Optimising data centre operation by removing the transport bottleneck

Tobias Moncaster

University of Cambridge
Computer Laboratory

Wolfson College

February 2018

This dissertation is submitted for
the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

It is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. I further state that no substantial part of my dissertation has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University of similar institution except as declared in the Preface and specified in the text.

This dissertation does not exceed the regulation length of 60 000 words, including tables and footnotes.

Abstract

Data centres lie at the heart of almost every service on the Internet. Data centres are used to provide search results, to power social media, to store and index email, to host "cloud" applications, for online retail and to provide a myriad of other web services. Consequently the more efficient they can be made the better for all of us. The power of modern data centres is in combining commodity off-the-shelf server hardware and network equipment to provide what Google's Barrosso and Hölzle describe as "warehouse scale" computers.

Data centres rely on TCP, a transport protocol that was originally designed for use in the Internet. Like other such protocols, TCP has been optimised to maximise throughput, usually by filling up queues at the bottleneck. However, for most applications within a data centre network latency is more critical than throughput. Consequently the choice of transport protocol becomes a bottleneck for performance. My thesis is that the solution to this is to move away from the use of one-size-fits-all transport protocols towards ones that have been designed to reduce latency across the data centre and which can dynamically respond to the needs of the applications.

This dissertation focuses on optimising the transport layer in data centre networks. In particular I address the question of whether any single transport mechanism can be flexible enough to cater to the needs of all data centre traffic. I show that one leading protocol (DCTCP) has been heavily optimised for certain network conditions. I then explore approaches that seek to minimise latency for applications that care about it while still allowing throughput-intensive applications to receive a good level of service. My key contributions to this are Silo and Trevi.

Trevi is a novel transport system for storage traffic that utilises fountain coding to maximise throughput and minimise latency while being agnostic to drop, thus allowing storage traffic to be pushed out of the way when latency sensitive traffic is present in the network. Silo is an admission control system that is designed to give tenants of a multi-tenant data centre guaranteed low latency network performance. Both of these were developed in collaboration with others.

Acknowledgments

I would like to acknowledge the tremendous support I received from my supervisor, Professor Jon Crowcroft. His boundless positivity and enthusiasm for ideas have been a real source of inspiration and his fame has helped open many doors for me. I would also like to acknowledge the generosity of my second Advisor, Dr. Andrew Moore. He has given me access to some of the best computing resources in the department and has suffered my presence within his lab for the entirety of my PhD.

While I have been a member of the Computer Laboratory I have collaborated with several other people, notably Dr. George Parisis with whom I developed Trevi; Dr. Anil Madhavapeddy who was my co-author on Trevi and PVTCP; Dr. Steve Hand and Zubair Nabi who were co-authors of PVTCP. Finally I owe a debt of gratitude to Dr. Richard Gibbens for sharing his expert knowledge of R.

The EPSRC INTERNET (Intelligent Energy Aware Networks) project funded me throughout the first three years of my PhD and without that funding I would not have had this opportunity. The Computer Laboratory provided me with funding for my tenth term, helping me to move a step closer to completing my dissertation and the Cambridge Philosophical Society provided me with both travel funding and a studentship which enabled me to complete my dissertation.

I also need to acknowledge the support of Wolfson College who provided outstanding pastoral care, accommodation and a life outside the department as well as providing generous travel grants.

During the course of my PhD I have been directly involved with the European Union's Seventh Framework Programme Trilogy 2 project, grant agreement 317756. This has been an invaluable experience and has allowed me to exchange ideas with some of the leading European figures in my field including Professor Mark Handley from UCL and Dr. Bob Briscoe. I was also fortunate to be given a 12 week internship at Microsoft Research, Cambridge working with Dr. Hitesh Ballani, Dr. Keon Jang and Justine Sherry. This resulted in the work on Silo which forms a key part of this dissertation.

Finally I want to thank Sasha East for being my only true friend when I was in a very dark place. The true test of friendship is when people stand by you in times of adversity.

Contents

List of Figures

List of Tables

Glossary

The following acronyms are used in this dissertation:

AIMD Additive Increase, Multiplicative Decrease

API Application Program Interface

AQM Active Queue Management

BIC Binary Increase Congestion Control

BRAS Broadband Remote-Access Server

CDN Content Delivery Network

CoDel Controlled Delay AQM

COTS Commodity off-the-shelf

DCCP Datagram Congestion Control Protocol

DCE Direct Code Execution

DCTCP Datacenter TCP

DFS Windows Distributed File System

DMA Direct Memory Access

EC2 Elastic Cloud Compute

ECMP Equal Cost Multipath

ECN Explicit Congestion Notification

EEE Energy Efficient Ethernet

FDS Flat Datacenter Storage

GFS Google File System

GUID Globally Unique Identifier

HFT High Frequency Trading

IaaS Infrastructure as a Service

IETF Internet Engineering Task Force

IPC Inter-Process Communication

IRTF Internet Research Task Force

KVM Kernel-based Virtual Machine

LAN Local Area Network

LEDBAT Low Extra-Delay Background Transfer

LPI Low Power Idle

LSO Large Segment Offload

MPTCP Multipath TCP

MTU Maximum Transmission Unit

NAS Network-Attached Storage

NAT Network Address Translation

NDIS Network Driver Interface Specification

NFS Network File System

NIC Network Interface Card

NSC Network Simulation Cradle

NTFS Windows NT File System

OLDI Online Data-Intensive

PCN Pre-Congestion Notification

PIE Proportional Integral Controller Enhanced AQM

PMTUD Path MTU Discovery

PUE Power Usage Effectiveness

PVTCP Polyversal TCP

QoS Quality of service

RDMA Remote DMA

RED Random Early Discard

RTO Retransmission Timeout

RTT Round Trip Time

SACK Specific Acknowledgement

SAN Storage Area Network

SCTP Stream Control Transport Protocol

SDN Software Defined Networking

SNMP Simple Network Measurement Protocol

SSD Solid State Drive

SSL Secure Socket Layer

TAPS Transport Services or Transport Services Working Group

TCP Transport Control Protocol

TFRC TCP Friendly Rate Control

TLT Tract Locator Table

UDP User Datagram Protocol

VL2 Virtual Layer 2

VLB Valiant Load Balancing

VM Virtual Machine

WAN Wide Area Network

Chapter 1

Introduction

Data centres lie at the heart of the modern Internet. They encompass everything from massive warehouse scale computers hosting search engines to cloud providers hosting numerous mobile start-ups. Their influence can be felt in all parts of modern society including finance, commerce, industry and people's social lives.

Modern data centres consist of thousands of cheap "off-the-shelf"[1] servers connected together with a high speed network. Barroso and Hölzle describe such a set-up as a warehouse-scale computer [14] (see Figure 1.1). Such data centres rely on their high speed internal network to share data between compute nodes and to allow multiple compute nodes to act as a single computing resource. In the early days, data centres were an evolution of high-performance computers so they often used proprietary networking technologies such as InfiniBand. However operators are increasingly moving to standard high speed Ethernet running at up to 40Gbps (and much InfiniBand hardware now also supports Ethernet). The reasons for this are prosaic—it is cheaper and easier to use off-the-shelf hardware and it is easier to find and train technicians to operate such hardware.

However networking data centres creates some unique problems, which has made this a hot topic of research in recent years. Advances have been made at all layers of the stack. At the physical and data link, novel topologies like Fat-tree [3] and CamCube [1] and new datalink protocols like VL2 [52] aim to provide location-agnostic network performance. At the datalink and network layer, efforts have been made to increase the number of addresses that can be reached without the need for routing [146]. At the transport layer, new transport protocols like DCTCP [5] or HULL [7] seek to improve the performance of latency-sensitive applications without sacrificing too much performance for long-running connections. At the application layer major advances include Partition-Aggregate schemes like Ciel [103], MapReduce [35], and Hadoop [58] which divide workloads between multiple worker nodes before aggregating the responses and distributed Key-Value stores like

[1] Often large data centre operators use custom chassis and motherboards, but the components themselves are largely standard server hardware. See `http:www.opencompute.org` for details of the Open Compute Initiative (accessed February 2018).

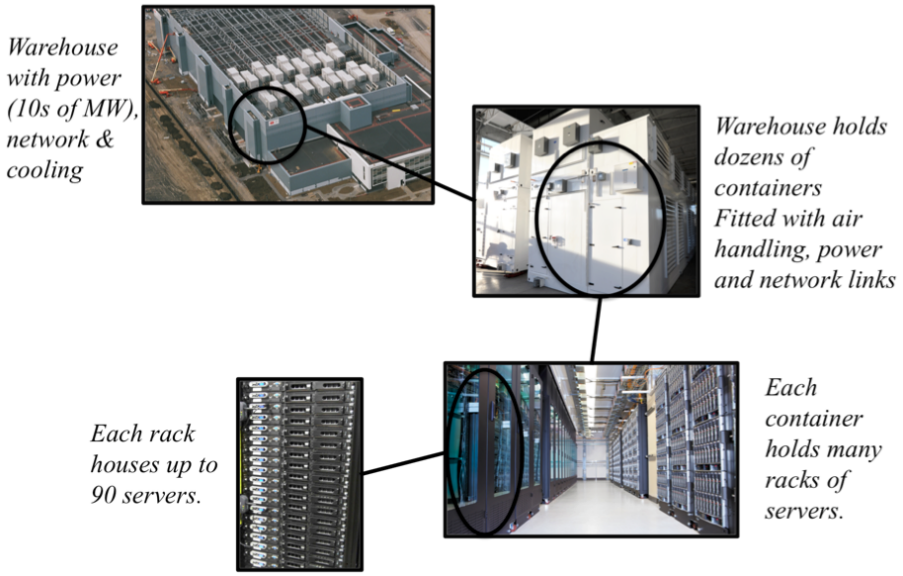

Figure 1.1: Data centres consist of large warehouses full of servers

Figure 1.2: Data centres often use containers to offer improved modularity and efficiency gains (image from Microsoft)

memcached [43], which offer fast access to data stored in RAM. There have even been suggestions like NDP [63] that re-architect the entire stack.

There are three key differences between the Internet and data centre networks. Firstly, round trip times within data centres are extremely short. Even allowing for delays in serialising data on and off the network they are typically of the order of 1 µs. By contrast, in the Internet round trips are more usually measured in milliseconds. Secondly, data centre networks are usually designed to minimise or remove bottlenecks. This means that, unlike in the Internet, end hosts are able to send at extremely high data rates (Gbps). Thirdly, there are significant differences in the service that traffic wants to receive. In the Internet, equivalent flows are expected to be treated equally (so-called TCP Fairness), and in many cases a flow wants to maximise its throughput. But in

data centres, latency generally matters far more than throughput, and flows need not be treated equally. While changes can be made at the datalink, network and application layers, the layer that traditionally has control over such end-to-end flow characteristics is the transport layer, and hence that is what I have concentrated on in this dissertation.

Most modern data centres rely too heavily on TCP and its variants. TCP was originally designed for use in the Internet and has been optimised to maximise throughput across the wide area. However within the data centre context there is a mix of traffic, much of which cares more about latency than throughput. Consequently TCP often becomes a bottleneck for performance.

Different data centre applications need different characteristics from their transport protocol. My thesis is that allowing them to choose a suitable protocol will give them improved performance compared to just using TCP.

1.1 TCP. The new narrow waist

Over the past decade, it has become apparent that the Internet is suffering from transport ossification. TCP has become the de facto narrow waist of the Internet, transporting almost all the bytes crossing the network. There are a number of reasons for this: Middleboxes such as stateful firewalls and intrusion detection systems assume all traffic must be either TCP or UDP; Network Address Translators (NATs) use TCP and UDP port numbers to multiplex many connections over a single IP address. Many developers have become used to the reliable, ordered byte-stream abstraction offered by TCP. Taken in combination this has meant that the only way a new transport can be deployed is if it looks like TCP or UDP on the wire. Perhaps surprisingly, these issues exist in data centres as well. Data centres rely on commodity hardware as it tends to be both reliable and cheap and often the network stack is virtualised. Consequently even data centres suffer from issues relating to over aggressive middleboxes that mistake novel protocols for attacks, or that seek to randomise the TCP sequence space to protect against non-existent sequence injection attacks. Indeed data centres often utilise additional highly specialised middleboxes such as load balancers, which only serve to exacerbate the problem.

1.1.1 Transport abstractions

TCP and UDP offer specific abstractions that are well suited to particular applications. TCP is designed for the reliable transmission of bulk data while UDP is intended to send short messages relating to interactive applications such as Telnet. Up till the mid-1990s these two applications covered pretty much all the uses of the Internet. However, the arrival of the World Wide Web with its concomitant growth of online commerce and social networking, the invention of Voice over IP, the explosive growth in streaming media

and the growth of interactive online services have changed the requirements for transport protocols.

Over the past two decades there have been numerous attempts to define new transport protocols that offer different abstractions and make different assumptions about what service the end user wants. SCTP [157] offers a message abstraction and separates control data from message data. TFRC [46] offers a stream abstraction but is intended to be better suited to real-time data that cannot adapt its rate rapidly. The datagram congestion control protocol (DCCP) aims to offer a suite of different congestion control mechanisms within a single wrapper. However, there has been almost no adoption of these new standards in the wider Internet. There have been notable success stories such as MPTCP [47] and QUIC [138, 24], but these have relied on concealing themselves as TCP and UDP respectively.

As I discuss in Section 1.2, data centres have a number of different types of traffic that map especially poorly to the abstractions provided by TCP and UDP. Even where the service they wish to receive does map well (for instance storage traffic mapping to TCP), using TCP or UDP can have a negative impact on the more interactive and latency-sensitive flows.

1.2 Data centre traffic types

One of the key challenges for data centre networking is the conflicting needs of different applications that all have to share the same network. Broadly speaking these applications produce traffic that can be grouped into three types. These are bulk data applications, online data intensive applications and short message traffic.

Bulk Data Traffic

Bulk data applications include storage as well as background maintenance tasks such as taking backups or writing disk images to machines that have been brought online. Typically these applications are elastic. Elastic applications still get some utility even at relatively low data rates, but usually want to receive a high average data rate. Importantly however, they are not sensitive to the rate they receive at any given moment in time. Storage traffic can come in a number of forms and depends to an extent on the particular way the data centre is set up. Some storage will be on traditional block stores, either on a network attached storage system or using a modern distributed block storage system such as Google Filesystem [51]. However some data centre storage is actually transient in-memory storage. Typically this might be used for distributed key-value stores data has to be retrieved as fast as posible.

