations

The intuition following this experiment is that the burstiness of data centre traffic, which arises from TCP short flows, is simply smoothed by using MMPTCP. In other words, the TCP protocol for handling short flows not only increases contention but also fails to handle it gracefully. However, MMPTCP not only prevents possible congestion by scattering packets via all possible paths, but also handles it effectively by shifting traffic away from congested areas, after switching to MPTCP. This is the main reason that MMPTCP achieves the lowest loss rate at various hotspot levels compared to other simulation setups. Even PS_{Pure} is not capable of dealing with hotspots effectively since it cannot detect them. These observations are highlighted in Figure 5.18(a).

To explore this discussion, we selected a single execution of the hotspot degree of 40%. We then extracted and plotted the *loss rate* for all the links at the network core. The results are depicted in Figures 5.19, 5.20, 5.21, 5.22 and 5.23 for $MPTCP_{SFTCP}$, MMPTCP, TCP_{Pure}, PS_{Pure} and PS_{SFTCP} simulations respectively. In each figure, the six hotspot cores (Core-1, 3, 5, 7, 8 and 14) are clearly visible since their links have higher loss rates than the other cores. Our goal for showing these figures is to provide

a big picture about the loss rate of links at the network core of each simulation setup. Thus, we do not intend to study these figures in detail.

By comparing MMPTCP (5.20) and $\text{MPTCP}_{\text{SFTCP}}$ (Figure 5.19), we can see how MMPTCP prevents and mitigates possible congestion, especially in the hotspot cores. Loss rates at the hotspot links of MMPTCP are mostly and consistently less than 0.5%, which is far lower than those in $\text{MPTCP}_{\text{SFTCP}}$. Figure 5.19 also shows that long flows in $\text{MPTCP}_{\text{SFTCP}}$ try to get away from hotspot cores by shifting their traffic to other cores. As a result, the loss rate of some random and non-hotspot links slightly increases (e.g. links in Core-2, 4, 6 and 15).

By comparing MMPTCP (Figure 5.20) and PS_{Pure} (Figure 5.22), one can observe that PS_{Pure} can even perform slightly worse than MMPTCP in dealing with hotspots. This is due to the fact that the PS protocol has no mechanism to detect hotspot areas, so the packets from both large and short flows are randomised into those areas. This is the main reason that most links in the hotspot areas of PS_{Pure} have a slightly higher loss rate than MMPTCP.

By looking at TCP_{Pure} (Figure 5.21), we can deduce why TCP achieves a lower flow completion time, a higher standard deviation and overall loss rate than MMPTCP. The reason is that a few links in the non-hotspot areas (e.g. Link-1 in Core-13) experience a high loss rate, but the rest have a loss rate of almost 0%. This implies that a majority of short flows delivers its data very fast, without experiencing any congestion, but some short flows get trapped in congested areas, which are mostly induced by collisions between large TCP flows. This is the main reason that TCP_{Pure} achieves a low flow completion time and a high standard deviation.

Figures 5.22 and 5.23 also bear out the fact that PS_{Pure} is very sensitive to congestion and in turn suggest PS_{SFTCP} achieves the lowest overall goodput in the various degrees of hotspots in Figure 5.18(c). The main reason is that if any packet of a PS flow gets dropped due to these hotspots, then the PS sender halves its transmission rate, degrading the overall network throughput significantly. The loss rate in the hotspot links of PS_{SFTCP} (Figure 5.23) is much higher than in those of PS_{Pure} (Figure 5.22). This is because the short TCP flows in PS_{SFTCP} induce more congestion than the short PS flows in PS_{Pure}. Thus, PS_{SFTCP} achieves the worst overall goodput in all hotspot degrees of all the simulation setups.