Online Data Intensive Application Traffic

OLDI applications are interactive applications such as web search and e-commerce. OLDI applications are highly latency sensitive. Generally, they involve the transfer of short flows consisting of several packets of data at a time. What matters here is not the individual packet latency but rather the latency of the entire transfer (the so called FCT or flow completion time).

Short Message

Short message traffic is mainly related to controlling partition-aggregate style applications where a control node sends a job to many other nodes simultaneously. This generates a burst of very short control messages (often only one or two packets long), which in turn generates a burst of acknowledgements coming back that can overwhelm the queue at the control node, a phenomenon known as incast [27].

1.2.1 Traffic patterns

Getting accurate measurements of data centre traffic has proved extremely hard. Part of the problem is that data centre operators view such information as commercially sensitive. But there are also significant problems with capturing traffic information at such high speeds and on such a large scale. There are a few papers that publish the results of data centre traffic analysis. These are summarised here and are explained in more detail in the next chapter.

Kandula *et al.* [81] measured a 1,500 node operational cluster over a period of more than two months. They observed that data centres exhibit strong traffic patterns that are closely related to the type of traffic being carried. They identified two broad patterns of traffic: Work Seeking Bandwidth and Scatter-Gather (see 2.4). Work seeking bandwidth traffic exhibits a large number of intra-rack flows and far fewer inter-rack flows. By contrast scatter-gather traffic leads to a large number of inter-rack flows. These patterns reflect OLDI and short message applications respectively.

In their paper on DCTCP, Alizadeh *et al.* did a detailed analysis of the traffic flowing across a single pod of an un-named data centre [5]. They observed that traffic splits into three categories: short messages, query traffic and long-running background traffic.

These observed traffic patterns can be mapped back to the traffic types listed above. Short message traffic shows up as a large number of short one to many flows. OLDI traffic consists of multi-packet many to many flows with a slight preference for rack locality. Bulk data tends to mainly be long running one to one flows.

More recent work from Facebook [141] shows more complex traffic patterns. In this network there are significant volumes of cache traffic and many flows span between multiple data centre sites. The result is a traffic matrix that shows neither strong rack-locality nor

all-to-all features. The traffic patterns are also stable across time spans of up to days, but the specific set of heavy hitters changes rapidly.

1.3 Data centre transport protocols

Transport protocols are a key element of any network and provide a number of functions (see Appendix A.1). As with the wider Internet, TCP is usually the transport of choice within a data centre, however as I explain below this is not necessarily ideal.

1.3.1 TCP issues

There are a number of well known issues with using TCP within a data centre. Put simply, TCP is not designed to work in the data centre environment. TCP has been optimised for relatively low speed connections with round trip times measured in milliseconds. Modern variants like FAST [163] allow it to function better over high delay-bandwidth product networks. However, in a data centre you have a combination of extremely high bandwidth and extremely low latency. Indeed, often the majority of the latency is not down to the actual network but is caused by serialisation/deserialisation delays in the end system NICs.

This leads to a couple of widely reported issues of which the most damaging is TCP incast [27]. This describes the situation where a large number of flows arriving simultaneously at a switch cause the buffer to overflow losing all the ACKs for a short message flow. This in turn causes TCP timeouts which can have a devastating impact on throughput and latency.

1.3.2 Specific transport issues for data centres

One of the unique issues for data centre transport protocols is that computations are often spread across multiple virtual nodes that may reside anywhere within the data centre. Nodes often migrate from one physical location to another, but still want to maintain their network sessions. So at one extreme, two nodes may reside on the same piece of silicon, and at the other extreme they could be on opposite sides of the physical network with several intermediate switches between them. The growth of multi-core machines has meant that the problem of intra-process communication on the same piece of silicon has been solved using shared memory and zero copy transports like FABLE [154]. Systems like FARM [37] enable RDMA (Remote Direct Memory Access) within data centres. This allows a node to access the memory of a remote node. This means shared memory transports become feasible for a wider range of applications.

Data centre architects attempt to reduce the impact of location by designing full partition-bandwidth networks where all paths between all nodes receive the same bandwidth. However the latency between different nodes can vary by an order of magnitude or more. Also if several nodes are communicating with one destination, even with the bandwidth available the queue can overflow. Since application developers have no knowledge of where an application will physically reside, they tend to make the conservative choice of using TCP as they know this will work in any situation.

1.4 Main contributions

This dissertation makes the following contributions.

1.4.1 Performance analysis of DCTCP

Within data centres one new transport protocol is reported to have gained significant traction. DCTCP, or Data Center TCP [5] is a version of TCP that is designed to favour short foreground flows over longer-running background flows. However, in Chapter 3, I present simulation results that suggest it may not work so well at the long tail. DCTCP uses a combination of ECN [137], a modified AQM marking algorithm and a congestion controller that responds to the rate of congestion marks that it sees.

DCTCP ensures short flows see low latency by not responding at all to congestion until a flow has seen more than one mark and by ensuring that any queues in the network are kept short by using an aggressive form of AQM. While my results show that DCTCP performs well for most flows, some flows can suffer enormous delays because they trigger timeouts. I also present results that suggest that simply adopting DCTCP's more dynamic AQM algorithm gives better overall performance, although at a slight cost in the median flow completion times.

1.4.2 Data centre storage

Chapter 4 presents the Trevi storage system [120]. Trevi uses fountain coding and multicast networking to provide a new approach for data centre storage. Trevi is a blob store, designed to allow storage traffic to act as a scavenger class, receiving lower priority at switches and allowing latency sensitive traffic to pass unhindered. Trevi uses a receiver driven flow control mechanism. This is ideal for storage systems where there is a potential for a mismatch between the size of a request and its response and where the instantaneous performance of the storage system can vary unpredictably depending on things like seek time.

The two main gains that multicast gives Trevi are the ability to easily replicate data and even more significantly, the ability to multi-source data from different replicas, allowing you to retrieve data from storage in a fraction of the time. The use of sparse erasure codes also means that if data is lost you do not need to retransmit exactly the same packets, you simply need to receive enough extra code blocks.

At the end of the chapter I present the results of extensive ns2 simulations of a simplified version of the Trevi protocol. These compare its performance against TCP-SACK and DCTCP. The results seem to indicate that Trevi improves the FCT times for foreground flows without adversely affecting the storage traffic. Given that the simulations do not use multicast, it is reasonable to suggest that the performance of the storage flows might be expected to also improve. A key finding is that Trevi is sensitive to changes in transmission rate, consequently it would perform best with active flow control. Flow control is discussed in some detail in section 4.4.3.

Trevi was joint work with others in the Computer Laboratory. My contributions to the work were: the idea of using sparse erasure coding in order to create a storage protocol that was able to operate as a scavenger class; basing the protocol on Microsoft's Flat Datacenter Storage[107]; the whole section on flow control. I also did all the evaluations at the end of the chapter.

1.4.3 Multi-tenant data centres

Chapter 5 presents the Silo system [78], a tenant admission system designed to offer predictable message latency in multi-tenant data centres. In the chapter I argue that to achieve truly predictable latency, a general cloud application needs guarantees for its network bandwidth, packet delay and burstiness. I show how guaranteeing network bandwidth makes it easier to guarantee packet delay and how this insight drives the VM placement system.

Silo enables bandwidth, latency and burst guarantees without any network or application changes, relying only on VM placement and end host packet pacing. Silo depends on a novel placement algorithm that uses network calculus to ensure that the requested guarantees can be met. The prototype achieves fine grained packet pacing with low CPU overhead. The evaluation shows that Silo can ensure predictable message completion time for both small and large messages in multi-tenant data centres without a high cost in terms of efficient utilisation of resources.

Silo came out of work I did during my internship at Microsoft Research in Cambridge. My main contributions to Silo were the design of the policer mechanisms that are used to enforce the guarantees, and the design of the discrete event simulations (both the micro benchmarks and the large-scale results) as well as identifying network calculus as a solution for calculating the impact of VM placements on the network.

Chapter 2

Background and related work

This chapter provides a summary of the current research in the field of data centre networking. For completeness it includes sections looking at data centre network architectures and application frameworks as well as related work on subjects such as software defined networking. It is divided into sections looking at Hardware & Architectures, Networking Protocols, Software and Data Centre Storage. The final section looks at simulation, and emulation of data centres.

2.1 Data centre hardware and architectures

Although there is a large degree of heterogeneity, most large scale production data centres seem to follow similar underlying architectural principles.[1] They are usually based on a two or three layer network topology, and use containerisation for the actual computing hardware. The hardware is often Commodity off-the-shelf (COTS) but increasingly large operators are producing customised designs for things like motherboards, server chassis and racks (which have a direct impact on the cooling of the facility). Some reports also suggest they may be designing custom switches, but these will still be based on the same switch silicon that commodity switches use.

The standard architecture is largely driven by external environmental factors. Key among these are the need to be able to physically access servers to deal with equipment failures, the need to provide efficient cooling, limitations on power distribution and the complexity of wiring such a large number of machines (see figure 2.1). There have been novel proposals for completely different physical architectures, but to my knowledge these still exist only on paper. I explore these later in this section.

[1] For the purposes of this section I am using data centre to refer to large warehouse scale installations with thousands of physical nodes. Data centre is an ambiguous term and is often used to describe much smaller facilities.

Figure 2.1: Data centres need significant amounts of wiring (image from Google)

2.1.1 The three layer data centre topology

In essence a data centre is just a collection of servers and storage nodes connected together with a high speed network. Many production data centres (especially those used for co-location) use Cisco's three-layer network topology[30]. This is based on a logical tree structure similar to that found in many LANs. Servers are arranged in racks and are joined together using a Top of Rack (ToR) switch. A number of racks are then grouped together into a cluster or pod using aggregation switches. Usually these connections are faster than the ones within the rack. Finally the clusters are joined using a routed core network, which is often fully meshed. This simple topology is shown in Figure 2.2.

Latency is a critical aspect of data centre design. Consequently it is essential to keep network latency to a minimum. In turn this means that fast Ethernet switches are the preferred switching hardware. However, even the fastest production switches only have switching tables able to handle tens of thousands of hosts. So to scale to a data centre with hundreds of thousands of servers requires the use of slower layer 3 (IP) routing in the core. Over the past 5 years there has been a shift towards using alternative topologies that offer full bisection bandwidth which are discussed below.

2.1.2 Full bisection bandwidth architectures

One of the goals of data centre network architects is to increase the bisection bandwidth available between every pair of nodes within the data centre. The nearer this becomes to being equal the less dependence you have on the physical location of nodes. This in turn makes job scheduling easier.

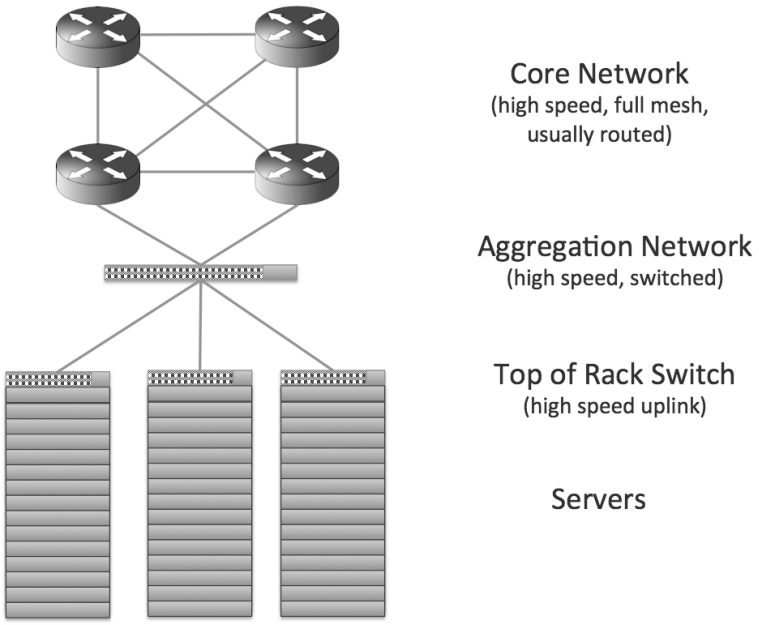

Figure 2.2: The simple 3-layer data centre architecture

2.1.3 Physical architectures

In its early incarnation, the hierarchical three-layer architecture described above provided some redundancy at the aggregation network (with switches being dual-homed) and a fully meshed core. Later incarnations saw the use of high speed links within the aggregation and core networks to try and reduce the level of oversubscription at these layers. By contrast, multi-stage switching architectures use techniques from telephone circuit switches to try and create full bisection bandwidth networks. Most of these topologies are based on Clos networks [31] or folded Clos networks. Clos networks were designed for non-blocking telephone circuit switches where the number of links exceeds the size of any feasible single switch element. A strictly non-blocking circuit switch provides you with a packet switched network that exhibits full bisection bandwidth while a rearrangable non-blocking switch provides near full bisection bandwidth when coupled with suitable flow scheduling.

The Fat-tree architecture[3] uses the eponymous fat tree folded Clos network. Fat-tree networks provide full bisection bandwidth across the whole data centre network. The network is a five stage Clos network which equates to three logical layers in the data centre. All switching elements in a Fat-tree network are identical. A network created from *k-port* switches is described as being a *k-ary* fat-tree. Such a network has k pods each containing $(k/2)^2$ servers connected to two layers of $k/2$ switches. The pods are connected to a core with $(k/2)^2$ nodes. A $k = 4$ network is shown in figure 2.3. Despite the increased complexity of the wiring, many of the largest data centres now use such networks for the performance improvements they give.

VL2[52] aims to create a virtual layer 2 network spanning the entire data centre. To

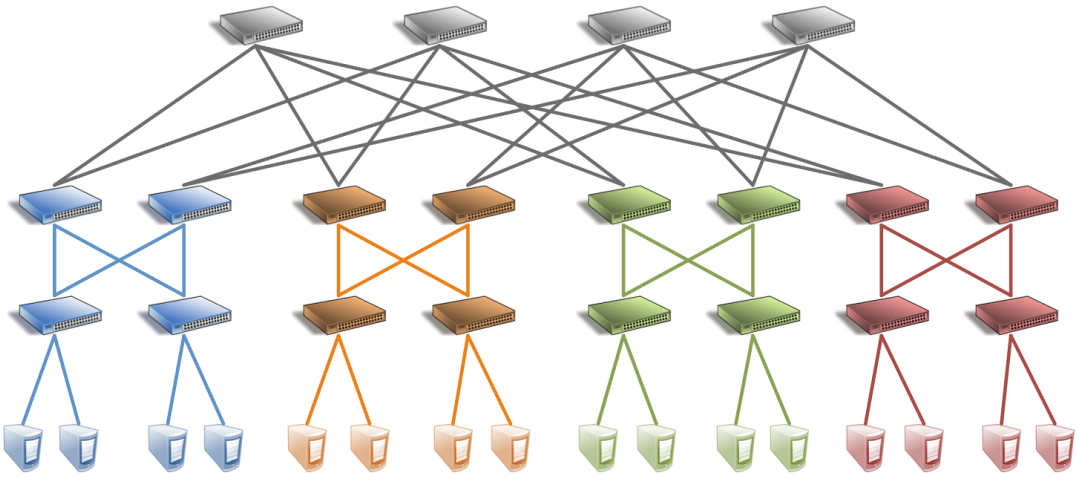

Figure 2.3: A $k = 4$ Fat Tree network

do this it uses flat network addressing, end-system address resolution and valiant load balancing (VLB)[82]. VLB spreads the load across the network without the need for central coordination through the use of a folded Clos network coupled with randomisation of path assignment. Having a single domain across the entire data centre avoids the need for slow path routing which would add significant latency to any network transfer.

BCube[55] advocates the provision of high data centre network connectivity by combining a large number of small, cheap switches arranged in a 3 (or more) dimensional array. In BCube, servers act as both sources and relay nodes, with each server being extensively multi-homed. Data is source routed and the network is optimised for bandwidth intensive applications. It exhibits graceful failure when switches and servers fail. In the related CamCube architecture[1], nodes are connected in a dense 3-D torus with each server connected to 6 others. As with BCube, servers participate in routing, and multiple routing strategies can exist in the same network. Both these network topologies exhibit desirable properties, but they scale poorly as the complexity of the wiring goes up exponentially with the size of the network.

2.1.4 Switch scheduling

Multi-stage switching topologies achieve full bisection bandwidth by providing multiple equal cost routes between every pair of nodes. In order to make eficient use of this additional bandwidth, switches have to schedule flows across all available routes. The simplest way to do this is using standard Equal Cost Multipath routing (ECMP). This hashes the 5-tuple of source and destination address, source and destination port and protocol ID and then assigns equal numbers of flow hashes to each available path. The trouble with this approach is that not all flows are equal. This is particularly true in data centres where the majority of flows are short (it is often claimed that 80% of the bytes

are carried by just 20% of the flows).

Hedera[4] tries to place large flows so as to achieve near optimal throughput across the network. Its central scheduling achieves up to 96% of the optimal bandwidth. They compare the performance of ECMP against two novel algorithms: global first fit and simulated annealing. Global first fit simply places large flows on the first available path that can accommodate the flow end-to-end. Simulated annealing seeks to find a near optimal placement for large flows by minimising the total excess capacity needed for the set of flows within a given number of iterations of the algorithm. Their results show that simulated annealing consistently outperforms both ECMP and global first fit and is not too computationally intensive across a range of different traffic patterns. However, it does require singificant changes to the control layer within the switches.

2.1.5 Software defined networking and OpenFlow

Software defined networking (SDN) is the generic term for any networking technology where you can re-configure the control plane in software and hence are free to implement non-standard routing, switching and marking protocols.

OpenFlow[98] is the most widely adopted SDN protocol to date. It was originally developed specifically to allow researchers to experiment with non-standard protocols over a real network without impacting existing traffic. OpenFlow is relatively simple - as flows arrive at the switch they are matched against a flow table (much as happens already for switching). The major difference is that OpenFlow allows matching on a far higher number of protocol fields (transport, network and datalink). Flows can then be switched as usual, directed to specific output ports or passed up to the OpenFlow controller for further processing.

While OpenFlow is a logically centralised system, controllers can be organised as a hierarchy. At each level of the hierarchy flows can be matched to rules, or if no rule is found, can be passed up the hierarchy. Once a rule is found, or a new rule is defined, the controllers can pass this down through the hierarchy to the actual switches.

Early OpenFlow switches and controllers suffered from poor performance[140] and only had small flow tables, limiting their utility for data centre networks. However there are now a number of hardware vendors selling production OpenFlow switches capable of being used in data centres[2]. In the Open Networking summit in 2012 Google revealed that they use a version of OpenFlow within their data centres [65]. Specifically they use it to manage their backbone network as it allows them to run centralised traffic engineering to give graceful recovery from failure. This approach also makes it easy to test “what

[2]These include the HP 8200 series (with up to 96 10GE ports), the IBM G8264 (up to 64 10GE ports) and the NEC PF5820 (up to 48 10GE ports). NEC also sell high-speed OpenFlow controllers that can integrate with other vendors’ switches.

if" scenarios. They claim the result is "A WAN that is higher performance, more fault tolerant, and cheaper".

2.2 Data centre network protocols

There is a significant and growing body of research looking at data centre network protocols. This reflects how critical the network is to the operation of any data centre. As mentioned in Chapter 1, many data centre applications are extremely latency sensitive. According to their then V.P. of Search, Google discovered that simply increasing the number of results returned from 10 to 30 reduced the number of searches performed by 20%[3]. Further investigation showed this was because the time taken to return the results was increased from 0.4s to 0.9s.

Data centres are based on general purpose operating systems such as Linux. As a result the applications are often built on top of the standard TCP/IP suite. But TCP was never intended for high-speed, low-latency data transfers. The limited research that has been published on data centre traffic patterns (see Section 2.4) also suggests that data centres have a high proportion of short flows (certainly much higher than the Internet). This prevalence of short flows has led to a number of proposed improvements including new transport protocols and the use of alternative datalink technologies. These are summarised here.

2.2.1 Transport protocols

Data Centre TCP (DCTCP)[5] is a protocol designed to favour short flows. It works by ECN marking[137] all packets as soon as any queue builds at the switch or router. The sender then reduces its transmission rate in response to the rate of marks it sees. This has the effect of forcing large, long-running flows to back off, creating more space in the network for short flows.

The authors of DCTCP state this clearly: "The main goal of DCTCP is to achieve high burst tolerance, low latency, and high throughput, with commodity shallow buffered switches."

Multipath TCP (MPTCP)[135] allows a single TCP connection to use multiple simultaneous paths. Each path carries a sub-flow and the MPTCP controller spreads the load between them to spread the traffic load more evenly across the network. In turn this reduces congestion, removes blocking and increases the size of the resource pool. The standard 3-tier architecture (Section 2.1.1) means that there is always more than one

[3]See `http://assets.en.oreilly.com/1/event/29/Keynote%20Presentation%202.pdf` for more details (accessed February 2018).

path between any two servers that are not in the same rack. Consequently MPTCP has the ability to significantly improve the performance of data centre networks.

Energy-Efficient Congestion Control [49] is a new proposal for improving the energy efficiency of networks without causing TCP to over-react and adversely affect the throughput. The authors propose the use of dynamic bandwidth adjustment in concert with RED [45] and TCP. Their algorithm attempts to dynamically adapt the bandwidth at each link according to an optimisation of local link state (average buffer size) and a given source rate. They claim their simulations demonstrate that the system works, however they are still only preliminary results.

2.2.2 Physical and datalink layers

cThrough[162] is a proposal for using optical circuit switching to provide additional core network capacity to carry background traffic. This has the effect of reducing contention for shorter flows. There are some issues with this approach since circuit switching is not ideally suited to packet switched traffic. However, the authors claim that their approach achieves the same performance as a network with full bisection bandwidth such as fat-tree[3]. Some researchers have also investigated the use of wireless technology as a means of providing additional capacity (wireless flyways[59]) and even as the basis for radical new topologies (see below).

At the datalink layer there has been a lot of interest in the use of Software Defined Networking (see §2.1.5) in data centres. ElasticTree[64] advocates the use of OpenFlow[98] switches to create networks that are able to selectively turn off links to reduce energy use by up to 50% while maintaining an appropriate level of fault tolerance and the ability to handle traffic surges. Hedera[4] detects large flows at the edges of the network and uses OpenFlow to set up suitable paths for these, leaving the rest of the network free to handle shorter flows. This achieves better utilisation of the network and performs better than static load-balancing. Researchers at the Stanford Experimental Data Center Laboratory[147] are also looking at ways to integrate OpenFlow into data centre networks.

Energy Efficient Ethernet or IEEE 802.3az [29] is a modified form of Ethernet which uses a low power idle (LPI) mode to reduce power consumption when there is no traffic on the link. If the link senses that it is idle it usually sends an idle message to indicate that the link is free. However, in EEE it can instead send an LPI notification. This causes all the transmitters to move to a low power "sleep" mode. After a period of time the transmitter can stop sending LPI and the link becomes dormant apart from periodic keep-alive signals. Once the transmitter has new data to transmit it sends a wake signal which wakes the link up. There has also been related work looking at the possibility of dynamically adapting the link rate to reduce energy consumption [54]. The initial results from that work have not been good, as it took too long to renegotiate to a lower rate.

2.2.3 NDP—a novel hybrid approach

DCTCP took the approach of adapting existing hardware and software stacks in order to improve performance. NDP[63] takes a radically different hybrid approach. Rather than try to improve performance incrementally based on existing approaches, the authors asked the question, how would one design a data centre network from scratch. They took several existing ideas and combined them into an elegant clean slate design.

The starting point is the observation that serialisation of data packets onto the link dominates the end-to-end latency. This means that in the time it takes a packet to be serialised, a control packet (e.g. an ACK), can traverse the entire network. Another key observation is that data loss in itself isn't an issue. What matters is loss of metadata. The third observation is that in data centres where receivers are coordinating flows from multiple senders, it is the receiver that best knows how to prioritise these flows (see 2.3.3 for an explanation of why this is).

In NDP, senders are allowed to send their first window of data at line rate. Intermediate queues are kept very short (maximumm of 8 packets). If the queue overflows, the packet data is trimmed and the header is priority forwarded. At the receiver, pull packets are sent back to senders to request new data, or to request retransmissions for missing data. After the initial window sent at line rate, senders can only transmit packets when they see a pull packet and so the network quickly reaches a stable operating point.

Because the receiver is in control and rate limits the pull packets, this ensures the aggregate rate seen at the receiver is exactly the available line rate. This completely solves the incast problem. Not only this, the receiver can also prioritise the data it needs to see. This aspect in particular is a much better fit with how MapReduce (see section 2.3) and similar frameworks are used.

NDP achieves astonishingly good performance, combining extremely high network utlisation with low latency. However because it is a complete clean-slate design requiring both custom switch hardware and a custom network stack, it is unlikely to be used in production data centres in the near future. However, the approach is extremely elegant and is bound to influence design decisions in future.

2.3 Data centre software

Data centres are used for four primary purposes. These are to perform analysis on large datasets (e.g. searching an index of websites), to perform high performance computing tasks (e.g. processing the data from high energy physics experiments), to provide networked storage (for content delivery networks and cloud-based services such as DropBox) and to provide virtualised servers and cloud services for business and personal customers.

Each of these purposes requires specialised software approaches, some of which I discuss briefly below.

Google pioneered the use of the MapReduce paradigm[35] for searching large datasets quickly and efficiently. In MapReduce the main job is decomposed into a number of smaller tasks. Each of these is then mapped to a server which processes the tasks assigned to it. Once a task has been processed the result is returned to a reduce server. This accumulates all the results and returns the final result. This has sparked a number of similar *partition-aggregate* approaches such as Hadoop[58] (an open source implementation of MapReduce), Dryad[73] (which increases the complexity of the data that can be handled) and CIEL[103] (which adds the ability to run iterative and recursive algorithms).

2.3.1 TCP Incast

TCP incast[27, 28] occurs when a large number of packets arrive at a switch at the same time. This causes the switch buffers to overflow and hence a large number of packets to be dropped. Usually this does not lead to too many issues, but there are circumstances where it becomes pathological. If the packets are a stream of TCP acknowledgements then this can trigger a TCP time-out adding significant delay. This becomes particularly bad when the acknowledgements are all related to a single application such as a MapReduce job or writing data to storage. In such cases, the problem can snowball as the re-transmitted packets trigger round after round of buffer overflows and retransmissions.

There seems to be some debate as to whether incast really poses a major problem or not. Benson *et al.*[18] saw no evidence for incast, but they accept that they only had limited access to TCP flow-level statistics. If it does exist it poses a challenge to the network and transport within the data centre. Although no complete solution exists, partial solutions include significantly reducing the TCP retransmission timeout[159] and changing the transport protocol and marking algorithms to give more priority to short-lived flows as is done by DCTCP[5].

2.3.2 TCP outcast

TCP outcast was identified by Prakash *et al.* [131]. It refers to the case where sets of flows destined to a common output arrive at a switch at the same time. This is common in multi-rooted tree topologies with scatter-gather traffic patterns (a common occurrence in data centres). They observed that where the sets of flows are of different size the smaller set is disproportionately penalised because of how TCP reacts to losses.

2.3.3 Stragglers

Partition-aggregate schemes also suffer from a particular issue when tasks get delayed, rescheduled or fail near the end of the Shuffle phase (when all data is collected ready for the Reduce phase). These tasks are known as stragglers and they mean the overall job takes much longer to complete, badly affecting performance and causing potential problems for the network.

Solutions to this include reducing the chances of job scheduling failure through the use of delay scheduling[173], the use of speculative scheduling for late-completing tasks to increase the chance of completion[58] and the use of co-workers to share the load for any task that is in danger of becoming a straggler[68].

2.3.4 Virtualisation

Virtualisation is the ability to present a set of real physical resources as a number of virtual machines (VMs). This is achieved through the use of a hypervisor which is responsible for sharing the real resources between the virtual machines and providing transparent interfaces between the virtual machines and the outside world. The three main hypervisors on the market are Xen[170], KVM[85] and VMWare[161].