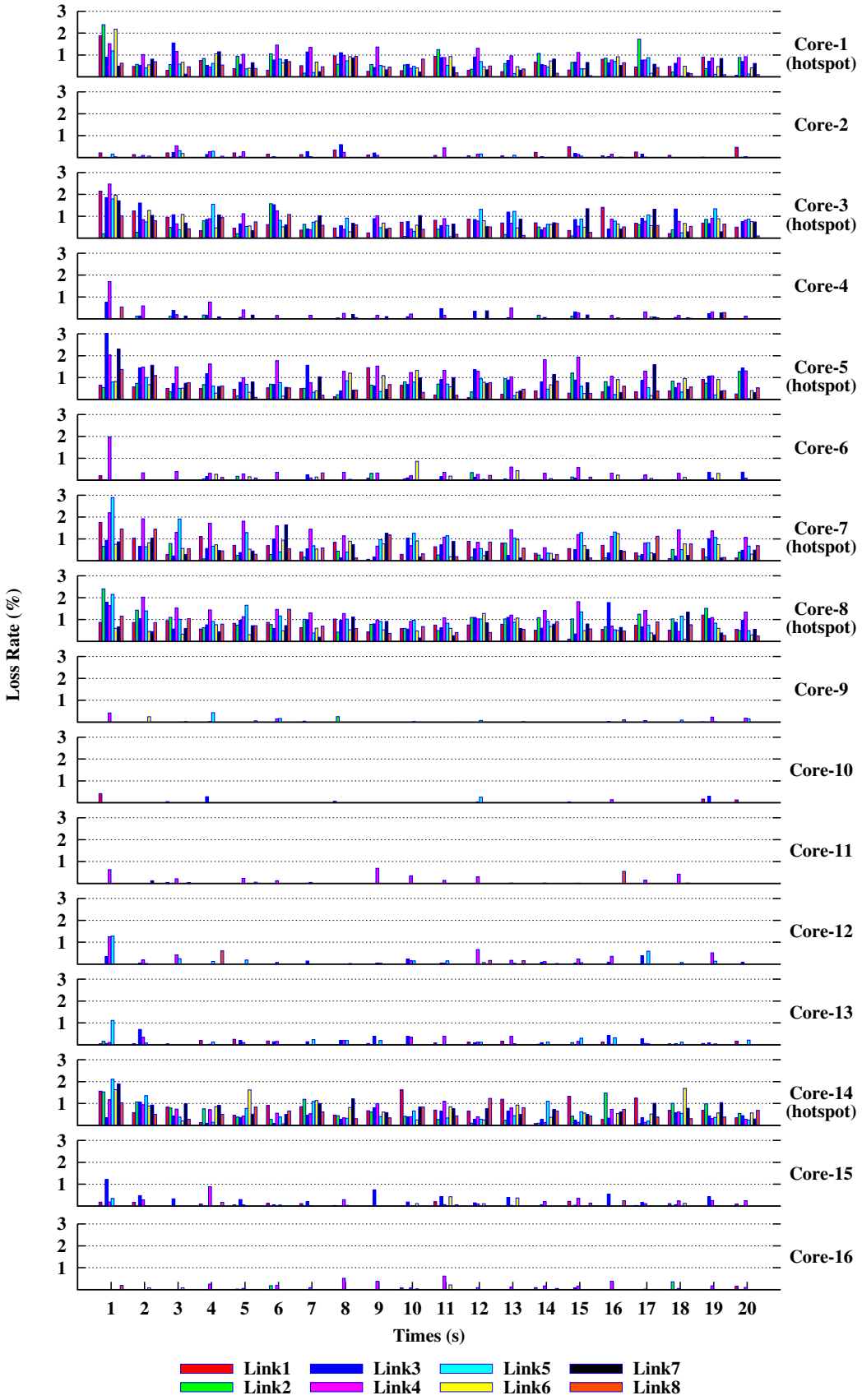


Figure 5.19: $\text{MPTCP}_{\text{SFTCP}}$

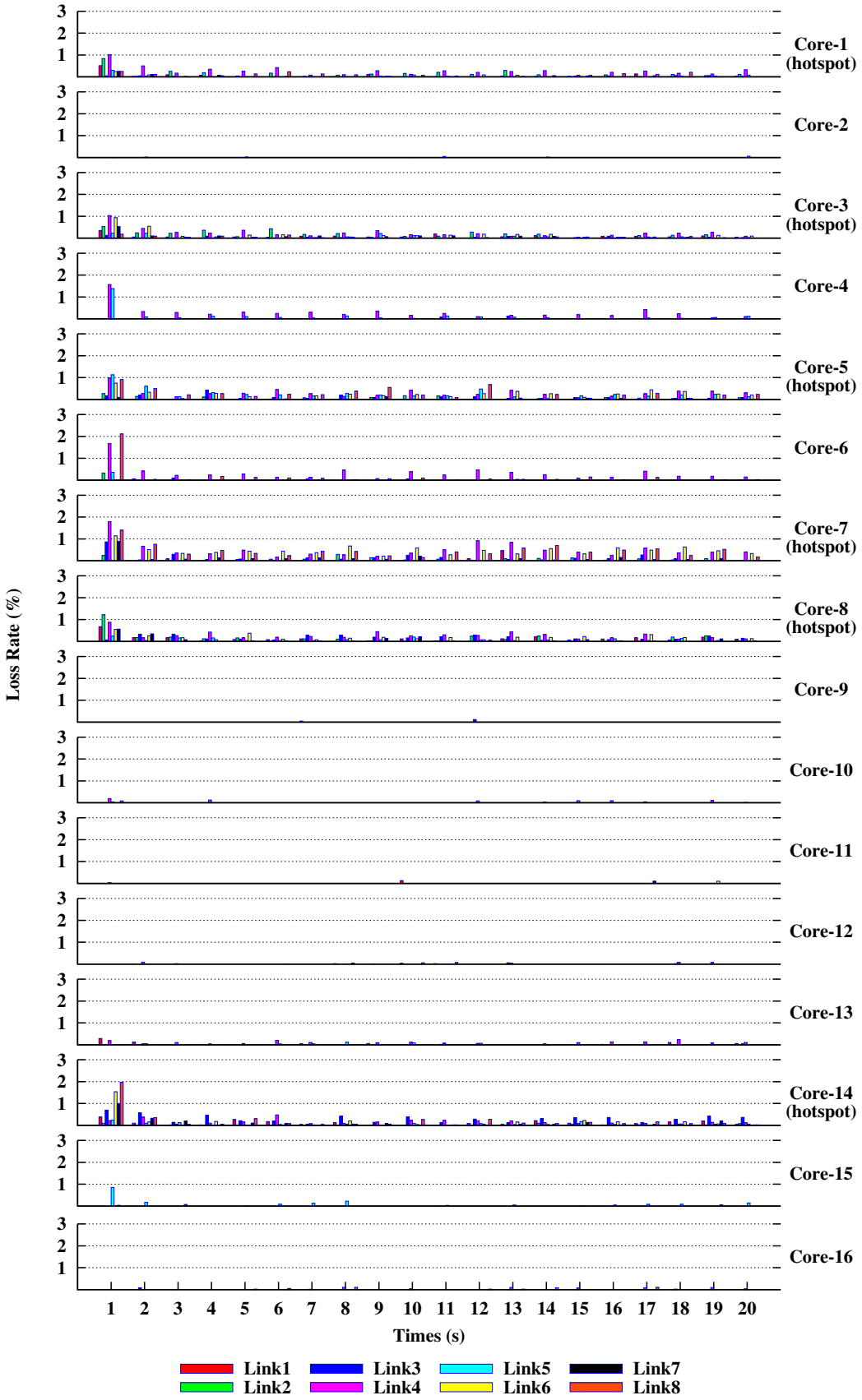


Figure 5.20: MMPTCP

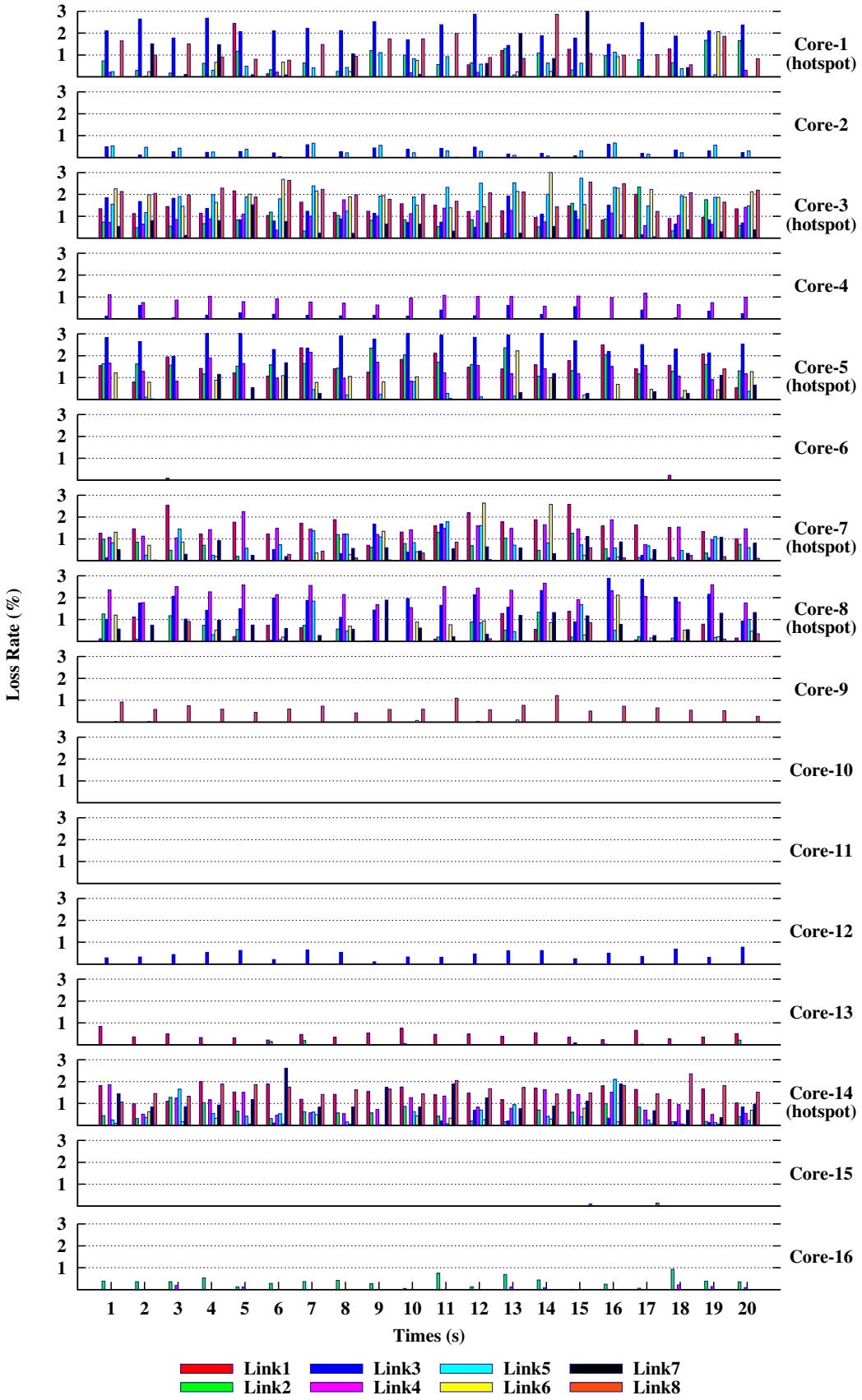


Figure 5.21: TCP_{Pure}

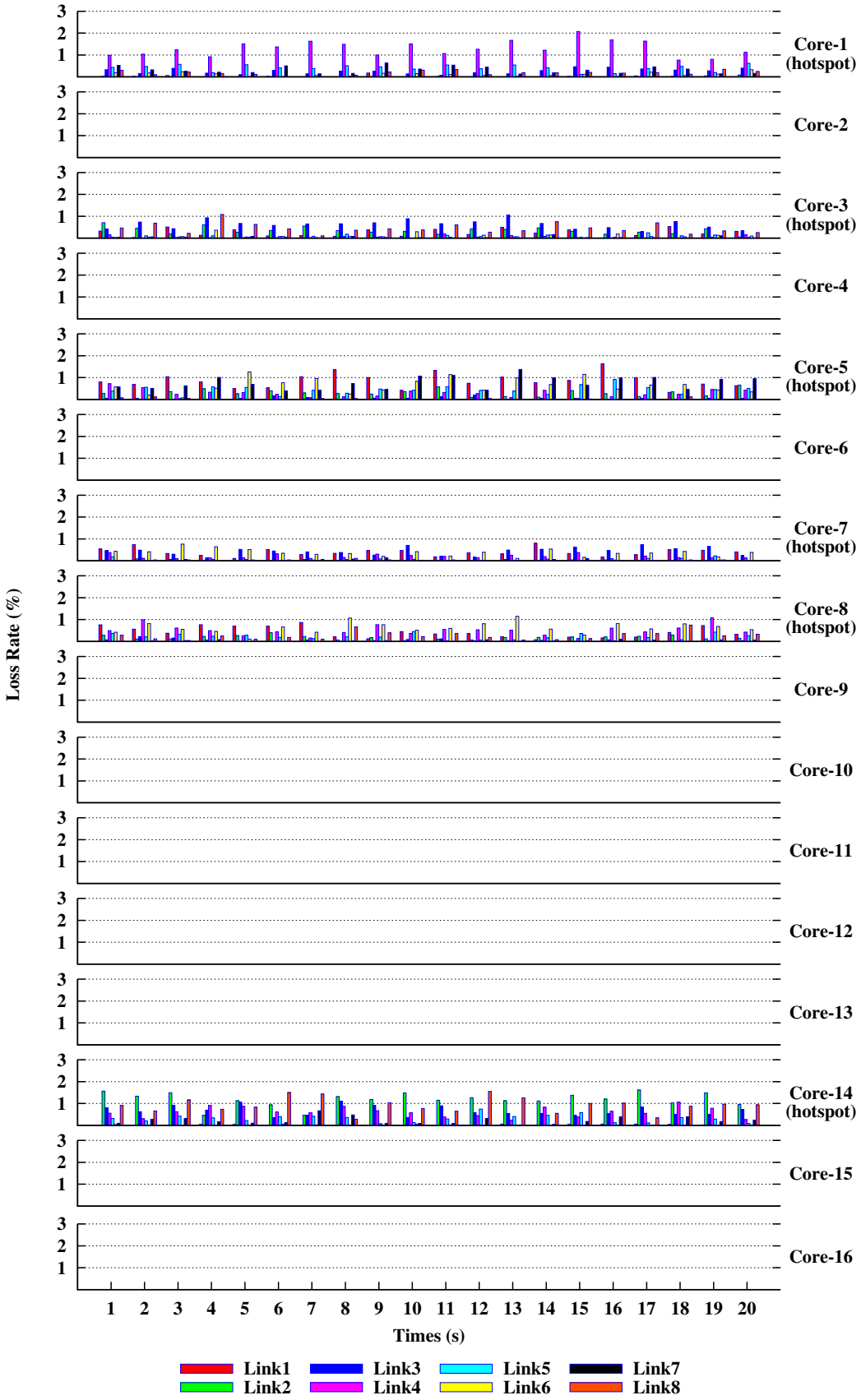


Figure 5.22: PS_{Pure}

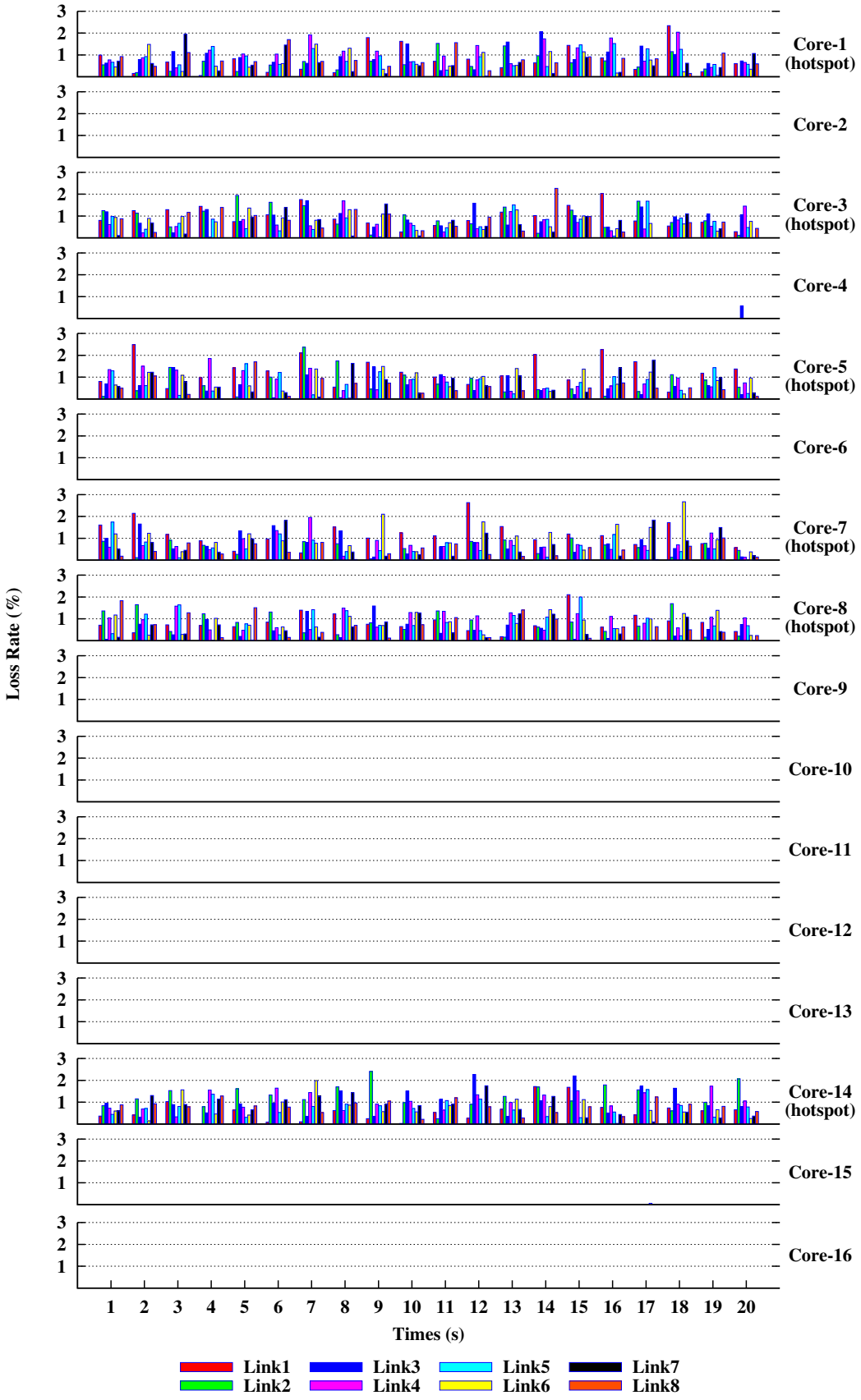


Figure 5.23: PS_{SFTCP}

5.8 Effects of Load

This section investigates the performance of MMPTCP, $\text{MPTCP}_{\text{SFTCP}}$, TCP_{Pure}, PS_{Pure} and PS_{SFTCP} under various network loads in a 2:1 oversubscribed FatTree topology. The objective of this section is to explore how transport protocols in each Sim_{Mix} prevent and react to congestion, especially when the network is highly loaded so that collisions among long flows are most likely to occur everywhere in the network.

To model this experiment, we change the percentage of nodes running long flows, ranging from 10% to 70% of total nodes. This way, as the number of long flows increases, so does network load. It must be noted that changing the percentage of nodes running long flows does not impact on the total number of generated short flows over the course of simulation. For example, if 10% of nodes running long flows then the remainder (90%) are only involved in sending short flows. In other words, if the mean arrival rate is 256 per second, the central scheduler generates 256 short flows per second in average and assigns them randomly to 90% of nodes. Thus, the total number of short flows is almost the same in all simulations conducted in this experiment and its value can be estimated by the Poisson arrival rate. Furthermore, the central scheduler starts scheduling 500ms after a simulation starts.

Figures 5.24, 5.25 and 5.26 depict the results for the flow completion time of short flows, goodput of long flows and loss rate of the network core respectively. As is evident from Figure 5.24, PS_{Pure} achieves the most stable results for the average flow completion time at various traffic levels; by increasing the network load, the mean and standard deviation increase slowly and consistently. TCP_{Pure} follows an unstable trend since the TCP performance is dependent on network condition and traffic pattern. This instability is highlighted at 30% (x-axis). MMPTCP and $\text{MPTCP}_{\text{SFTCP}}$ perform similarly (MMPTCP with less deviation) since both simulation setups handle their long flows by the MPTCP protocol.