The "traditional" paravirtualised hypervisor approach first commercialised by the Xensource team aimed to present the guest operating system with the complete set of virtualised resources in the underlying physical system. Work coming out of the EU-funded EUROSERVER project has come up with a novel alternative called the microvisor[134]. This is a full Type-1 hypervisor but with a reduced footprint designed to operate efficiently on ARM chips. Rather than control each node's resources with an individual controller, resources are pooled across nodes and presented to each node with minimal overhead. This allows a storage device to be mapped almost directly onto the underlying Ethernet interface, providing significant performance improvements and reducing energy consumption. Recently, researchers in the Computer Laboratory have also started to look at using efficient, specialised micro-kernels to replace the full software stack that usually runs on top of the hypervisor[95]. These "unikernels" are extremey efficient and can boot in a fraction of the time taken by a full image. This work has been spun out and is now part of Docker.

2.4 Data centre traffic measurement

Traffic characteristics and workloads are a fundamental part of understanding how any network performs. However, data centres are hard to investigate because most operators view the operational details of their data centres as commercially sensitive, due to the

limitations of packet capture at high speed and because the scale of data centres means any dataset that is collected rapidly becomes unmanageable.

Despite this, there have been a few studies of data centre networks. One of the most useful of these comes from Microsoft Research and the University of Wisconsin[17]. This study obtained measurements from a number of different sources. This includes full packet traces topology and SNMP data for three university data centres, SNMP data and topologies for two private data centres and SNMP data from five commercial data centres. Their key finding was that there seems to be an On-Off pattern to most data centre packet traces, but the actual parameters of the distribution are hard to define. Other key findings include the fact that many applications send frequent small (200 byte) keep-alive packets, over 80% of the flows are less than 10kB long and that there is no correlation between links that are identified as hotspots (greater than 70% utilisation) and loss.

An earlier paper from the same researchers at Microsoft Research showed that it was possible to identify the type of workload from the flow-level traces[81]. Their traces clearly show two types of traffic that they call "work seeks bandwidth" and "scatter-gather". These represent OLDI and MapReduce respectively. They are so named because in the first case the work seems to be distributed to maximise the bandwidth received by aiming to keep flows rack-local and in the second case the work is distributed across the data centre. A number of other papers include limited measurement data to back up their conclusions[52, 5, 59]. Of these the most significant is the DCTCP paper [5] which shows results for a single pod of a production data centre (believed to be used for the Microsoft search engine, Bing). These results largely support the view that data centres carry a mix of traffic including short messages, multi-packet flows and long running background traffic.

In 2015, Facebook published the results of an analysis of traffic within and across their data centres [141]. Their results to some extent contradict previous studies, in large part because their traffic mix includes large volumes of cache traffic that spans across multiple data centres. Their key findings were:

1. Traffic does not exhibit particularly strong rack locality ("work seeks bandwidth") but nor is it all-to-all ("scatter-gather"). Instead it is a hybrid of these. However they did find that the locality patterns were stable across extremely long time periods (up to days).

2. Although many flows are very long-lived, these flows do not transfer significant volumes of bytes. Also the presence of load-balancing spreads the load from heavy hitters across the network. This means the set of heavy hitters changes rapidly and their long-term flow size is not far above the median.

3. Most packets are tiny (median length of less than 200 bytes)—this agrees with the observation of Benson *et al.* in [17]. However, unlike in that paper, the Facebook traces do not exhibit On-Off behaviour.

4. Hosts often communicate with hundreds of other hosts concurrently. However, most of the traffic is destined to hosts in a handful of other racks.

2.5 Data centre storage approaches

The rapid increase in scale and capacity of data centres has brought a renewed focus on high-performance storage systems. Data centre storage covers everything from dedicated Storage Area Network (SAN) systems like the appliances provided by NetApp to in-memory key-value storage systems like memcached [43]. Data centre storage systems have to provide a range of different services often with conflicting demands on the underlying storage system. The key metrics of interest are latency, reliability and block size. Customer-facing applications like web search or webmail[4] need a combination of fast retrieval for indexing and searching along with a reasonable degree of replication for reliability. Backup tasks need to store things like log files and possibly disk images. These may need to be stored reliably, although latency is not a key issue. Maintenance tasks need to access disk images to allow failed nodes to recover quickly or to bring a new virtual machine online quickly.

Storage-related traffic can make up a significant proportion of the total traffic crossing the data centre network. Consequently, it has a direct impact on the performance of the network. As with some OLDI applications, storage traffic can be highly asymmetric in its impact—a 40 byte `Read` query may trigger a response that is several Mbytes or more.

Many modern storage systems are built with commodity hardware and TCP/IP networking to save costs. One issue here is that scheduling storage resources is far harder than scheduling simple network resources since you have to take account unpredictable factors such as disk seek time.

Data centres bring new challenges to the design and operation of storage systems. Several conflicting requirements need to be met at the same time, including scalability, data integrity [60, 9] and resilience, consistency and line-speed performance. Often, the only cost-effective solution is to relax some of the requirements. Traditional client-server network storage systems like NFS [123], NBD [20] and DRBD [40] have been largely succeeded by distributed ones where functionality is distributed across multiple nodes in the network. In some cases, like PVFS [25], OCFS [117], Lustre [145] and Google FS [51], metadata servers are used to resolve the location of data and help maintain an updated view of the

[4]Gmail stores a customer's most recent few emails on fast SSD storage, but older emails that are accessed less frequently will be stored on slower-access storage such as spinning media.

storage resources so that consistency and data resilience is preserved in case of failures. Other systems, like Ceph [164], DHTbd [121], Flat Datacenter Storage [107], GPFS [143], FAB [142] and Panasas [165], distribute said functionality to multiple nodes or even across *all* storage nodes in order to support decentralisation and ease of management.

A storage system consists of some underlying physical storage media overlaid with a block device which divides the storage up into logical blocks for writing. On top of this will sit some form of filesystem which will give the user access to the actual data. The following subsections explain these in more detail.

2.5.1 Physical storage

There are three main architectures for physical storage within data centres. The first is where dedicated storage racks and controllers are presented to the servers as locally-attached storage. This is known as Storage Area Networking. The second is distributed storage where every server hosts one or more disks, but these can be accessed by processes running on other servers. The final architecture is in-memory where data is stored within volatile memory on local machines.

In the case of SAN and distributed storage, the physical data store may be a spinning disk or it may be solid state. These offer very different media access patterns and this has to be taken into account in the design of any storage system built on top of them. Some systems also use SSDs to provide caching of frequently accessed data and to prevent write-blocking of processes.

2.5.2 Block devices

Storage systems often rely on the concept of a block device which divides the underlying physical storage into discrete blocks to improve I/O efficiency by buffering I/O operations. This also serves to abstract the underlying physical device and thus provides a uniform view of the storage to the file system. Because of the buffering, the file system may believe a write has been made permanent when it is in fact buffered. There is a direct link between block size and I/O efficiency. Large blocks make it more efficient to read and write large files, but have a negative impact on small transactions. Equally if you make the block size too small large files are more likely to be fragmented, leading to inefficient reads and writes.

2.5.2.1 Virtual block devices

Virtual block devices allow physical storage to be assigned to virtual machines within a virtualised or paravirtualised environment. To the guest operating system the storage

appears to be local, but in fact it may be shared with many other virtual machines. Some such as the Xen Virtual Block Devices [13] and the virtio block device used by KVM are designed for use in paravirtualised environments. Others such as Google's Colossus [42], Amazon Elastic Block Store [175] and Blizzard [100] are specifically designed for use in data centres.

2.5.3 File systems

File systems are the traditional abstraction by which an operating system interacts with an underlying block store. They provide filenames and directory structures, metadata covering things such as date created, date last accessed and access permissions as well as resilience and the ability to track changes to files (in the case of journaling or transactional filesystems).

There are dozens of file system standards including the Extended File System family of file systems (ext, ext2, ext3 and ext4), Microsoft's FAT and NTFS and more exotic systems such as Reiser-FS and ZFS.

2.5.3.1 Object stores

While many data centres do use file systems they often use a different abstraction known as an object store. Instead of being able to write and read to and from anywhere in the file, an object store presents large blobs or tracts of data as a single object. This abstraction is well suited to the distributed nature of data centre storage. By treating the complete object as a single blob of data, tracking changes to objects becomes easier. This is particularly important where the storage offers redundancy with multiple copies of objects distributed across the data centre. Examples include Amazon's S3 (Simple Storage Service) and Microsoft's Flat Datacenter Storage.

2.5.4 Distributed storage protocols

The concept of accessing storage remotely across a network is not new. Distributed file systems go back as far as the 1960s when the Incompatible Timesharing System (ITS) [96] allowed file operations to be carried out on a remote machine over the ARPAnet as if the machine was local.

In the 1980s Sun created the Network File System or NFS [23]. This was the first widely-adopted filesystem to use the IP protocol and has gone on to become something of a standard for networked storage. More recently NFSv4 [148] allowed stateful transactions and NFSv4.1 [149] has added the concept of sessions, which makes it better suited to data centre environments. There are many other distributed storage systems such as GlusterFS

and the Windows DFS (distributed file system). These systems are designed to operate over wide area networks and are often built on top of TCP.

There have been several network storage protocols specifically designed for use in data centres. These include the Google File System [51] and more specialised protocols such as the Hadoop distributed file system (HDFS) [152], and Flat Datacenter Storage (FDS) [107].

FDS is designed to maximise storage throughput across a distributed blob store. It is specifically targeted at modern data centres with full bisection bandwidth. This allows it to be locality-oblivious and simplifies the design. Disks are also able to utilise their full bandwidth, removing the network as a bottleneck in the system. In FDS the storage is divided into blobs, each with a GUID (globally unique identifier). Data is written to the blob as sequentially numbered tracts, with the size of the tract used to optimise throughput (similar to how block size is used in block devices). Each physical device has a tractserver which processes read and write requests as they come over the network.

Rather than using a centralised metadata sever as used by HDFS [152] and GFS [51], FDS uses a distributed tract locator table at each blob. The TLT indexes the location of each tract. It consists of rows of data giving a version number and the location of all the blobs with replicas of that tract. Indexing into the table is done using the following function:

$$Tract\_Locator = (Hash(g) \ + i) \quad modulo(TLT\_Length) \tag{2.1}$$

The TLT is periodically refreshed across the whole data centre. If a blob goes offline for any reason it is removed from the relevant rows of the TLT, the version number is incremented and new blobs are added to the table to keep the correct replication level.

2.6 Simulation, emulation and testbeds

Many of the papers cited in this chapter rely on simulation or emulation to justify their conclusions. This is because the scale and complexity of data centres makes them prohibitively expensive to perform real experiments on. The need to model complex systems is not new. Simple systems can be modelled mathematically, but as you add complexity the models become increasingly hard to solve and eventually become impossible. This is where simulation, emulation and testbeds come in.

2.6.1 Simulation

Simulations attempt to model a complex system by abstracting a number of the complexities of the real world system. Simulation is a powerful tool because it allows you to replicate and repeat results, but it has definite limits [122]. There are two common

approaches to simulating computer systems [77]. Discrete event simulation tracks every state change in the system and can very accurately model complex interactions. Typically it simplifies or abstracts away things like the underlying physical link (instead modelling it as a statistical function). Traffic matrixes may be produced from a statistical distribution or may come from an actual traffic trace. However, the number of events in a complex system increases rapidly as the system grows and hence eventually you reach a limit of scale. For really large systems you have to use fluid model simulation. This models the state of the system as a series of fluid flows and statistical models of how they interact. The trouble is such fluid-flow techniques are bad at modelling highly dynamic systems which severely limits their utility for data centre simulations (as data centres seldom exhibit steady state behaviour).

There are a number of well known discrete event simulation frameworks. Probably the most widely used of these are ns2, ns3, Opnet Modeller and Omnet++. All of these were designed with different optimisations in mind.

- **Ns2** [109] has a long association with protocol work done in the Transport Area of the IETF. Consequently it aims to accurately model as many IETF transport protocols as it can. It is a very mature framework and is currently on version 2.35. Because of this maturity it has been used in a number of research papers and there are models available for several key data centre transport protocols. It has several known constraints. However its widespread use means that results are able to be compared with previous work.

- **Ns3** [110] was intended as a replacement for ns2. However, its codebase shares next to nothing in common with ns2. Work on ns3 has been largely driven by the wireless sensor network community and so it seeks to accurately model physical and datalink layers, in particular wireless networks. As a result it includes remarkably accurate layer 2 models which are notably absent in ns2. However, it has a far more limited array of transport protocols available, and very few researchers have used it for data centre networks research.

- **Opnet Modeller** [116] is a commercial framework that is only available to institutions that pay a license fee. Its main aim is to allow network operators and managers to explore "What if...?" within their existing networks. It has extremely accurate models of many real world devices and models the full network stack. However it is hard to model novel protocols and it lacks the scalability of ns2 and ns3. It partly makes up for this by allowing background traffic to be modelled as fluid flows, but this sacrifices accuracy and only helps where the background traffic is steady and predictable.

- **Omnet++** [111] is a little different in that it is a general network simulation framework that can be used for a number of applications including sensor networks, wire-

less ad-hoc networks, photonic networks as well as for Internet protocols. It does have models for physical networks and the TCP-IP stack, but it is hardly ever used in published research papers.

2.6.2 Data centre scale simulation

Over the course of my PhD I have developed a large scale simulation testbed in ns2. This testbed is capable of simulating both a classical three-tier data centre and a more modern full bisection bandwidth fat-tree topology. Details of these topologies can be found above in Section 2.1. The simulation can scale to several thousand nodes connected at 10Gbps. This pushes ns2 well beyond its normal operating range and has required me to optimise the simulator in several ways. Firstly, I have created my own data capture classes that are implemented directly within the C++ simulation code. This speeds up data capture and significantly reduces memory overhead. It also allows for accurate tracking of packet latency. Secondly, I have added code that is able to track flow completion times (FCTs)—this is a much more relevant performance metric for many data centre applications than simple throughput. My modification accurately identifies the first packet in a new flow, notes how many bytes there are in that flow and then monitors the flow across the network to identify when the last bytes of the flow are successfully delivered to the application layer at the far end. This is quite a significant modification because of the complex nature of ns2's TCP models. Thirdly, I have optimised the code to reduce unnecessary overheads such as unwanted packet headers. Finally, I have adapted a simulation scripting framework written by Keon Jang, a co-author of mine on a paper at Microsoft Research (see Chapter 5). This framework simplifies and automates the task of creating complex simulations and topologies. I was able to expand it to simulate classic 3-tier topologies or more complext Fat-Tree topologies.