Figure 5.25 shows the overall goodput of long flows. It is expected that by increasing the number of long flows, the average goodput will decrease gradually in all simulation setups. This is due to the fact that the network capacity is shared with more long flows. The only simulation setup that achieved slightly different results to the others is related to the TCP_{Pure} since its overall goodput dropped dramatically from 10% to 30% (73.4 to 54.2 Mbps) and then gradually decreased to 37.9 Mbps at 70% load,

the lowest goodput seen in this experiment.

Figure 5.26 shows the mean loss rate at the network core under various network loads. It is expected that when the packet scattering approach is used for handling long flows then the mean loss rate decreases significantly. This is observable by comparing PS_{Pure} or PS_{SFTCP} to other simulation setups. This implies that the packet scattering approach effectively allows a network to handle more traffic with less congestion. In fact this approach increases the fairness among network flows as it equalises the loss rates across all the links at the network core. However, as discussed in Section 5.7, although this approach fails to react to congestion gracefully, it is certainly preventing hotspots. MMPTCP achieves a better mean loss rate at all network loads than TCP_{Pure} and $MPTCP_{SFTCP}$. In fact, TCP_{Pure} performs the worst of all the simulation setups.

We conclude that PS performs well and consistently as the network load increases because large PS flows are able to use all available capacity throughout the network. MMPTCP perform better and more consistently than TCP. However, the MPTCP based protocols are not generally effective in utilising network resources when the network is highly loaded because there is no spared capacity to exploit by their subflows.

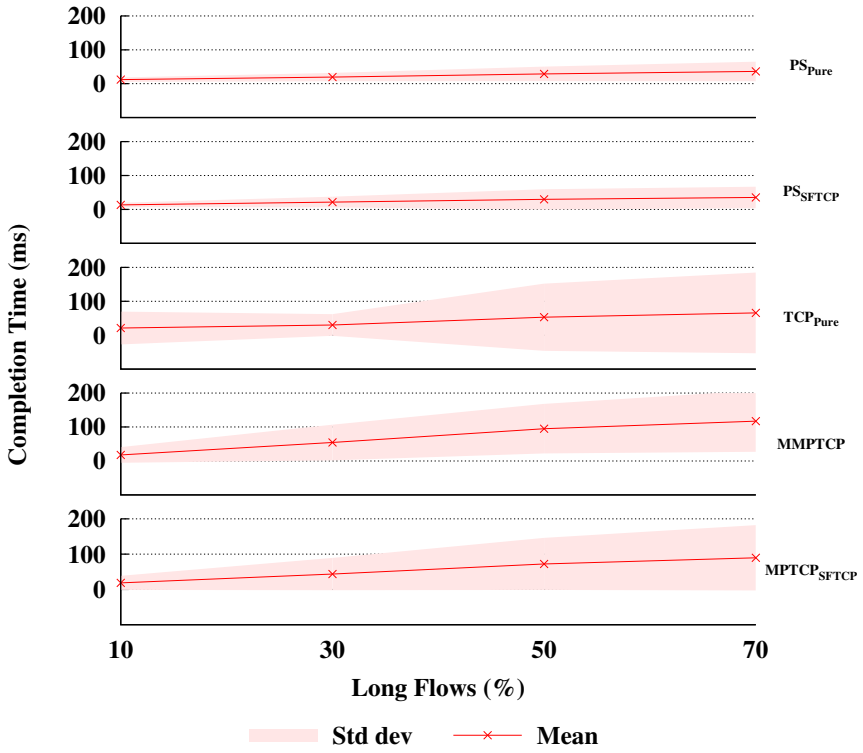


Figure 5.24: Short flow completion time in a 2:1 $FatTree_{256}$ topology under various loads

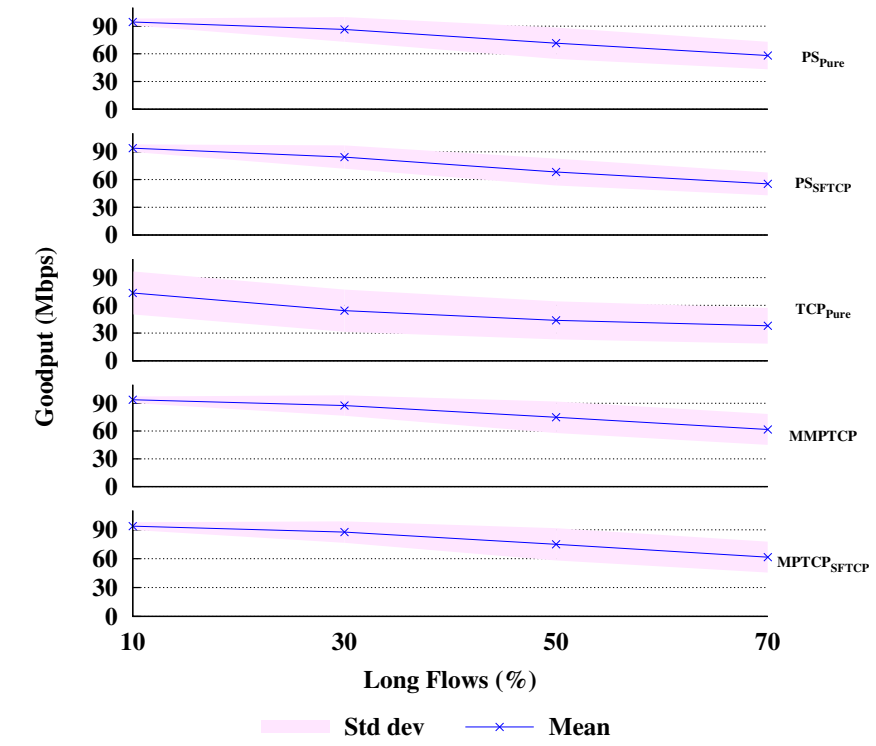


Figure 5.25: Mean goodput of long flows in a 2:1 FatTree$_{256}$ topology under various loads