Before relying on my modified code I verified that my new data capture classes were workind as expected. For this I used small scale simulations and manually tracked the packets throughout the simulation. This allowed me to verify that the data I was collecting was indeed correct. I used a similar appoach to verify my FCT code, manually tracking flows and verifying that the flows had indeed completed in the time reported. This proved that the new classes were behaving as expected. The performance improvements were evident from the fact I was able to successfully increase the scale of the simulations by at least an order of magnitude.

To give a sense of the scale of the simulations produced by this system, one simulation of a 3 tier topology with some 3,000 nodes used 182GB of RAM before suffering a segmentation fault. Careful analysis of the segmentation fault revealed a fundamental underlying issue present in both ns2 and in the more modern ns3. In order to speed up simulations and reduce memory load both these simulators create a free packet pool. Initially the simulation creates a new packet every time one is needed. Once a packet is no longer

needed it is not deleted. Instead it is placed in the free packet pool. Next time a new packet is needed the simulator tries to use a packet from this pool. Each packet in the network has a unique identifier which is an unsigned 32 bit integer. Normally this process works well as there are many fewer than 2^{32} packets simultaneously in the network. However a simulation with 3,000 nodes all sending data at 10Gbps soon reaches the state where it needs more active packets than can be identified. This in turn leads to a segmentation fault as two events try to access the same packet. In theory, the solution is simple—just use a 64 bit integer for the packet ID. However, it turned out that this affected so much of the code base that it proved impossible to solve[5].

2.6.3 Emulation

An emulator attempts to replicate the actual working of a large complex system within a smaller system. In network protocol engineering this can be as simple as using Linux boxes to emulate routers in the Internet. The aim is to capture all the essential behaviours of the system without the complexity or expense of the specialised hardware. Usually there will be some trade-off such as speed, data throughput or application workload. The implications of these trade-offs must be understood as they may reduce the utility of the approach in certain scenarios.

A realistic emulation of a data centre requires the ability to emulate all the constituent parts of the system including the network and end-hosts on a much smaller scale system. Mininet and Mininet-HiFi [87, 61] create complex networks using a combination of an OpenFlow controller, one or more virtual switches and a large number of virtual machines emulating the hosts. Mininet trades bandwidth for scale with "links" typically constrained to 10s of Mbps. In contrast Selena [124] combines real hardware with Xen virtualisation and uses time dilation to allow the system to scale without sacrificing emulated bandwidth. The strength of both these techniques is they allow you to experiment with real-world applications and code. However they are still only able to scale to relatively small networks.

The Network Simulation Cradle (NSC)[6] and Direct Code Execution (DCE) are attempts to bridge the gap between simulation and emulation. NSC allows one to use real TCP/IP stack code in ns2 or ns3. This ensures that the simulation is accurately measuring end-host behaviour. However it is less good when you wish to simulate novel protocols. DCE is a plugin for ns3 that allows you to run native code and connect to the simulator as if it were a real network. Obviously, this allows accurate measurement of applications over

[5]It is worth noting here that ns2 source code has evolved over the last 20 years and now includes over 4,800 files totalling more than half a million lines of code written in a mix of C++, TCL and oTCL (an object-oriented version of TCL).

[6]See http://research.wand.net.nz/software/nsc.php (accessed February 2018)

a simulated network. However, it suffers from a lack of standard library support which severely limits its actual application.

2.6.4 Testbeds

Testbeds are the classic way to perform "real" experiments on networks. Most network and computing testbeds are centralised facilities. UCL's HEN (Heterogeneous Experimental Network) [69] is a good example. But recently, there has been a trend towards creating distributed testbeds. PlanetLab [127], OneLab [113] and GENI[50] are all examples of this. The main driver for this is economic — such systems allow research organisations to access a much larger-scale facility than they could afford directly. However, there is also a certain political element — both the EU and the National Science Foundation promote collaboration by targeting their research funding.

Chapter 3

Latency matters

This chapter will discuss why I and many other researchers have focussed on latency as the main metric for data centre transports. This serves as the main motivation for my thesis as well as motivating the three main contributions discussed later in this dissertation.

3.1 Controlling latency in the Internet

Within any network that follows the layering and end-to-end principle, the transport protocol, or more specifically the congestion and flow control protocol, is responsible for controlling transmission rate and hence latency. Typically Internet transport protocols such as TCP aim to increase the transmission rate in order to gain the highest throughput. This is because most "traditional" Internet applications such as world-wide-web or email are typically bulk-data applications (at the bottom left in figure 3.2). Van Jacobson's additive increase, multiplicative decrease (AIMD) congestion controller[74] worked well with low delay-bandwidth product networks where the aim was to maximise throughput for bulk data applications. However, such controllers cannot make use of high delay bandwidth networks and have the unfortunate side effect of causing queues to grow too long.

A combination of increasing access speeds, the advent of the so-called Web 2.0, the growth in social media and the astonishing rise in online streaming media have had an impact on congestion controller design. FastTCP [163], High-Speed TCP [44], BIC and CUBIC [57] are all designed to maximise throughput in high delay-bandwidth product networks. Compound TCP [155] combines a traditional AIMD style controller with an equation-based controller similar to CUBIC. LEDBAT [139] (and its close cousin μTP [158]) are designed to be sensitive to queue delay and are designed to try and avoid unnecessary queue build-up. There has also been renewed interest in Active Queuing mechanisms (AQMs) with CoDel [106] and PIE [119] offering new alternatives to RED (random early discard) [45].

Explicit Congestion Notification [137] has also become much more widely available, with most commodity switches now implementing it by default.

Despite these advances, latency is still relatively unpredictable in the Internet at large and is dominated by the propagation delay and by access network bottlenecks. Content owners seek to alleviate this by moving content closer to the end-user. Companies such as Google and Facebook set up data centres across the world, using load balancers to share load between them, seeking to balance the load on the data centre with the delay caused by longer transmission distances. Content delivery networks (CDNs) peer content in access providers' networks for the same reason. There have even been suggestions to co-locate data at access network line devices such as the BRAS (broadband remote-access server).

3.2 Latency in the data centre

Message latency in a data centre is made up of two parts. The time taken to transmit that number of bytes plus the end-to-end delay of the network path. More formally it can be given by the following equation:

$$L = D + T \tag{3.1}$$

$$\begin{aligned} \textit{where} \quad & L = \textit{the overall latency } (\textit{seconds}) \\ & D = \textit{the maximum per-packet end-to-end delay } (\textit{seconds}) \\ & T = \textit{the time to transfer the message } (\textit{seconds}) \end{aligned}$$

and

$$T = (M \times 8)/B \tag{3.2}$$

$$\begin{aligned} \textit{where} \quad & M = \textit{the message size in bytes} \\ & B = \textit{the available bandwidth } (\textit{bits per second}) \end{aligned}$$

Note that in the above equation the available bandwidth may be much lower than the theoretical bandwidth because either the flow is application/protocol limited or is being rate limited by the network. By contrast with the Internet, in a data centre the transmission distances are so short that the delay component is not dominated by the propagation delay—even in a large data centre the maximum wiring run might be of the order of 200m, giving a maximum propagation delay of about 1 microsecond. Consequently what matters is any queuing and serialisaton delays.

The latency experienced by a single packet flow is dominated by the end-to-end delay which means D in equation 3.1 dominates the latency. Such flows can be described as delay-sensitive. By contrast the latency of a long flow is dominated by the time to transmit the data, this in turn is dominated by the available bandwidth (B in equation 3.2) and hence they are bandwidth-sensitive. Between these extremes lie flows with short numbers of packets such as those created by OLDI applications and by the shuffle stage of map-reduce. Of course the available bandwidth is itself in part dependent on the delay and the mix of traffic being sent.

3.2.1 End-to-end delay

Most modern data centres employ some form of virtualisation. By virtualising your servers you gain flexibility, improve your robustness to failure and can abstract away the underlying hardware. The main virtualisation approaches are discussed briefly in Chapter 2. While this virtualisation offers many positive benefits like the ability to hot migrate workloads and to have "hot spare" servers that can take up the load seamlessly, it does come with a latency cost. In order to virtualise the physical resources of a server, most hypervisors operate a control domain which is responsible for passing the hardware resources through to the virtual machine running on top. The drawback to this is that it increases the data path which in turn increases the latency. However, this can be solved by ensuring you do not have VM over-subscription and using techniques such as vSched [91] and vSlicer [171] which can achieve low VM scheduling delay even with CPU oversubscription. So for this discussion I assume a virtualised environment with no over-subscription of resources (e.g. there is no more than one VM per CPU core).

Sources of end-host delay come from dividing the data stream into TCP segments, encapsulating these within IP packets, inserting these packets into Ethernet frames and then serialising these onto the physical wire. Modern NICs have been designed to take as much of the processing load off the server in order to reduce the delay to a minimum. Techniques include Large Segment Offload (LSO), which allows the end host to send large batches of data to the NIC which then becomes responsible for segmenting and encapsulating them as well as TCP checksum offload which allows all checksumming to be done in hardware on the NIC.

However, in a virtualised system the hypervisor's NIC driver becomes responsible for sending data from a number of VMs down into the physical NIC. This driver should maintain one queue for each hardware queue on the physical NIC. So if the NIC has a fewer queues than VMs, a small message can get queued behind large messages from other VMs. This also implies that the use of batching techniques like LSO might exacerbate delays by allowing VMs to send large batches of packets to the driver. During my internship with Microsoft Research we conducted a simple experiment to measure this.

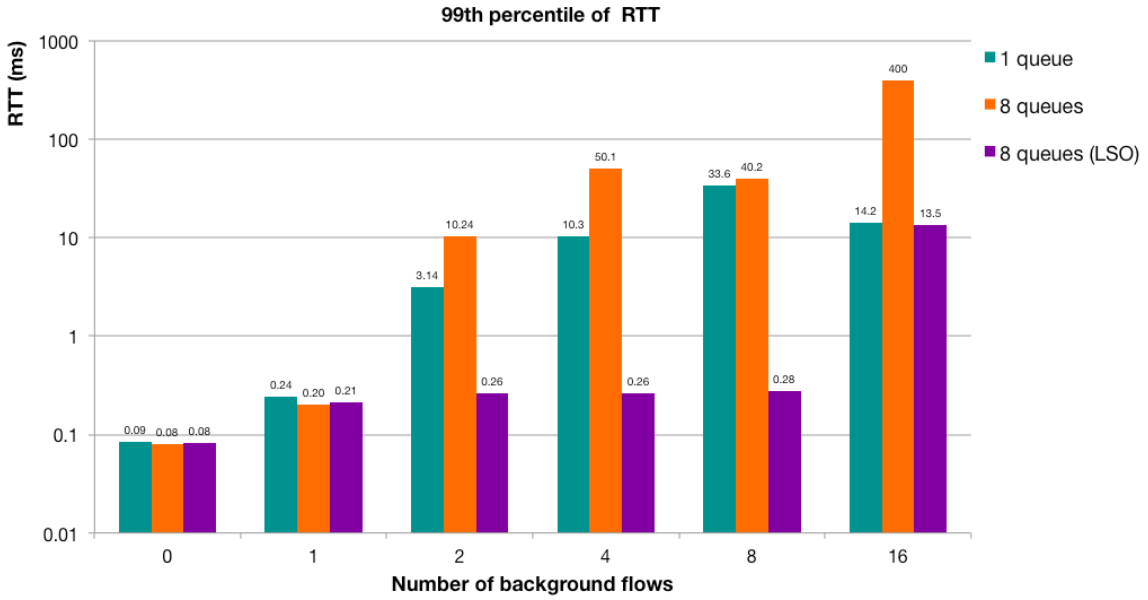

Figure 3.1: 99th-percentile of the RTT between two hosts (vswitch to vswitch) with varying number of background flows

Two physical servers are connected via a switch using 10Gbps Ethernet. Each server has two hexa-core Intel E5-2665 CPUs running at 2.66GHz. The servers run the Microsoft Windows Hyper-V hypervisor. A simple "ping pong" application continuosuly sends 1kB messages back and forth between the servers. Different runs were performed with different settings. Each run lasts for seceral minutes (generating extremely large numbers of flows). Figure 3.1 shows the 99th percentile of the round trip time. With LSO enabled, and with a single queue, the application has a RTT of 100 µs in the absence of competing flows. As the number of background flows increases, this RTT grows beyond 10ms. Turning off LSO has a pronounced impact on the RTT in the presence of background traffic, with RTTs growing to more than 100 ms in the worst case. This suggests that LSO is essential to reduce RTTs. Finally, the number of queues is increased to 8. Here the RTT stays stable at about 400 µs until the number of flows exceeds the number of queues on the NIC, at which point it goes up dramatically.

In summary, as long as there are sufficient queues available on the NIC, and as long as the driver can access those queues, the end-host latency is stable and is bounded O(100 µs). This tallies with the intra-rack average RTT of 100 µs observed by the authors of the DCTCP paper in the absence of any queues (see section 2.3.3 of [5]).

3.2.2 Queuing delay

In a data centre network, queuing delay can quickly dominate the overall delay. Most data centres adopt one of two topologies—either a traditional hierarchical topology with racks of servers arranged into pods that are connected with a core network or a fat-tree style

topology [3]. In both cases workload locality will have a significant impact on the number of switches a given flow has to traverse. In the best case a flow in a traditional topology has to traverse a single top-of-rack (ToR) switch. A modern data centre ToR switch such as a Cisco 4900 series has 16MB of buffer shared between 48 ports. Even assuming the buffers are assigned equally to each port that gives 350kB at each port. At a line speed of 10Gbps that equates to 280 µs of delay. In practise unless all ports are fully loaded each port could have access to over 1MB of buffer which pushes this delay up to more than 1ms. In a fat-tree topology a flow may traverse three or even five switches. Thus, in the worst case, a packet could see cumulative queuing delays of several milliseconds. The measurements in the DCTCP paper support this observation. Figure 9 in their paper shows they saw delays of up to 14ms for some packets [5].