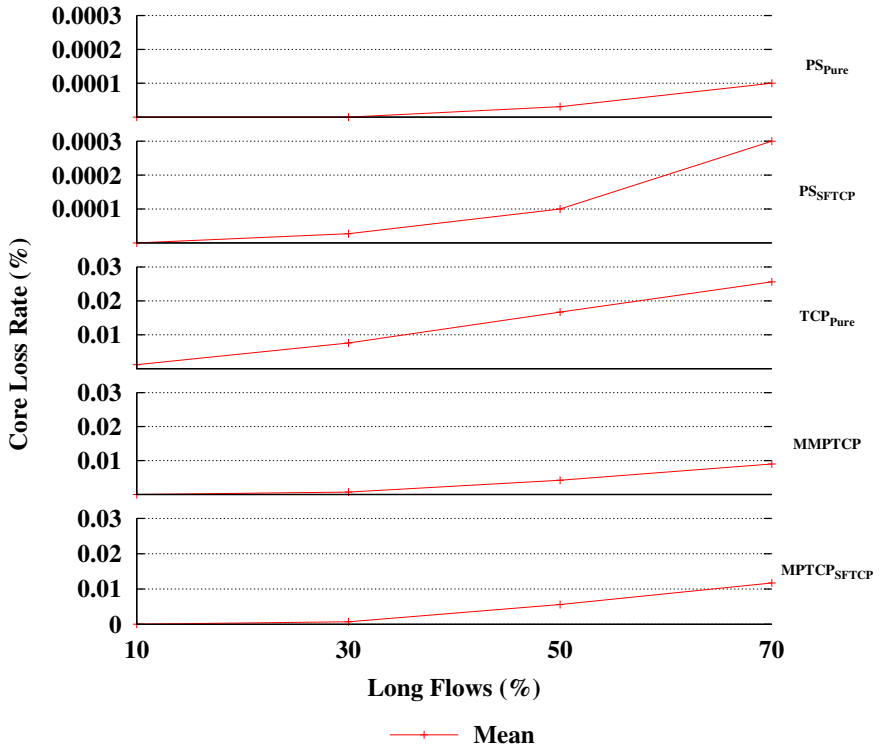


Figure 5.26: Mean core loss rate in a 2:1 FatTree$_{256}$ topology under various loads

5.9 MMPTCP and Multipath Congestion Control

This section explores the performance of MMPTCP protocol when the Fully Coupled (FC), Uncoupled-TCP (UC) or Linked Increases (LI) congestion control algorithm is used during its MPTCP phase. The main goal of this section is to explore how multipath congestion controls affect the completion time of short flows and the overall goodput of long flows. To achieve this goal, we set up a similar simulation to that presented in Section 5.7 (Effects of Hotspots) but ran it with different congestion control algorithms. Table 5.6 depicts the results. As we expected, UC achieves the worst overall short flow completion time and highest standard deviation; almost doubled compared to FC and LI. Additionally, UC achieves the lowest overall goodput and network utilisation compared to other algorithms since it increases the overall host loss rate more than one order of magnitude compared to LI and FC. The main reason for this high loss rate is that eight subflows of MPTCP independently and aggressively compete for the sender's access link capacity, so the majority of loss events happens in that layer. Therefore, the core and aggregation layers with UC have a lower loss rate than FC.

Simulation Name	Short Flow Finish Time (mean/stdev)	Long Flow Goodput (mean/stdev)	Core Layer Utilisation (mean)	Core Loss Rate (mean)
$\text{MMPTCP}_{\text{FC}}$	114/±127 ms	58.9/±18 Mbps	70.8%	0.014%
$\text{MMPTCP}_{\text{UC}}$	179/±270 ms	50/±18.5 Mbps	63.8%	0.010%
$\text{MMPTCP}_{\text{LI}}$	116/±101 ms	61.9/±20 Mbps	74.9%	0.007%

Table 5.6: MMPTCP with Fully Coupled, Uncoupled-TCP and Linked Increases

FC doubles the mean core loss rate and achieves a lower network utilisation than LI. The most likely reason is that FC shifts its traffic to the least congested paths, but those paths may become congested quickly since all short flows also use those paths, so FC has to shift its traffic again. That is, FC may create traffic oscillation between subflows. This oscillation causes long delays in some short flows completing their data delivery. This reasoning is observable from the cumulative distribution function of short flow completion times in Figure 5.27. As discussed earlier in this section, UC has a high standard deviation and hence its flow completion time is heavy-tailed compared to LI and FC. The same is also true for FC compared to LI, but with less severity. The lower average flow completion time of FC compared to LI can be observed in this figure since the majority of short flows of FC complete their flows faster (up to 90 percentile).

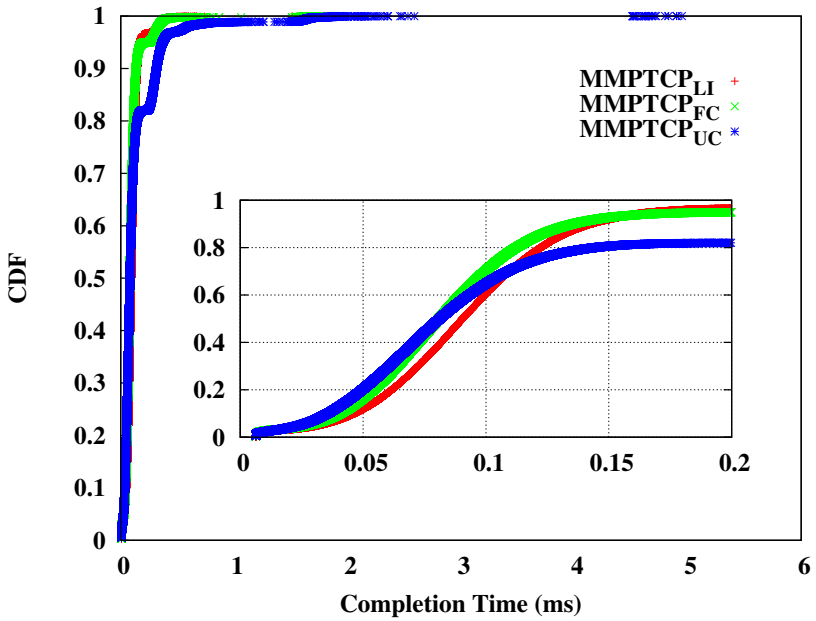


Figure 5.27: Cumulative distribution function of short flow completion times with Fully Coupled, Uncoupled-TCP and Linked Increases. The small plot is a zoom of the big plot. Most short flows of FC achieve a better flow completion time than LI.

It is certainly clear that a multipath congestion control has great impact on overall network utilisation and short flow completion times. The interesting research questions here are as follows:

1. How fast should the MPTCP congestion control react to congestion events?

2. Is TCP congestion control, which is designed to operate on single path, a suitable algorithm to be used in PS phase of MMPTCP protocol? For example, if any packet gets dropped due to a possible transient congestion somewhere in the core layer, why should MMPTCP halve its sending rate since the received signal of congestion implies that neither a particular path in the network nor the entire core layer is congested?

The simplest answer to the second question is to design a new mechanism that allows the MMPTCP sender to infer where a congestion signal comes from and hence react to it accordingly. For example, if a congestion signal is received due to congestion on the access link of a sender/receiver, then the TCP congestion control seems to be a suitable approach. However, if a congestion signal is received from the other layers of the network, the TCP congestion control seems to be an overkill approach.

DCTCP congestion control might be an interesting solution here as it reacts in proportion to congestion. This implies that if a congestion originates from an access layer then a fraction of marked packets will be significant as there is no multipath in this layer. The reaction to congestion thus becomes similar to TCP as the worst case scenario (i.e. halving a sending rate). However, if a congestion signal is random and possibly comes from different core switches during an RTT then a proportion of marked packets will be very low. PS therefore will not reduce its sending rate in such cases.

A possible approach to the first question is to integrate the decrease part of a DCTCP-like congestion control algorithm to the decrease part of MPTCP congestion control algorithm (i.e. the increase part of MPTCP congestion control algorithm remains unchanged). In this way, MPTCP may detect and react to a potential collision at a network link much faster than the Linked Increases congestion control algorithm. Further research is essential to understand the behaviour of MPTCP with such congestion control algorithm in a wide range of network scenarios.

We believe that PS/MMPTCP might perform better than what we have shown so far with the right congestion control algorithm. We plan to conduct further research on improving PS/MMPTCP in future.

5.10 MMPTCP and Limited Transmit

Limited Transmit (LT) is an enhancement to TCP loss recovery and attempts to prevent Retransmission Timeouts (RTOs) in TCP, especially when the congestion window size is very small [85, 34, 72]. LT allows a TCP sender to transmit new segments only upon arrival of the first two duplicate ACKs on a segment, i.e. before the fast retransmission mechanism is triggered.

We modified this algorithm so that a TCP sender allows new segments to be sent before fast retransmission is triggered regardless of the *dupthresh* value. For example, if *dupthresh* is 19 then a TCP sender allows to send 18 new segments before triggering the fast retransmission. In this way, a TCP sender can prevent timeouts when a packet gets dropped and *cwnd* is smaller than *dupthresh*. We have integrated this new algorithm into the PS phase of MMPTCP.

To evaluate the performance of MMPTCP with LT, we designed two new Sim_{Mix}: one with MMPTCP without LT, and the other with LT; we refer to these simulations

as MMPTCP and MMPTCP$_{LT}$ respectively. Both follow the same simulation setup, as follows: a 2:1 oversubscribed FatTree$_{256}$ topology running the Permutation traffic matrix with a short flow arrival rate of 256 per second in average. 53% of nodes (135 nodes) send long flows and the remaining 47% of nodes (121 nodes) send short flows. The first short flow schedules 500ms after simulation starts in order to let long flows become stable. Table 5.7 depicts the results for MMPTCP and MMPTCP$_{LT}$. MMPTCP$_{LT}$ improves mean flow completion time and standard deviation significantly without damaging overall network utilisation.