3.3 Understanding the requirements of data centre traffic

Much research on data centre transport protocols has been motivated by the results of a limited number of small traffic studies as described in Section 2.4. These studies suggest that data centre traffic broadly splits into two groups, short foreground message traffic and longer-running background traffic. It is important to note that these studies specifically focus on single tenant data centres. Such data centres are used by online companies to host search engines, social media sites and online retail sites. They may also be used by large corporations to host their internal systems.

Data centre traffic generally comes from three types of application:

Partition-aggregate applications: These applications send small queries out to a large number of nodes (the partition stage). Each query then generates a small response, often only a single packet long. These arrive at the aggregation node where they are combined to give the result. The efficiency is directly related to the time taken to retrieve all the responses from the partition stage.

Online data-intensive (OLDI) applications: These include web applications such as search engines and e-commerce where the results of a transaction have to be sent to an end user. These applications generate short flows of messages that may be a few tens of packets long. A study conducted in 2009 suggested that there is a direct correlation between increase in latency and a reduction in revenue. Consequently OLDI applications have strict deadlines by which their results must be passed on to the user.

Long-running applications: These include applcations transferring data sets to and from storage as well as maintenance related tasks such as creating backups or replicas

and installing new software images. As a general rule it has been assumed that such traffic is not latency sensitive. However, sometimes this storage traffic may be seen by the end-user, leading to an increased use of SSD storage in many data centres for so-called "hot" storage. Examples of this might include the first page of emails in someone's online email account and the content for adverts that appear on search engines and social media pages.

The authors of data center TCP (DCTCP) studied the traffic generated by a 6,000 node production cluster over a one month period [5]. This cluster is dedicated to partition-aggregate style traffic. In common with most data centre traffic studies, most of the logs were at the granularity of sockets or flows, although they did collect some packet and application level logs to extract latency information. They identified two of the above classes of traffic which they term *query traffic* and *background traffic*. They further sub-divide the background traffic into *update flows* which are used to refresh the data at each worker node and *short message flows* which are used for controlling the cluster.

The traffic pattern generated by this combination of traffic sources is far from simple. This is borne out by other traffic studies which have tried (and largely failed) to characterise data centre traffic mathematically.

3.3.1 The importance of low latency

Figure 3.2 attempts to show how different applications are sensitive to bandwidth, to delay or to both. As can be seen, data centre applications tend to be sensitive to both, and OLDI or partition-aggregate applications are especially sensitive to latency.

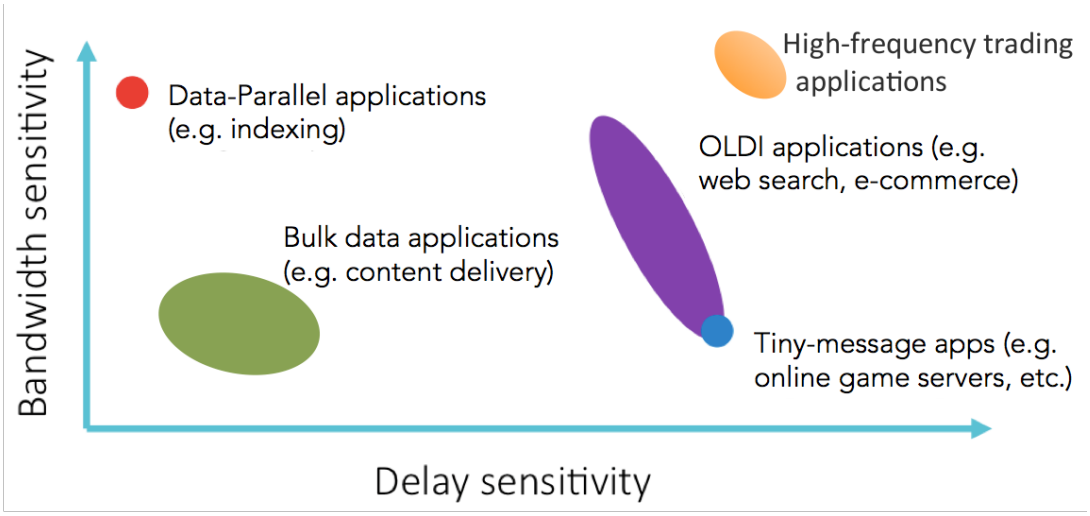

Figure 3.2: Comparing the latency requirements of different traffic types

In the case of partition-aggregate applications, delaying messages exacerbates the problem of "stragglers", or partition nodes that return their results late, leading to an overall delay

in the aggregation stage. While some partition-aggregate applications can choose to ignore results that arrive late, others may be highly sensitive. Imagine for instance searching a distributed key-value store for a key that only occurs once. In this case you may have to wait until every query result returns before finding the value you want. As mentioned above, a response message is generally only a single small packet and so this traffic is highly sensitive to per-packet latency. Partition-aggregate is also especially sensitive to TCP incast [27] where the arrival rate of packets causes the buffers at a single node to overflow, leading to packet losses across multiple flows and triggering time outs.

OLDI traffic is characterised by short message flows that may consist of tens of packets. The entire message is critical and so this traffic is sensitive to what might be termed "message latency" or "flow completion time"—the total time taken to receive the entire message. Often this latency requirement is imposed externally. Take for instance a web search engine. When the user submits a search request they have come to expect to get the response within less than a second. In that time the entire index has to have been searched (using some form of partition-aggregate scheme), the search results have to have been ranked and if the search engine uses an advertising revenue model a relevant set of adverts must have been chosen (which may even have involved an instant auction). In order to hit the final deadline, each stage of the process has a strict deadline. Search results that return late are simply ignored. However, the success of a search engine is measured by the accuracy and relevance of the results it returns to the user and so it pays to not ignore too many partition results. Such stragglers also represent a waste of compute resource and reduce the overall efficiency of the data centre.

3.4 Controlling latency

In any network, latency control can be imposed by the network hardware using QoS mechanisms such as pacing, priority queuing, etc., or it can be controlled by the end systems through careful flow and congestion control (both the responsibility of the transport protocol). In this section, I explore how this has been done in the wider Internet before looking in detail at some of the approaches suggested for data centre networks.

3.4.1 Transport protocols in the Internet

Internet Transport protocols have been researched extensively over several decades, but in nearly all cases the aim has been to increase the data throughput of the network. This is because most classic Internet applications such as world-wide-web or email are inherently bulk data applications. This has led to numerous improvements to the TCP congestion controller. In 1988 Van Jacobson introduced the concept of the Additive Increase, Multiplicative Decrease (AIMD) controller [75]. In 1990 the fast retransmit and

fast recovery algorithms were proposed [76], later standardised as RFC2001 [156]. These were designed to react to congestion more promptly and to prevent the congestion window from stalling during retransmission. As network speeds began to increase in the late 1990s and early 2000s it became clear that unmodified AIMD protocols do not perform well in high delay-bandwidth product networks because slow start takes too long to reach steady state. A number of alternatives were suggested including FastTCP [163], High Speed TCP [44], BIC [172] and CUBIC [57].

Various approaches were also suggested to allow end hosts to react to congestion more quickly, thus reducing queue build up and improving goodput[1] across the network. Active queue mechanisms such as RED [45], CoDel [106] and PIE [119] are designed to start signalling congestion before a queue has grown too large. Explicit Congestion Notification (ECN) [137] allows the network to signal congestion using an explicit flag rather than by dropping packets. Delay sensitive congestion controllers such as TCP Vegas [19] and LEDBAT [139] use delay variation as an indicator that queues are building on the path and hence congestion is increasing. There has also been work on transport protocols for real time traffic. The Real-time Transport Protocol (RTP) [144] is a UDP-like transport controlled by the related Real-time Transport Control Protocol (RTCP). Pre-Congestion Notification (PCN) [99] is an admission control mechanism that combines virtual queues at switches with marks in the ECN field of the IP packet header [101] to indicate whether a network is becoming congested and hence whether any new traffic can be allowed to enter.

3.4.2 Transport protocols in data centres

As noted in Section 3.2 above, data centre latency for single packet flows is entirely dominated by queuing delay, whilst the latency for longer flows depends more on the average transmission rate achieved by the flow. This has led a number of researchers to focus on reducing queue lengths in order to reduce latency for short flows. Other than reducing the physical size of the queue, the only way to reduce queue length is to ensure that the combined traffic rate at any given queue is the same as or less than the service rate of the queue. There are three broad approaches that can be used:

1. **In-Network approaches.** Active Queue Management or AQM involves starting to drop packets before the queue has grown. This in turn signals to the end-hosts to slow down their transmission rate. There are several possible mechanisms for this.

    - Random Early Discard (RED) [45] and gentle RED monitor the average queue length over time. If this exceeds a lower threshold the probability of dropping

[1] Goodput measures the fraction of bytes transmitted that are actually delivered to the application. It is a standard measure of the efficiency of any transport protocol.

an arriving packet starts to grow. Once it exceeds an upper threshold all packets are dropped.

- Controlled Delay (CoDel) [106] is a simpler mechanism that measures the minimum queuing delay experienced by packets over a time period. If it exceeds 5ms then a packet is dropped and the interval is incrementally reduced. This continues until the minimum delay drops below 5ms again. FQ-CoDel is similar, applying CoDel to the individual queues within a weighted round robin stochastic fair queue.
- PIE (Proportional Integral Controller Enhanced) [119] uses the smoothed average of the queue drain rate as an estimator for the average queue delay. This is used to calculate a drop probability, with the probability re-calculated every 30ms.

2. **Hybrid approaches.** These approaches require the cooperation of the end system.

   - Congestion marking (using ECN) is a common-sense extension to AQM. If packets are being dropped before a queue has built up then logically it would be better not to drop them but to still signal the end-hosts to slow down. This is achieved by setting the "Congestion Experienced" or CE codepoint in the IP header. Because it relies on senders and receivers being correctly configured and responding correctly, ECN has struggled to find traction in the wider Internet. However, Congestion Exposure (ConEx) [32] provides a solution to this by forcing senders and receivers to declare how many CE marks they expect to see. Within a controlled environment such as a single tenant data centre, ECN is easy to deploy.
   - DiffServ-like mechanisms involve defining traffic classes and assigning packets to those classes at the network edge. The network then applies traffic management mechanisms at each switch depending on the class of traffic. Typically classes might include normal traffic that receives no special treatment, expedited traffic that receives increased priority in the queue, and scavenger traffic that is preferentially dropped at the queue.

3. **Network edge approaches.** These approaches include IntServ-like policer mechanisms that seek to rate-limit traffic entering the network.

   - Admission control systems like pre-congestion notification (PCN) [99] are generally designed for real-time traffic. PCN works by using a "virtual" queue to measure incipient congestion in the network. When this virtual queue exceeds a threshold, packets are marked and the admission controller uses the rate of these marks to make admission decisions. A similar approach could be used for controlling data centre traffic.

- Traffic pacing at end hosts or switches allows you to strictly limit the rate each flow or end-user receives. Later, in Chapter 5, I describe Silo, a system that combines traffic pacing with a network calculus-based workload placement to ensure that queues can never build beyond a known limit.

No single approach alone can achieve the required low latency performance, so current proposals such as DCTCP [5], HULL [7], qjump [53] and Silo (see Chapter 5) combine these techniques.

3.4.2.1 Additional requirements

Traditionally, there has been a clear divide between inter-process communications (which happened on a single piece of silicon), data transfers (which happened between processes on different physical machines) and storage (where data was transferred between memory and non-volatile storage). Modern data centres have blurred these boundaries. Applications may require IPC between different physical machines leading to solutions such as remote DMA (rDMA) [37]. Virtualisation means that apparently remote processes may be running on the same machine or even the same piece of silicon. Fault tolerance and migration means that processes may move physical location while an application is running and storage may be widely distributed across the data centre. All this means a data centre transport protocol should be able to:

- Provide predictable flow completion times.
- Be resilient in the face of failure or migration. While the network layer copes reasonably well with disruption to paths, transport protocols like TCP may well fail if the end host migrates.
- Control congestion at both layers 2 and 3 of the stack. This is important as it leads to smaller buffers, lower delays, less jitter and even to lower energy use.
- Minimise complexity within the physical network. One of the features that distinguishes the modern data centre from the high performance computers that went before it is the reliance on simple commodity hardware. There are also interesting constraints on things like the complexity of wiring needed[2].
- Reduce the impact of incast.

In addition to the above requirements there are some other desirable behaviours for any data centre transport.

[2]Complex wiring is both expensive and difficult to maintain. Furthermore, it can actually have a marked negative impact on the energy efficiency of a data centre by disrupting airflow.

- Efficiency (both computationally and in energy terms), for instance the ability to migrate traffic away from under-utilised paths to enable network equipment to save energy by "sleeping".
- The ability to utilise multiple paths where these exist. As described in Section 2.1, data centre topologies are becoming flatter and more diverse.

3.5 DCTCP—the current best-of-breed?

The authors of DCTCP claim that it meets all the requirements for an optimal data centre transport protocol. It is designed to keep queues short by aggressive use of ECN marking. Short flows are favoured over long running flows, but long running flows still get reasonable throughput. Incast is reduced by adoptimg a shorter *minRTO* (as recommended in [159]). Finally, it is designed to work with commodity hardware simply by re-configuring the RED parameters at the switches. As a result DCTCP has gained considerable traction in the community and has even been standardised by the IETF[16]. I have chosen to compare the performance of DCTCP and TCP because both these are used in production data centres, whereas HULL requires more extensive modifications and is not reported to be used in any real data centre.

When analysing DCTCP I noticed two things that might explain why it seems to perform so well. Firstly, DCTCP effectively allows short flows to avoid any congestion response at all. DCTCP's congestion response is controlled by a parameter α. The current value of α is set according to the EWMA of the proportion of ECN CE marks seen in the last window of data. α is then used to determine the new congestion window according to the formula:

$$cwnd \leftarrow cwnd \times (1 - \alpha/2) \tag{3.3}$$

Thus the congestion response varies between nothing (if α is 0) and halving (if α is 1). In the DCTCP paper, α is initially set to 0, thus guaranteeing that flows of less than 2 RTTs will never respond to congestion. Secondly, the results are based on an assumption that there is always a significant fraction of background traffic to absorb any congestion. Effectively the authors have designed an algorithm that is highly optimised for the specific workload they measured in their own study.

In order to understand the impact of these two factors, I performed a number of ns2 simulations as explained below. My aim was to find out the extent to which DCTCP has been optimised for a specific traffic matrix.

My simulations are all based on ns2 [109], more specifically ns2.35. It models idealised transport protocol behaviour and uses a simplified models of the underlying network consisting of a set of links with delays, bandwidth and error rates connected by queues.

While ns2 suffers from a number of issues it is still one of the most accurate network simulators available for simulating transport layer behaviour. It is also the simulator used by the DCTCP authors for the simulations in their paper. As a result it is the ideal choice for this set of experiments.

3.5.1 Modifying ns2

One of the weak points of ns2 is its tracing facilities. The built-in trace helpers are optimised for tracing individual packets or queues. As explained above, for data centres what matters is not the individual packet latency but the latency of the whole flow. As explained in section 2.6.2, I have modified the TCP models in ns2 so that you can trace the flow completion time (FCT).

Having added my flow completion time code I also had to extensively modify the traffic generator model in order to generate appropriate flows. My traffic generator is designed to model a large number of different traffic types that have been identified in data centres. It takes a number of parameters: If `size_ = 0` then it generates flows according to a Pareto distribution of shape `shape_` and mean flow size `mean_`. Otherwise, if `size_` is set to a positive integer then it will generate flows of exactly that size. Pareto was chosen to reflect the observation that Data Centre traffic in the wild exhibits an On-Off pattern[17]. This also allows me to roughly model the flow distributions observed in [17] and [5].

The inter-arrival time of the flows is controlled by `interval_`. If `interval_` is set to zero then it draws arrival times from a built in distribution (generated from the interarrival times of a telnet application—this is similar to the activity from a typical end user). Otherwise if `interval_` is set to a positive number then the inter-arrival times are drawn from an exponential distribution with mean set to `interval_`. Accurately modelling inter-arrival times in a data centre is infeasible, but the measurements shown in Figure 3 of the DCTCP paper[5] show clear exponential characteristics.

The actual code needed for DCTCP was copied from the code released by the DCTCP authors[3].

3.5.2 Microbenchmarks

I used a simple microbenchmark simulation to compare TCP new Reno with DCTCP. These simulations used a simple bottleneck topology with a number of sources sending data to a single destination. This is designed to replicate the aggregation phase of partition-aggregate and is known to be liable to trigger incast.

[3]Downloaded from `http://simula.stanford.edu/~alizade/Site/DCTCP\_files/dctcp-ns2-rev1.0.tar.gz` (accessed February 2018).

Figure 3.3 shows the simulation set up. All links have a bandwidth of 10Gbps and delays as indicated. The bottleneck queue has a length of 250 (for the TCP simulations). For the DCTCP and RED simulations the RED parameters are set according to those given in the DCTCP paper.

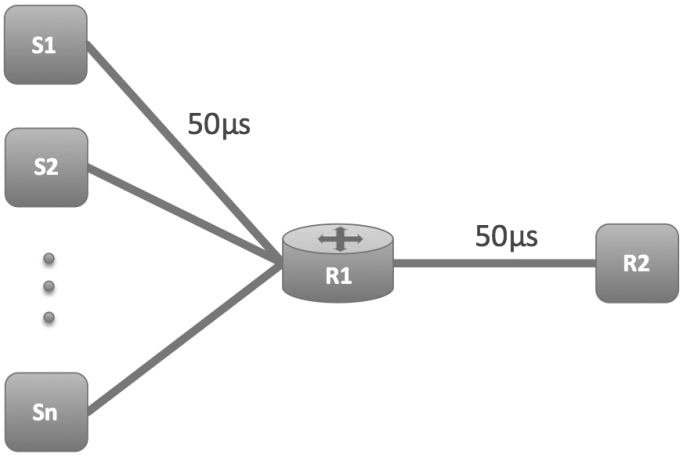

Figure 3.3: The simulation setup for the microbenchmarks

Two sets of simulations were performed. The first with the same traffic model as that used in the HULL paper [7] with packet sizes generated according to a Pareto distribution with mean 100kB and shape 1.05 and the second with a distribution designed to give a greater proportion of short flows (Pareto with mean 50kB and shape 1.20)[4]. The inter-arrival times were exponential with a mean of 10ms between new flow requests (obtained by curve fitting to figure 3 of the DCTCP paper).

Each set of simulations was run with 10 and 20 sources. These reflect typical map sizes that are chosen for partition-aggregate tasks[5]. Each run was repeated 10 times with different seeds, and the results combined into a single set. This reduces the risk of artefacts in the simulation relating to the specific seed used. Figure 3.4 compares TCP with DCTCP for the longer tail distribution. The flow times are given relative to a normalised FCT (e.g. assuming the flow was transmitted at full line rate with no additional queueing delay). The authors of DCTCP claim that it reduces the FCT for short flows, while having minimal impact on longer flows. These results would seem to support that claim, though it is notable that the improvement is less clear with 20 sources. It is also interesting that for 10 sources DCTCP has more outliers than TCP.

Figure 3.5 shows the same results but for the shorter-tail distribution which is designed to stress DCTCP more. DCTCP was designed for a traffic matrix where there is sufficient

[4]The first distribution gives 75% short flows (less than 21kBytes) containing 45% of the bytes while the second gives 85% short flows containing 52% of the bytes.

[5]See `https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html#Mapper`, accessed February 2018.

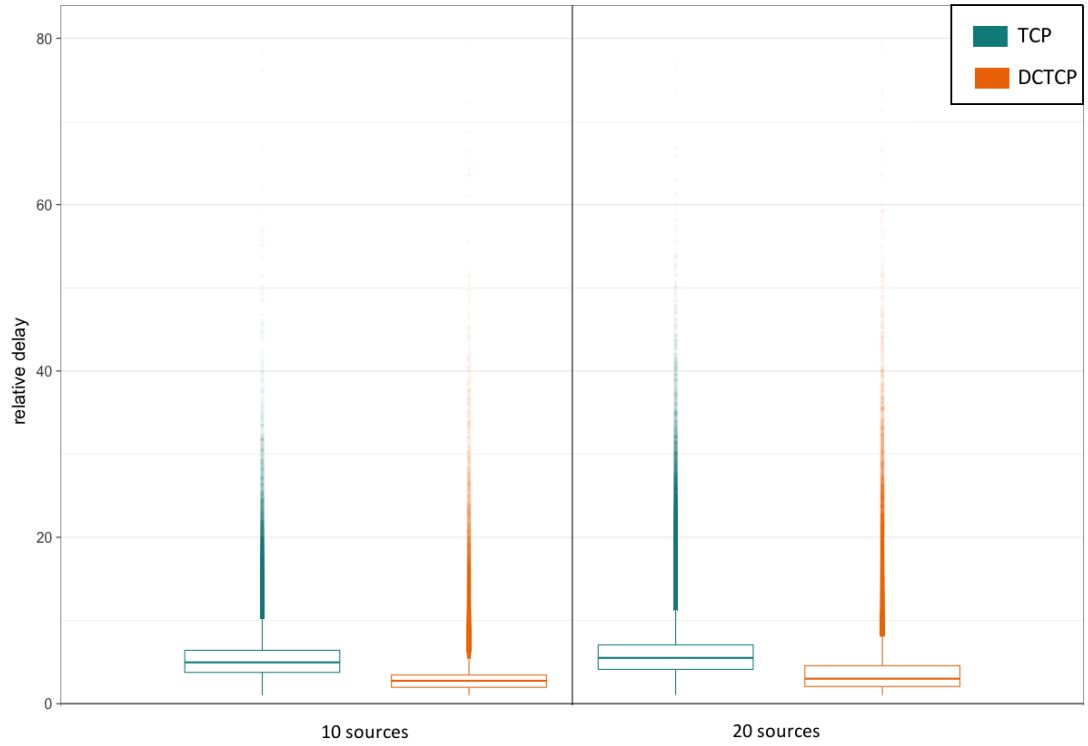

Figure 3.4: Comparing normalised FCT for TCP and DCTCP (long tail)

background traffic to absorb all the congestion. While the results for 10 sources are broadly similar to the ones with the longer distribution it is notable that with 20 sources, DCTCP gives many more outliers than TCP with at least one flow seeing an FCT 100x worse than it should be. I suspect this is because with such a large number of short flows, DCTCP's lack of initial congestion response means it is unable to respond to the increased congestion and ends up suffering from incast.

When I looked in detail at the relationship between the flow size and the relative flow completion time it became obvious that shorter flows were exhibiting far more variation. Figure 3.6 shows the results for 10 sources, comparing the relative FCTs for TCP and DCTCP against flow size. In all cases, DCTCP performs better than TCP on average. However, the fact that some of the short flows are seeing FCTs that are worse than those given by TCP is not ideal, especially given these results are designed to reflect the traffic matrix that DCTCP was designed for.

3.5.3 The impact of DCTCP's RED algorithm

As indicated in Section 3.5, DCTCP uses two approaches to improve the latency of short flows. Firstly flows initially do not respond to congestion. This clearly favours short flows but this lack of initial congestion response causes issues in a highly congested network as the results in Figure 3.5 indicate. Secondly it uses a modified version of the RED

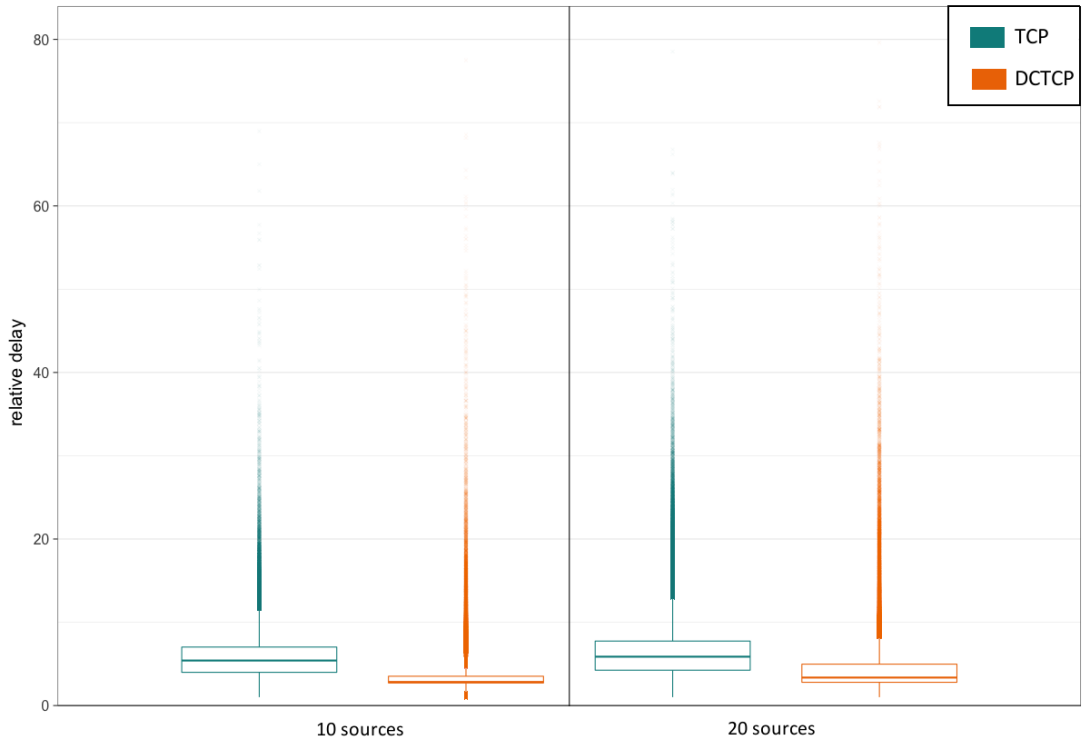

Figure 3.5: Comparing normalised FCT for TCP and DCTCP (short tail)

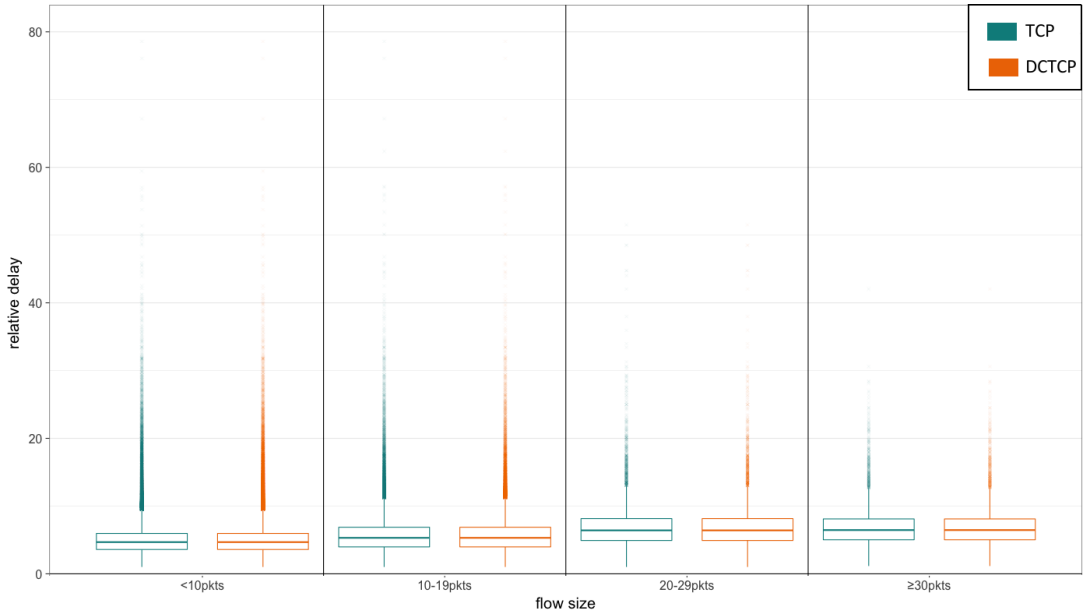

Figure 3.6: Comparing normalised FCT against flow size for TCP and DCTCP (long tail)