Simulation Name	Short Flow Finish Time (mean/stdev)	Long Flow Goodput (mean/stdev)	Core Layer Utilisation (mean)	Core Layer Loss Rate (mean)
MMPTCP	98.9/±74.8 ms	72.9/±17.3 ms	72%	0.0053%
MMPTCP_LT	89.1/±67.2 ms	72.9/±18.0 ms	72%	0.0051%

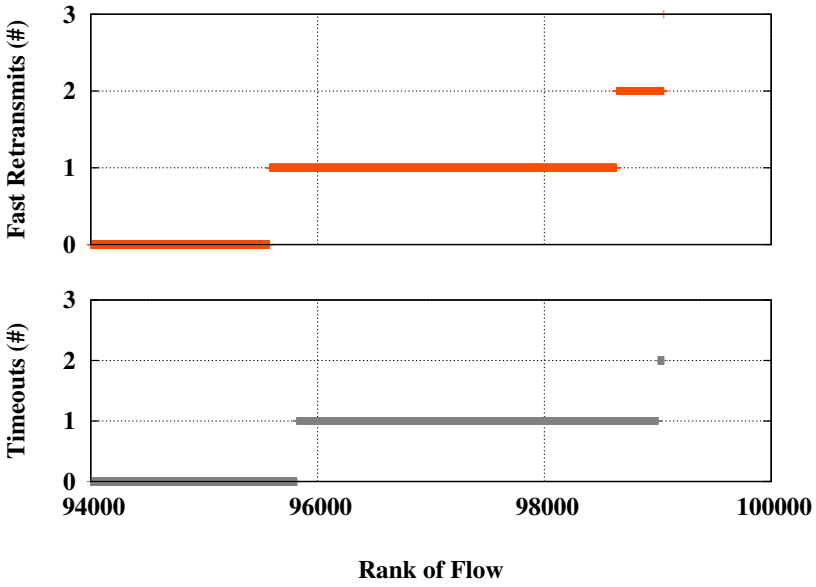
Table 5.7: MMPTCP Compared to MMPTCP$_{LT}$

For both simulations, we extracted and plotted total fast retransmissions and timeouts in each individual short flow along with their flow completion times. Figures 5.28 and 5.29 shows the results. As expected, MMPTCP$_{LT}$ increases the number of fast retransmissions in favour of decreasing the number of timeouts compared to MMPTCP (Figure 5.28). This implies that MMPTCP$_{LT}$ protects short flows from losing their ACK clocks to a great extent when a high *dupthresh* value is used (e.g. 19). MMPTCP$_{LT}$ therefore decreases the flow completion time of a majority of short flows compared to MMPTCP. For example, Figure 5.29(b) has a higher concentration of short flow completion times before 0.4 seconds than Figure 5.29(a).

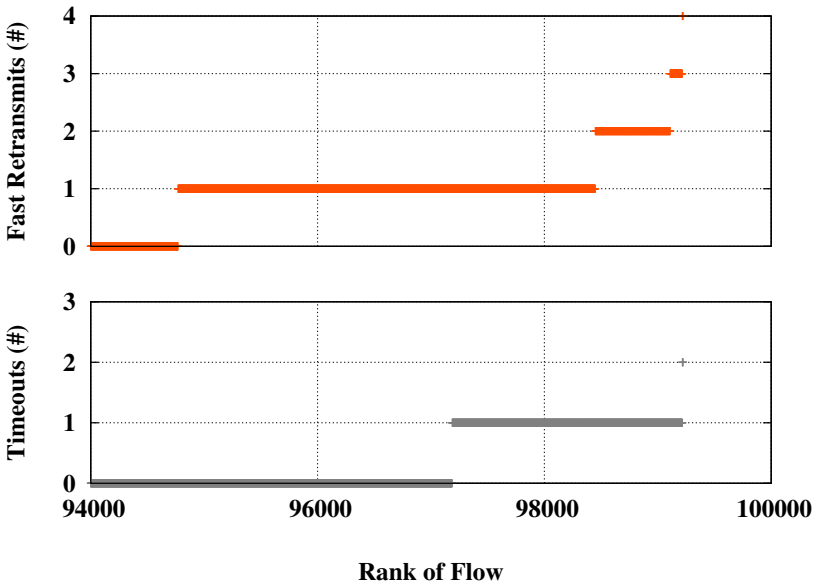
It is argued that LT is an essential mechanism for *preventing* TCP from losing its ACK clock, especially when *dupthresh* is adjusted automatically and is large (e.g. 19) [72]. However, LT becomes more aggressive as the *dupthresh* value increases. This may be less critical for MMPTCP because its short flows use all possible paths in data delivery. However, even with MMPTCP, if there is a hotspot in the access layer then this aggressiveness becomes important and may hurt other competing flows in such a case. Further research is required in order to understand how LT should be used when *dupthresh* is very large.

Note should be taken that neither the *dupthresh* adjustment nor the LT mechanism is helpful in *detecting* and *mitigating* spurious retransmissions caused by packet

reordering events. We therefore believe that the performance of MMPTCP can be significantly improved by replacing TCP NewReno with DSACK TCP in the PS phase since a TCP sender is able to detect and recover from spurious retransmissions gracefully. We plan further research to explore this in future.

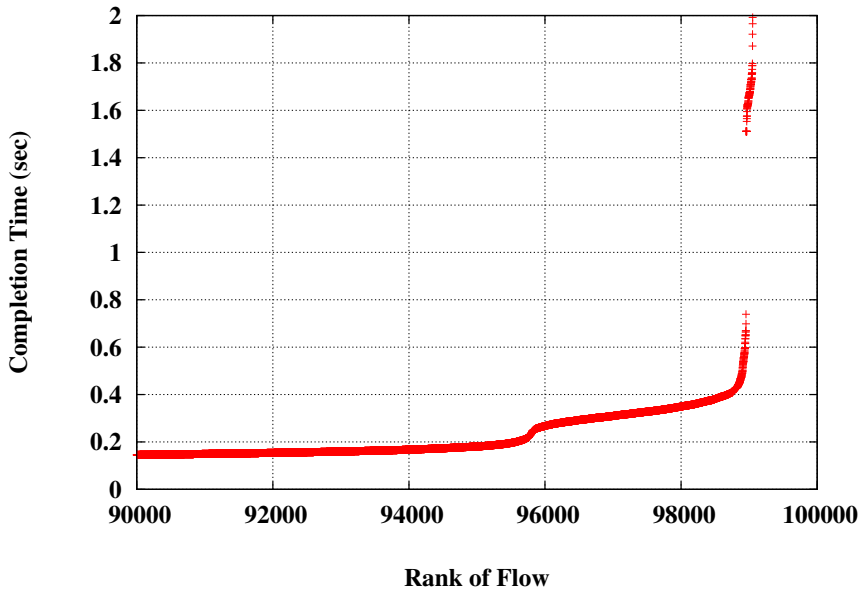


(a) MMPTCP

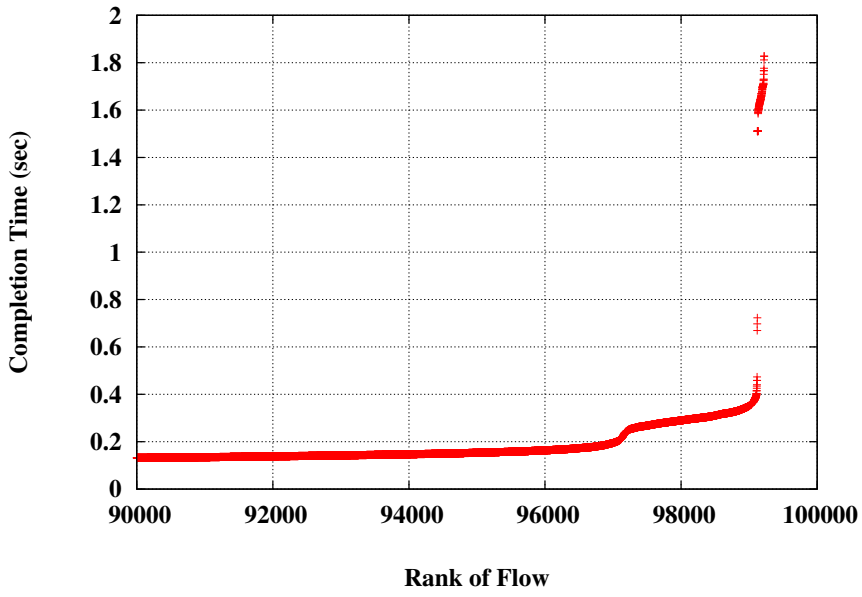


(b) MMPTCP$_{LT}$

Figure 5.28: Timeouts and fast retransmissions (MMPTCP against MMPTCP$_{LT}$)



(a) MMPTCP



(b) MMPTCP$_{LT}$

Figure 5.29: Flow completion times (MMPTCP against MMPTCP$_{LT}$)

5.11 MMPTCP Switching Mechanism

In this section, we investigate the effects of the MMPTCP switching point with two scenarios: (1) the completion time of short flows when the size of short flows is lower or higher than a switching point; and (2) the goodput of long flows.

For the first scenario, we conducted a range of simulations with varying short flow sizes over various switching points. The simulation setup is as follows: a 2:1 oversubscribed FatTree_{256} topology running the Permutation traffic matrix with a short flow arrival rate of 256 per second in average. 53% of nodes send long flows and the remaining 47% of nodes send short flows. The first short flow is scheduled 500ms after simulation starts.

Table 5.8 shows the results. It is clear that changing the switching threshold does not exert any negative effect on the completion time of short flows since the results for a flow size (e.g. 70KBs) with different switching thresholds are very consistent. This is a very important outcome because it is very likely that some short flows in a data centre have larger sizes than a switching threshold.

To begin with the second scenario, let us assume MMPTCP is running only long flows. Switching to MPTCP is thus expected to occur whenever the switching threshold is reached. At the time of switching, though, MMPTCP might have a very large window of data in flight. Unlike TCP, the congestion window here does not relate to the congestion state of a specific path. After switching, therefore, each new established subflow ought to probe the network in order to prevent congestion collapse. In other words, new established MPTCP subflows are not allowed to burst their traffic into the network after switching because they assume that MMPTCP was sending aggressively before switching.

The important question is how the MMPTCP switching mechanism might affect the goodput of long flows.

To answer this question, we designed two Sim_{Long} simulations, one with MMPTCP and other with MPTCP with eight subflows. We refer to them as MMPTCP and MPTCP simulations respectively. The setup of both simulations is as follows: a FatTree_{128} topology running the Stride traffic matrix in which each node sends a single long flow to a single destination; as discussed in Section 5.2.2, with Stride, all

Short Flow Size (KB)	Switching Threshold (KB)	Short Flow Finish Time (mean/stdev)	Long Flow Goodput (mean/stdev)	Core Layer Utilization (mean)	Core Layer Loss Rate (mean)
50	100	86.2/±66.6 ms	73.0/±17.0 Mbps	71.8 %	0.0052 %
50	300	85.2/±66.4 ms	72.8/±18.6 Mbps	71.6 %	0.0042 %
50	500	86.3/±71.5 ms	73.0/±18.2 Mbps	71.7 %	0.0040 %
50	1000	86.2/±71.2 ms	72.9/±18.1 Mbps	71.7 %	0.0035 %
50	10000	82.2/±73.2 ms	72.3/±17.6 Mbps	71.6 %	0.0029 %
70	100	98.9/±74.8 ms	72.9/±17.3 Mbps	72.9 %	0.0053 %
70	300	98.4/±79.0 ms	72.7/±18.5 Mbps	71.7 %	0.0043 %
70	500	97.7/±74.8 ms	72.8/±18.2 Mbps	71.9 %	0.0041 %
70	1000	98.5/±75.1 ms	72.6/±18.1 Mbps	71.7 %	0.0037 %
70	10000	94.3/±77.5 ms	72.2/±17.5 Mbps	71.7 %	0.0034 %
200	100	151.9/±109.7 ms	70.7/±17.1 Mbps	72.6 %	0.0061 %
200	300	150.6/±107.9 ms	71.4/±18.3 Mbps	72.3 %	0.0051 %
200	500	150.5/±109.0 ms	71.6/±17.9 Mbps	72.5 %	0.0049 %
200	1000	152.0/±111.0 ms	71.5/±18.0 Mbps	72.4 %	0.0045 %
200	10000	144.8/±108.2 ms	71.0/±17.3 Mbps	72.5 %	0.0039 %
400	100	228.0/±152.8 ms	69.7/±16.7 Mbps	73.4 %	0.0071 %
400	300	227.4/±147.5 ms	69.5/±17.9 Mbps	73.2 %	0.0063 %
400	500	228.7/±152.4 ms	69.6/±17.5 Mbps	73.4 %	0.0058 %
400	1000	228.5/±152.1 ms	69.5/±17.5 Mbps	73.3 %	0.0056 %
400	10000	221.7/±152.8 ms	69.1/±17.3 Mbps	73.4 %	0.0048 %
600	100	324.8/±198.0 ms	67.9/±16.2 Mbps	74.4 %	0.0080 %
600	300	321.8/±194.7 ms	67.6/±17.4 Mbps	74.2 %	0.0068 %
600	500	312.5/±196.7 ms	67.8/±17.0 Mbps	74.4 %	0.0064 %
600	1000	325.3/±195.8 ms	67.7/±17.0 Mbps	74.3 %	0.0062 %
600	10000	315.2/±198.4 ms	67.3/±16.8 Mbps	74.4 %	0.0061 %

Table 5.8: MMPTCP Switching Threshold Sensitivity

flows traverse via the network core. The network traffic is also expected to be distributed evenly in such a topology. We repeated this simulation with various simulation durations, ranging from 1 to 20 seconds.

Table 5.9 shows the results and they are within our expectations. MMPTCP achieves almost identical results to MPTCP. We conclude that the switching mechanism of MMPTCP protocol has insignificant or no negative effect on the goodput of long flows since newly established subflows after switching can fully utilise the access link capacity in a few RTTs.

Simulation Duration	Transport Protocol	Long Flow Goodput (mean/stdev)	Core Layer Utilisation (mean)	Core Layer Loss Rate (mean)	Jain's Fairness Index
1 Second	MPTCP	67.2/14 Mbps	73.8 %	0.002 %	0.956
	MMPTCP	66.9/14 Mbps	73.5 %	0.002 %	0.954
2 Seconds	MPTCP	75.5/9.8 Mbps	82.0 %	0.003 %	0.983
	MMPTCP	75.2/9.8 Mbps	81.6 %	0.003 %	0.983
5 Seconds	MPTCP	81.7/7.5 Mbps	88.1 %	0.003 %	0.991
	MMPTCP	81.7/7.4 Mbps	88.0 %	0.003 %	0.991
20 Seconds	MPTCP	86.1/5.0 Mbps	92.6 %	0.003 %	0.996
	MMPTCP	86.1/4.9 Mbps	92.6 %	0.003 %	0.996

Table 5.9: MMPTCP compared to MPTCP via a Sim_{Long} in a FatTree_{128} topology running a Stride matrix of long flows

5.12 Effects of Incast

The purpose of this section is to investigate the performance of MMPTCP, TCP, MPTCP and PS under various incast scenarios with both short and long flows. In Section 3.7, we discussed the TCP incast problem. In short, the incast problem might happen when a link, typically at the access layer of the network, needs to handle a large number of synchronised flows. When those flows are short-lived, a transient congestion might occur that lead to a long flow completion time in some of those flows. When those flows are long-lived, a persistent congestion might become apparent that lead to the collapse of throughput in several of those flows.

To model the incast for short flows, we designed a simulation in a 1:1 oversubscribed FatTree_{128} topology running various number of parallel short flows, ranging from 20 to 100 flows, to a single random destination. Parallel short flows are scheduled every 500ms over the course of simulation (20s) to repeat this condition for several times. Each network link has the speed of 100Mbps with delay of 20μs and drop-tail queue of 100 packets.

Figure 5.10 shows the results, including median (50th percentile) and upper quartile (75th percentile). MPTCP with eight subflows achieved the lowest performance and has the lowest number of completed flows compared to other transport protocols. The main reason for the latter outcome is that a majority of short MPTCP flows could not establish a connection to their destinations since their SYN packets got dropped. Those short flows who succeeded in establishing their connections suffer from excessive timeouts, so that they cannot complete their data delivery before simulation stop

time (40 seconds after the simulation duration); this condition become worse as the number of parallel short flows increases.

$\text{MMPTCP}_{\text{LT}}$ performs almost closely to TCP. However, unlike $\text{MMPTCP}_{\text{LT}}$, MMPTCP performs slightly worse than TCP because it cannot prevent timeouts when packets get dropped and the *cwnd* value is smaller than the *dupthresh* value.[12] It is clear that non of these transport protocols perform well when the network bottlenecks are at the access layer of the network.

			Short Flow Finish Time		
No. of Parallel Flows	Transport Protocol	No. of Completed Short Flows	Mean/Stdev	Median	Upper Quartile
20	MMPTCP	16000	261.3/81.0 ms	277.2 ms	288.5 ms
	$\text{MMPTCP}_{\text{LT}}$	16000	180.0/38.9 ms	99.5 ms	108.7 ms
	TCP	16000	118.4/59.6 ms	99.9 ms	109.4 ms
	MPTCP	7884	6.5/5.5 s	6.22 s	6.26 s
40	MMPTCP	32000	416.9/285 ms	452.7 ms	486.8 ms
	$\text{MMPTCP}_{\text{LT}}$	32000	186.0/68.3 ms	165.2 ms	249.9 ms
	TCP	32000	203.4/76.4 ms	169.4 ms	273.4 ms
	MPTCP	1975	7.6/6.8 s	6.2 s	10.5 s
100	MMPTCP	47486	1.5/3.4 s	767.4 ms	935.3 ms
	$\text{MMPTCP}_{\text{LT}}$	64825	2.2/4.3 s	746.6 ms	1.6 s
	TCP	61110	2.1/4.5 s	726.0 ms	1.0 s
	MPTCP	263	6.9/7.8 s	4.5 s	10.5 s

Table 5.10: Incast scenarios with short flows

To model the incast for long flows, we repeated the above simulations but this time only with long flows. Parallel long flows are scheduled only at the beginning of the simulation and compete for a shared bottleneck link at the access layer over the course of the simulation. The results are depicted in Figure 5.11. As we expected, PS and TCP are achieved almost similar performance. The intuition is that when the network bottlenecks are at the access layer, which does not provide any multipaths, a multipath data delivery can not improve the overall network performance compared to a single-path data delivery.

MMPTCP achieved a lower mean goodput and standard deviation compared to MPTCP. This result does not imply that MMPTCP can perform better than MPTCP in incast scenarios since both transport protocols use MPTCP for handling long flows. The

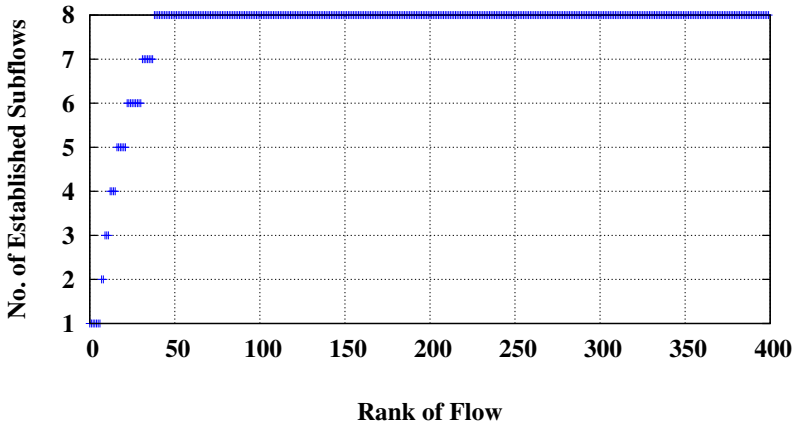
[12] In this experiment, MMPTCP behaves identically to PS since all flows are short-lived and have smaller flow size than the MMPTCP switching threshold.

No. of Parallel Flows	Transport Protocol	No. of Completed Long Flows	Long Flow Goodput (mean/stdev)	ToR Layer Loss Rate (mean)	Jain's Fairness Index
20	MMPTCP	400	4.62/4.84 Mbps	0.009 %	0.435
	TCP	400	4.78/1.81 Mbps	0.012 %	0.859
	MPTCP	400	4.64/7.58 Mbps	0.007 %	0.233
	PS	400	4.75/1.77 Mbps	0.005 %	0.849
40	MMPTCP	800	2.27/2.12 Mbps	0.013 %	0.458
	TCP	800	2.39/1.06 Mbps	0.017 %	0.791
	MPTCP	800	2.34/9.99 Mbps	0.003 %	0.049
	PS	800	2.37/0.89 Mbps	0.010 %	0.807
100	MMPTCP	2000	0.88/0.81 Mbps	0.024 %	0.358
	TCP	2000	0.95/0.50 Mbps	0.025 %	0.490
	MPTCP	2000	0.94/8.23 Mbps	0.001 %	0.127
	PS	2000	0.94/0.40 Mbps	0.019 %	0.516

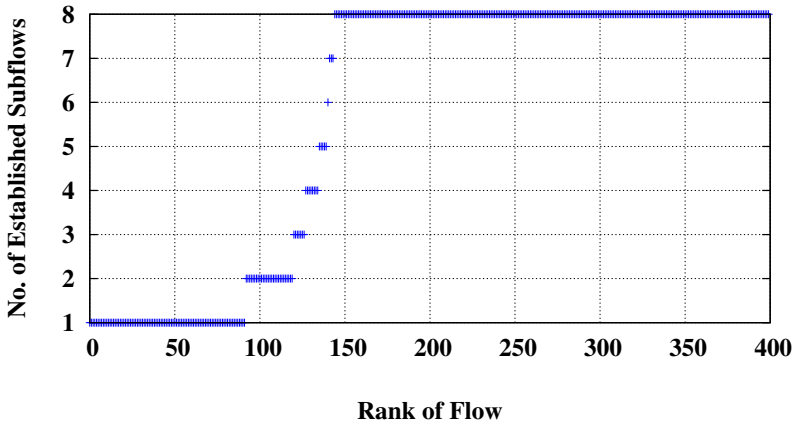
Table 5.11: The incast scenarios with long flows

main reason is that MMPTCP switches to MPTCP shortly after the simulation is started. In this condition, several SYN packets of the newly initiated subflows of a flow may be dropped because the access link of the receiver is already congested. Furthermore, the switching to MPTCP is not performed until at least one newly initiated subflow is established. As a result, some of the MMPTCP flows may continue their data deliveries with less number of subflows than they initiate; the less number of subflows implies the less number of timeouts.

To demonstrate above explanation, we extracted and plotted the number of established subflows for each individual flow of MMPTCP and MPTCP from the simulations with 20 parallel long flows (Table 5.11). Figure 5.30 shows the results. MPTCP has less than $\sim$50 flows with eight subflows, including 16 flows with less than 5 subflows. However, MMPTCP has $\sim$150 flows with less than eight subflows, including $\sim$90 flows with only one subflow.



(a) MPTCP



(b) MMPTCP

Figure 5.30: Number of established subflows per each individual long flow.

5.13 Summary

In this chapter, we compared and analysed the performance of MMPTCP against existing solutions in a wide range of network scenarios in a FatTree topology. Our evaluation showed that MMPTCP outperformed MPTCP with eight subflows running for both short and long flows. MMPTCP also outperformed all existing transport protocols when there were hotspots in the network. MMPTCP consistently achieved a low overall loss rate and high overall network utilisation in all the experiments presented in this chapter. MMPTCP realised all these high performances with TCP NewReno and without a mechanism to detect and mitigate spurious retransmission due to packet reordering, such as DSACK.

Our examination has shown that our solution for adjusting *dupthresh* prevents spurious retransmission to a great extent. We have also found that increasing *dupthresh* to

a high value increases timeouts in some short flows with small *cwnd*. In turn, these timeouts can be mitigated by the Limited Transmit mechanism to a great extent.

We examined the MMPTCP switching mechanism by comparing MMPTCP with MPTCP, when both protocols run only long flows. The results indicated that the MMPTCP switching mechanism does not degrade the goodput of long flows.
Furthermore, we inspected the effect of the MMPTCP switching threshold on the flow completion time of short flows when they have a flow size higher or lower than a switching threshold. The results indicated that the switching threshold does not exert any negative effect on the flow completion time of short flows in such conditions.

We studied the performance of MMPTCP, TCP, PS and MPTCP under various incast scenarios both with short flows and with long flows. The results confirmed that none of the above transport protocols can perform well under an incast scenario.

We investigated the feasibility of simulation via the ns-3 simulator as the network size and link rate increase. The results demonstrated that it is impractical to model a large-scale data centre network via the packet-level event-driven simulator.

Chapter 6

Conclusions

In this thesis, we conducted an in-depth study of MPTCP for short flows in a FatTree topology. We observed that MPTCP is not the right solution for handling short flows. A fraction of short flows complete their flows with a long delay because they incur excessive timeouts. We proposed MMPTCP as a means to address this problem. Our evaluation showed that MMPTCP is practical and decreases flow completion time for short flows while retaining high throughput for long flows over MPTCP with a fixed number of subflows. We also observed that MMPTCP not only reacted to congestion gracefully but also prevented it in a great extent, thereby significantly decreasing the overall loss rate of all links in the network.

One of MMPTCP's challenges is to prevent, detect and react to spurious retransmission due to packet ordering, during its initial phase of delivery. In this thesis, we proposed a solution to *prevent* spurious retransmissions. Our solution is based on the FatTree IP addressing scheme as it allows us to locate end-hosts according to their IP address. That is, the *dupthresh* is adjusted according to the destination IP address of a flow at connection establishment. Our investigation showed that adjusting *dupthresh* in this way significantly prevents spurious retransmission.

MMPTCP includes a switching threshold for switching to MPTCP. Currently, this threshold is triggered when a certain amount of data (e.g. 1MB) has been transmitted. We observed that this switching mechanism does not exert any negative effect on the connection throughput of long flows since the opening of eight subflows after switching can fully utilise access link capacity in a few RTTs.

We conclude that MMPTCP is rapidly deployable in existing data centres as it coexists with other transport protocols and operates based on existing data centre tech-

nologies such as ECMP. It can handle all network flows without high-level information from application layers (e.g. flow sizes and deadlines). It decreases the bursty nature of data centres by leveraging parallel paths for delivering short flows.

6.1 Future Directions

During our evaluation, we realised that employing TCP congestion control during the initial phase of MMPTCP seems an overkill approach. Our intuition arose from the fact that when a congestion signal originated from a random link at the network core, it seems overkill to react to that congestion by halving the sending rate. However, if a congestion signal comes from a bottleneck link at the access layer, then the reaction of TCP congestion control is correct. The research question here is how can we distinguish these two signals and react appropriately. Our hypothesis is that reacting to congestion proportionally to the extent of congestion will allow detection of these two signals. We thus believe that employing the DCTCP-link congestion control could be an interesting solution for distinguishing these two signals. If a congestion signal comes from random links at the network core then the proportion of congestion signals, during one RTT, is very low so DCTCP does not reduce its sending rate. However, if it is from a bottleneck link at the access layer, DCTCP reacts similarly to TCP. Further investigation is required to determine best practices, parameter adjustments, and so on.

MMPTCP is capable of utilising multi-homed network topologies that make no sense with TCP. Unlike MPTCP, MMPTCP is capable of delivering all network flows via all available network interface devices. This nice feature potentially allows the TCP Incast problem to be addressed by adding more interface devices to end-hosts. We plan to conduct further research on the performance of MMPTCP over multi-homed topologies, such as Dual-Homed FatTree (DHFT) [12].

In this thesis, we evaluated MMPTCP with TCP NewReno, which is a widely deployed TCP version. However, TCP NewReno is not an ideal solution when packet reordering is the norm. There are three aspects to dealing with out-of-order packets: preventing, detecting and mitigating spurious retransmissions. We explored a solution for preventing packet reordering by increasing *dupthresh* in order to postpone the triggering of the fast retransmission mechanism. However, any solutions that attempt to increase the value of *dupthresh* may increase timeouts when a packet gets dropped while

the congestion window is smaller than *dupthresh*. To address this, we activated TCP limited transmit during the initial phase of MMPTCP. We believe increasing *dupthresh* and coupling it with limited transmit is the right approach for preventing spurious retransmissions in modern data centres, but further research is required into the degree to which limited transmit should react to duplicate ACKs. We also plan to investigate how DSACK will improve the performance of MMPTCP, as it can help to detect and mitigate spurious retransmissions.

Advance QoS features have become increasingly available in data centre switches [13]. Our hypothesise is that if packets of the initial phase of MMPTCP are marked high priority and routed through a different queues, then MMPTCP effectively and seamlessly helps latency-sensitive short flows to meet their deadline. The packet of short flows is thereby routed from a different queue to the long flows so that the chance of random packet drop significantly decreases, especially at the network core.

We used the ns-3 simulator for modelling our data centres since it can closely model a real network, e.g. each ns-3 node can include a networking stack similar to Linux Kernel. We however observed that it is almost impractical to model a very large data centre network with ns-3 due to the long simulation completion time. We thus initially ran all of our simulations with the link rate of 100Mbps in a small-scaled FatTree topology to achieve a flexibility in running a large number of network scenarios in a reasonable time-scale (e.g. a few experiments per day). We then ran our final results (a small number of experiments) with the 1Gbps link rate in a small-scaled FatTree topology and observed that MMPTCP performed slightly better with the 1Gbps link rate compared to 100Mbps (it achieved a small performance differences with other transport protocols in question). A solution to model a very large network is to use the flow-level simulator such as the *htsim* simulator [86]. The flow-level simulator does not model a network with its real network properties (e.g. there is no networking stack as it exists in real operating systems, SYN packet, hashed-based ECMP as it exists in network switches, or even the RTOs are not triggered by a timer in the *htsim* simulator). We however believe that it is now the right time to examine MMPTCP in data centres with realistic network sizes and link rates via a flow-level simulator (*htsim*) and compare its results with the results achieved from ns-3.

Bibliography

[1] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *ACM SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, December 2008.

[2] C. Kim, M. Caesar, and J. Rexford. Floodless in Seattle: A Scalable Ethernet Architecture for Large Enterprises. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 3–14, 2008.

[3] C. Raiciu, C. Pluntke, S. Barre, A. Greenhalgh, D. Wischik, and M. Handley. Data Center Networking with Multipath TCP. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 10:1–10:6, 2010.

[4] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. *Communications of the ACM*, 54(3):95–104, March 2011.

[5] M. Al-Fares, A. Loukissas, and A Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, 2008.

[6] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 63–74, 2009.

[7] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: A Scalable and Fault-tolerant Network Structure for Data Centers. In *Proceedings of the ACM*

SIGCOMM 2008 Conference on Data Communication, SIGCOMM '08, pages 75–86, 2008.

[8] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, 2010.

[9] C Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. RFC 2992, November 2000.

[10] L.G. Valiant. A scheme for fast parallel communication. *SIAM journal on computing*, 11(2):350–361, 1982.

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[12] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, and M. Wischik, D.and Handley. Improving Datacenter Performance and Robustness with Multipath TCP. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 266–277, 2011.

[13] D. Zats, T. Das, P. Mohan, and R.H. Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. *ACM SIGCOMM Comput. Commun. Rev.*, 42(4):139–150, August 2012.

[14] A. Ford, C. Raiciu, and M. Handley. Architectural guidelines for multipath TCP development. RFC 6182, 2011.

[15] A. Dixit, P. Prakash, Y. C. Hu, and R. R. Kompella. On the Impact of Packet Spraying in Data Center Networks. In *INFOCOM, 2013 Proceedings IEEE*, pages 2130–2138, April 2013.

[16] T. Hoff. Latency is Everywhere and it Costs You Sales - How to Crush it, July 2009. Available from: `http://highscalability.com/`

latency-everywhere-and-it-costs-you-sales-how-crush-it [accessed 1 May 2015].

[17] M. Alizadeh, A. Greenberg, D.A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 63–74, 2010.

[18] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 50–61, 2011.

[19] M. Kheirkhah, I. Wakeman, and G. Parisis. A Multipath Transport Protocol for Data Centers. In *Proceedings of the 2016 IEEE International Conference on Computer Communications (INFOCOM)*, 2016.

[20] M. Kheirkhah, I. Wakeman, and G. Parisis. Short vs. Long Flows: A Battle That Both Can Win. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 349–350, New York, NY, USA, 2015. ACM.

[21] M. Kheirkhah, I. Wakeman, and G. Parisis. Multipath TCP model in ns-3. The Workshop on ns-3 (WNS3), 2014. Available from: http://www.uclmail.net/users/m.kheirkhah/mptcp-wns3-2014.pdf [accessed 1 May 2015].

[22] V. Jacobson. Congestion Avoidance and Control. In *Symposium Proceedings on Communications Architectures and Protocols*, pages 314–329, 1988.

[23] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782, April 2004.

[24] J. Mahdavi, M. Mathis, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, October 1996.

[25] S Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An Extension to the Selective Acknowledgement (SACK) Option for TCP. RFC 2883, July 2000.

[26] K. Rojviboonchai and H. Aida. An evaluation of multi-path transmission control protocol (M/TCP) with robust acknowledgement schemes. *IEICE transactions on communications*, 87(9):2699–2707, 2004.

[27] H.-Y. Hsieh and R. Sivakumar. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*, MobiCom '02, pages 83–94, 2002.

[28] H. Han and S. Shakkottai and C. V. Hollot and R. Srikant and D. Towsley. Multi-path TCP: A Joint Congestion Control and Routing Scheme to Exploit Path Diversity in the Internet. *IEEE/ACM Transactions on Networking (TON)*, 14(6), 2006.

[29] C. Raiciu, M. Handley, and D. Wischik. Coupled Congestion Control for Multipath Transport Protocols. RFC 6356, July 2011.

[30] M. Allman, V. Paxon, and E. Blanton. TCP Congestion Control. RFC 5681, September 2009.

[31] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. *ACM SIGCOMM Comput. Commun. Rev.*, 26(3):5–21.

[32] J. Postel. Transmission Control Protocol. RFC 793, September 1981.

[33] Y. Chen, R. Griffith, J. Liu, R. H. Katz, and A. D. Joseph. Understanding tcp incast throughput collapse in datacenter networks. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 73–82, 2009.

[34] V. Vasudevan, A. Phanishayee, H. Shah, E. Krevat, D. G. Andersen, G. R. Ganger, G.A. Gibson, and B. Mueller. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 303–314, 2009.

[35] V. Jacobson, R. Braden, and D. Borman. TCP Extensions for High Performance. RFC 1323, May 1992.

[36] D. Wischik, C. Raiciu, A. Greenhalgh, and M. Handley. Design, Implementation and Evaluation of Congestion Control for Multipath TCP. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 99–112, 2011.

[37] D. Wischik, M. Handley, and M. Braun. The Resource Pooling Principle. *ACM SIGCOMM Comput. Commun. Rev.*, 38(5):47–52, September 2008.

[38] J. Qadir, A. Ali, Yau K. L. A., A. Sathiaseelan, and J. Crowcroft. Exploiting the power of multiplicity: a holistic survey of network-layer multipath. arXiv preprint arXiv:1502.02111, 2015.

[39] J. R. Iyengar, P. D. Amer, and R. Stewart. Concurrent multipath transfer using SCTP multihoming over independent end-to-end paths. *IEEE/ACM Transactions on Networking (TON)*, 14(5):951–964, October 2006.

[40] D. Wischik, C. Raiciu, and M. Handley. Balancing Resource Pooling and Equipoise in Multipath Transport. *Submitted to ACM SIGCOMM*, 2010.

[41] F. Kelly and T. Voice. Stability of end-to-end algorithms for joint routing and rate control. *ACM SIGCOMM Comput. Commun. Rev.*, 35(2):5–12, 2008.

[42] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[43] Apache Hadoop Project. Available from: `http://hadoop.apache.org` [accessed 1 May 2015].

[44] S. Ghemawat, H. Gobioff, and S.-T Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, 2003.

[45] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wal-Lach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4:1–4:26, June 2008.

[46] G. Decandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, 2007.

[47] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-parallel Programs from Sequential Building Blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, 2007.

[48] B. Ussery. Behind the Scenes of a Google Query. Available from: `http://blogoscoped.com/archive/2008-07-08-n70`, July 2008. [accessed 1 May 2015].

[49] Introducing data center fabric, the next-generation Facebook data center network. Available from: `https://code.facebook.com/posts/360346274145943/`, November 2014. [Accessed 1 May 2015].

[50] Cisco Data Center Infrastructure 2.5 Design Guide. Available from: `http://www.cisco.com/application/pdf/en/us/guest/netsol/ns107/c649/ccmigration_09186a008073377d.pdf` [accessed 1 May 2015].

[51] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '09, pages 202–208, 2009.

[52] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, pages 267–280, 2010.

[53] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding Data Center Traffic Characteristics. In *Proceedings of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 65–72, 2009.

[54] R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, ..., and A. Vahdat. PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 39–50, 2009.

[55] A. Greenberg, P. Lahiri, D.A. Maltz, P. Patel, and S. Sengupta. Towards a Next Generation Data Center Architecture: Scalability and Commoditization. In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, pages 57–62, 2008.

[56] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the Data Center Network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 309–322, 2011.

[57] K. C. Webb, A. C. Snoeren, and K. Yocum. Topology Switching for Data Center Networks. In *Proceedings of the 11th USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'11, pages 14–14, 2011.

[58] J. Moy. OSPF Version 2. RFC 2328, April 1998.

[59] R. Zhang-Shen and N. McKeown. Designing a Predictable Internet BackBone Network. Third Workshop on Hot Topics in Networks HotNets-III, 2004.

[60] G. Cormode and M. Thottan. Valiant Load-Balancing: Building Networks That Can Support All Traffic Matrices. In *Algorithms for next Generation Networks*. Springer-Verlag London Limited, 2010. [Chapter 2].

[61] G. Wang, D. G. Andersen, M. Kaminsky, Ng Papagiannaki, K., M. T. S., Kozuch, and M Ryan. c-Through: Part-time Optics in Data Centers. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 327–338, 2010.

[62] D. Halperin, S. Kandula, J. Padhye, P. Bahl, and D. Wetherall. Augmenting Data Center Networks with Multi-gigabit Wireless Links. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 38–49, 2011.

[63] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A Simple Network Management Protocol (SNMP). RFC 1157, May 1990.

[64] B. Vamanan, J. Hasan, and T. Vijaykumar. Deadline-aware Datacenter TCP (D2TCP). *ACM SIGCOMM Comput. Commun. Rev.*, 42(4):115–126, August 2012.

[65] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking (TON)*, 1(4):397–413, August 1993.

[66] S. Floyd. TCP and Explicit Congestion Notification. *ACM SIGCOMM Comput. Commun. Rev.*, 24(5):8–23, October 1994.

[67] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. Less is more: Trading a little bandwidth for ultra-low latency in the data center. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 19–19, 2012.

[68] D. Katabi, M. Handley, and C. Rohrs. Congestion Control for High Bandwidth-delay Product Networks. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '02, pages 89–102, 2002.

[69] Dukkipati N. Flach, T., A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, , J. Ankur, H. Shuai, E. Katz-Bassett, and R Govindan. Reducing Web Latency: The Virtue of Gentle Aggression. In *Proceedings of the ACM SIGCOMM 2013 Conference*, SIGCOMM '13, pages 159–170, 2013.

[70] T. Flach, N. Dukkipati, Y. Cheng, and B. Raghavan. TCP Instant Recovery: Incorporating Forward Error Correction in TCP. Experimental, TCP Maintenance Working Group, IETF, January 2014.

[71] R. Ludwig and R. H. Katz. The Eifel Algorithm: Making TCP Robust Against Spurious Retransmissions. *ACM SIGCOMM Comput. Commun. Rev.*, 30(1):30–36, January 2000.

[72] M. Zhang, B. Karp, S. Floyd, and L. Peterson. RR-TCP: A Reordering-Robust TCP with DSACK. In *Proceedings of the 11th IEEE International Conference on Network Protocols*, ICNP '03, pages 95–, 2003.

[73] E. Blanton and M. Allman. On Making TCP More Robust to Packet Reordering. *ACM SIGCOMM Comput. Commun. Rev.*, 32(1):20–30, January 2002.

[74] G. Parisis, T. Moncaster, A. Madhavapeddy, and J. Crowcroft. Trevi: Watering Down Storage Hotspots with Cool Fountain Codes. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 22:1–22:7, 2013.

[75] Network Simulator 3 (ns-3). Available from: `https://www.nsnam.org/` [accessed 1 May 2015].

[76] ns-3.23 manual. Available from: `https://www.nsnam.org/docs/release/3.22/manual/html/index.html` [accessed 1 May 2015].

[77] B. Chihani and D. Collange. A Multipath TCP model for ns-3 simulator. The Workshop on ns-3 (WNS3), 2011.

[78] S. Barré, C. Paasch, and O. Bonaventure. MultiPath TCP: From Theory to Practice. In *Proceedings of the 10th International IFIP TC 6 Conference on Networking - Volume Part I*, NETWORKING'11, pages 444–457. Springer-Verlag, 2011.

[79] S. Floyd and T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582, April 1999.

[80] S. Floyd, T. Henderson, and A. Gurtov. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 3782, April 2004.

[81] K. Fall and S. Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. *ACM SIGCOMM Comput. Commun. Rev.*, 26(3):5–21, July 1996.

[82] A. Appleby. Murmur3 Hash Function. Available from: `https://code.google.com/p/smhasher/` [accessed 1 May 2015].

[83] C. Dah-Ming and J Raj. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Comput. Netw. ISDN Syst.*, 17(1):1–14, June 1989.

[84] T. Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. *ACM SIGCOMM Comput. Commun. Rev.*, 33(2):83–91, April 2003.

[85] M. Allman, H. Balakrishnan, and S. Floyd. Enhancing TCP's Loss Recovery Using Limited Transmit. RFC 3042, January 2001.

[86] C. Raiciu, D. Wischik, and M. Handley. htsim. Available from: `http://nrg.cs.ucl.ac.uk/mptcp/implementation.html` [accessed 1 May 2015].