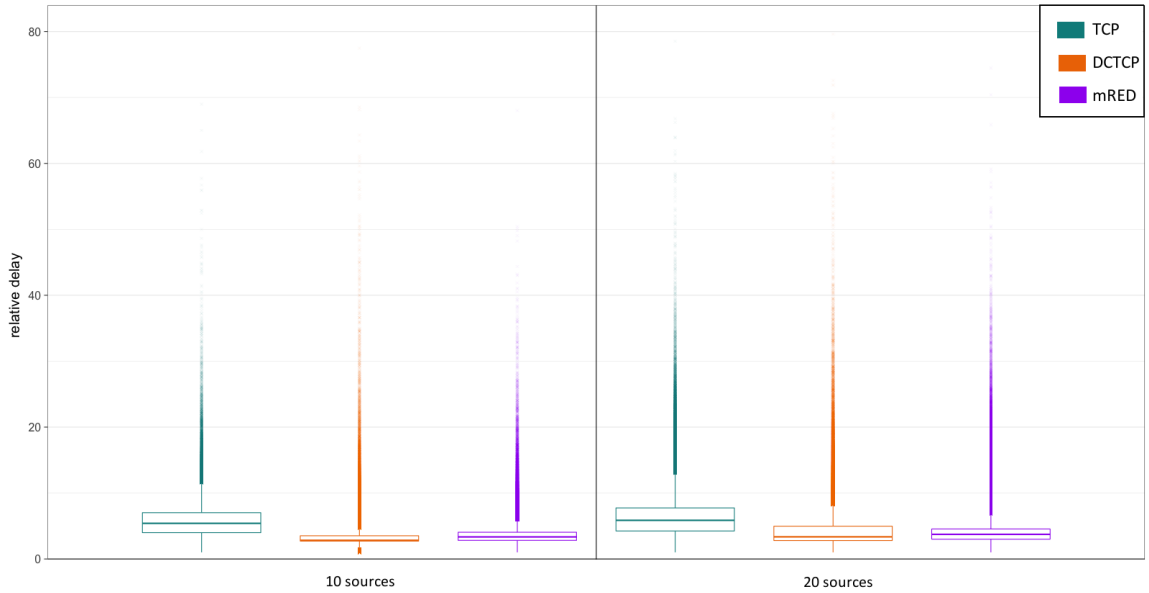

Figure 3.7: Comparing the impact of a modified RED on normalised FCT

algorithm that removes the averaging function from the standard RED algorithm [45] and instead CE marks all packets once they cross a given threshold. I decided to test how well this algorithm behaves at controlling latency when used without DCTCP's aggressive congestion response. I repeated the microbenchmarks as above but using a modified RED queue at the bottleneck with TCP new Reno as the transport.

Figure 3.7 compares this modified RED with TCP and DCTCP for the shorter tailed distribution. It is striking that this simple AQM mechanism, coupled with standard TCP, performs almost as well as DCTCP. The median FCT is higher, and for the 10-source run, the IQR is a bit wider. However, in both scenarios, the number and scale of outliers is much reduced with the modified RED. This highlights the importance of congestion control—failing to respond to congestion only works as an approach if other flows are responding in your stead. This is also significant as it can work without the need to modify the end-host transport protocol.

3.6 Conclusions

In this chapter I have shown why latency is the key performance criteria for data centre applications. I explored the sources of latency within any computer network and showed how these impact data centre networks. I contrasted the different approaches for controlling latency in the Internet and data centres. I also showed how DCTCP performs less well at the long tail than TCP or TCP with a simple modified RED algorithm.

As a result of my activity in this field, I became involved in a new IRTF Research Group looking at Data Centre Latency Control [72] which sought to gain access to traffic traces

from major data centre operators as well as trying to gain a better understanding of the various transport and network protocols that have been proposed. Sadly the attempt was abortive as operators such as Google, FaceBook and Amazon see such data as commercially sensitive. However, it is to be hoped that in time they may be persuaded to release more data.

Chapter 4

Storage protocols in the data centre

This chapter explores the interactions between storage and networking within the data centre. Unlike many other data centre applications, storage is often highly bandwidth sensitive. Consequently it has very different requirements from the transport layer that may bring it in conflict with latency-sensitive traffic. In this chapter, I present Trevi [120], a novel approach for data centre storage based on multicast and fountain coding. Trevi is a blob store, designed to allow storage traffic to act as a scavenger class, receiving lower priority at switches and allowing latency sensitive traffic to pass unhindered.

Trevi emerged from an idea I had to use some form of sparse erasure coding to provide a storage protocol able to act as a scavenger class within the data centre network. I collaborated with Dr. George Parisis as he had significant experience in network coding. My contrbutions were the original idea, basing the protocol on Microsoft's Flat Datacenter Storage[107] and the whole section on flow control. I also did all the evaluation at the end of this chapter.

4.1 The conflict between storage and latency

Data centre application developers have an adage that it is better to move the application to the data than the other way round. Typically this means distributing the application to nodes that already have the correct data in memory or cache. However, there is always a need to populate the data in the first place, as well as the need for other tasks, such as backup, replication and maintenance. This means there is always a large amount of storage traffic using any data centre network. Furthermore, there are some data-intensive tasks such as indexing and sorting that require large amounts of data to be transferred to and from storage.

One of the big challenges is to allow storage traffic to co-exist with latency sensitive traffic on the same commodity network hardware. HULL [7] and DCTCP both choose to trade lower throughput for background (e.g. storage) traffic in exchange for lower latency for

short foreground flows. However, as my simulations in Chapter 3 indicate, this can come at a heavy cost for the background traffic flow completion time if there is a greater share of foreground flows. Some more novel solutions have been suggested such as qjump [53], which allows traffic to trade bandwidth for lower latency, offering hard guarantees on bot.,However, that requires modification at the switches and servers.

4.2 The need for better storage

As described in Chapter 2, data centre storage systems range from in-memory key-value stores for regularly accessed query data through to tape and spinning media for "cold" storage of backups and log information. Some of these stores are simple blob stores, some are block stores and some are hybrid.

Common to all existing storage systems is the need to meet demand for high throughput while keeping the cost of deployment and maintenance low. Consequently they are usually built on top of TCP and exploit commodity hardware to communicate, process and store data. TCP leads to a number of known limitations:

TCP Incast A well known consequence of TCP's usage is TCP Incast: *"a catastrophic TCP throughput collapse that occurs as the number of storage servers sending data to a client exceeds the ability of an Ethernet switch to buffer packets"* [126]. Incast is obvious for specific I/O workloads, like synchronised reads, but can also occur whenever severe congestion plagues the network, as TCP's retransmission timeouts are orders of magnitude higher than the actual Round Trip Times (RTTs) in data centre networks. Several techniques have been proposed to mitigate the TCP Incast problem [160, 176, 169], but their deployment is hindered as they require extensive changes in the OS kernel or the TCP protocol itself, or because they need network switches to actively monitor their queues and report congestion to end-nodes.

Wasting network resources in exchange for resilience Existing systems like Google FS [51], Ceph [164] and DHTbd [121] either send multiple copies of the same data or apply an erasure code to the data and send the encoded pieces to multiple storage nodes [2, 36] to support resilience. The first approach effectively divides the performance of every write request by the number of stored replicas since each one of them has to be unicast separately. The latter does better in terms of required storage space, but erasure blocks need to be updated when writing to one or more blocks of data, and their old value must be fetched before the update.

Expensive switches to prevent packet loss Realistic solutions to the TCP incast problem often involve using network switches with large (and energy-hungry) memory for buffering packets in-flight. This in turn has a negative effect on the latency-

sensitive application traffic and is another example of the sort of trade-offs that data centre designers must make at present.

Lack of parallelism when multiple replicas exist In systems that store copies of the same data in multiple locations (and when consistency among replicas is maintained, or where outdated replicas are flagged) only a single storage node is used to fetch the data, leaving the rest of the nodes idle. Data reads must be parallelised by striping a single blob to multiple disks and hoping that I/O requests are uniformly large. Usually, deep *read-ahead* policies are employed to force the system to fetch multiple stripes simultaneously, although this approach can be wasteful for workloads with small random reads.

No (or basic) support for multipath transport Most data centres now offer multiple equal (or near-equal) cost paths through their fabric [52, 3] and are thus in an ideal position to support multipath transport, but exploiting these paths in parallel is hard. Protocol extensions like MPTCP [47] require extensive changes in the network stack (though [135] explores using MPTCP in data centres). Other efforts seek to balance flows across different paths in a data centre in a deterministic and rather static fashion [3, 67]. More dynamic approaches to balance packets across different paths to the same host are prohibitive because out-of-order packets can degrade TCP's performance significantly.

4.3 A strawman design for Trevi

Trevi is designed to operate as an object store. This follows a trend in many recent data centre storage systems [51, 107, 165, 114]. I start by describing a strawman design that focuses on how blobs are transferred between clients and servers. I abstract out details of the OS integration (e.g. as a distributed block device, file system or key-value storage library), and omit details on how blobs are resolved to storage nodes, failure recovery or system expansion. Trevi can be integrated in any storage system that assigns blobs or stripes of blobs to storage servers deterministically.

4.3.1 A coding-based blob transport

In its simplest form, Trevi is a unicast scavenger transport that can make use of multipath approaches to store data blobs across different nodes in the data centre. For the purposes of this strawman I will assume that Trevi requires the following characteristics. Firstly, it should be resilient to the loss of large numbers of packets so that it can efficiently scavenge network bandwidth. Secondly, it must be agnostic to packet re-ordering so that packets can be sent down multiple paths. Thirdly, it should not require explicit retransmission

of missing data. In other words it should utilise some form of forward error correction coding.

4.3.2 Multicast or unicast?

In its simplest form above, Trevi is a unicast system. However this would mean it suffers many of the same issues as TCP. Consequently we decided to make Trevi multicast-enabled. As will be seen below, Multicast is an ideal approach for data centre storage systems. It allows multiple copies of data to be stored at the same time across different nodes simply by those nodes subscribing to the multicast group. It is also possible to modify it so that data can then be read in parallel from multiple nodes, thus reducing the time needed to get the complete file.

4.3.3 A simple flow control

The last part needed for our strawman design is a simple flow control. As explained above, Trevi is based on a combination of fountain coding and multicast. Because fountain coding allows Trevi to be agnostic to data re-ordering and loss we adopt a simple receiver-driven flow control as discussed below. This has several benefits as explained in section 4.4.3 below. It is also a good match to a storage system where there are additional variables such as seek latency and storage-controller bottlenecks that affect transfer rates.

4.4 The Trevi system

I now move on to a description of the complete Trevi system. This is made up of three main parts. A storage architecture, a multicast data transport and a simple receiver-driven flow control.

4.4.1 The underlying storage architecture

Trevi requires a simple architecture that allows blobs of data to be multicast to a set of nodes with all nodes being aware of the location of each blob. As it is a storage system it needs to also cope with updates, deletions and node failures.

Rather than re-invent the wheel I suggested that Trevi should build on the FDS system [107]. FDS places stripes of larger *blobs*, namely *tracts*, in one or more *tract storage servers* in a deterministic way. A *Tract Locator Table* (TLT) is used to determine the location(s) at which a given tract is stored. This TLT is cached locally at every client and

is updated whenever a node fails or a new node joins. The only extension required to support our approach is the addition of a column that stores some multicasting information (e.g. an IP multicast group) in the TLT.

4.4.2 Fountain Coding

Looking at the requirements set out in the strawman, we decided to base Trevi on Fountain Coding. Fountain coding allows us to provide a storage service that is tolerant to packet loss, but without the need for explicit retransmissions or timeouts. Fountain coding was introduced a decade ago[93, 151] as a way of reliably multicasting over unreliable, broadcast-enabled mediums, where losses are the norm and no feedback channel exists[22]. The big advantage of fountain coding for Trevi is that it is specifically designed as a multicast system. As with any system, fountain coding carries some penalty in exchange for the benefits listed above. Specifically, it requires extra computing resources to encode and decode symbols and also has a small penalty in terms of bandwidth required.

Figure 4.1 shows a simple example of fountain coding. Here the data has been uniformly divided into 6 chunks or symbols, D_1–D_6. These have been encoded into 8 codewords, C_1–C_8, by combining *degree* number of *neighbours* using XOR (shown as + in the figure). The receiver sees a stream of incoming codewords C_1, C_2, C_7, C_3, C_4 and C_5. It sequentially uses any codewords with *degree 1* to partially or fully decode other symbols by XORing them with the decoded symbol. This form of encoding does carry a penalty in that it requires slightly more codewords to be generated than there were original symbols. In turn this leads to the network bandwidth penalty mentioned above.

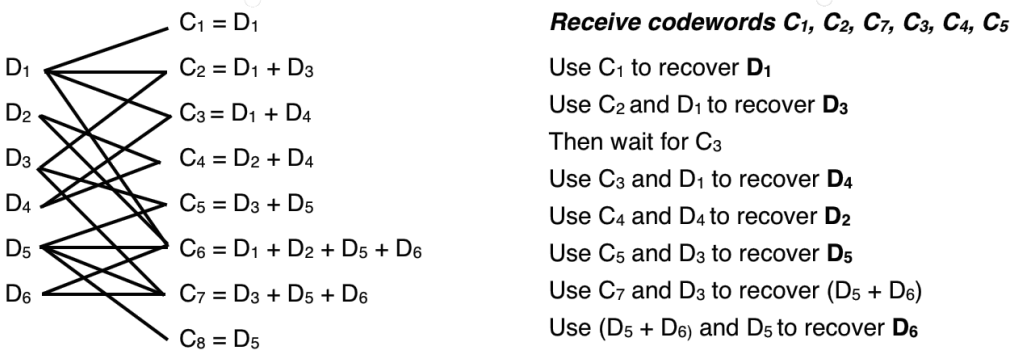

Figure 4.1: A simple fountain coding example. The + symbol indicates XOR.

Fountain coding allows you to recover the original data regardless of which codewords are received and in what order. If a codeword is lost it need not be retransmitted, all that's needed is another codeword covering the same data. Thus fountain codes are resilient to network re-ordering and loss. This makes them ideal as the basis for scavenger-style transports. The key to fountain coding lies in the choice of statistical distribution used to

choose the *degree* and *neighbour set*. Different statistical distributions have been proposed [93, 151]. The overhead they introduce can be as low as 5% [26] and proprietary raptor code implementations report a network overhead of less than 1%[1].

4.4.3 Trevi flow control

Traditionally, the fountain coding-based transport model is *push* based. Senders start sending symbols until all receivers have decoded the data and have sent a notification to the sender or unsubscribed from the multicast group. In layered multicast [97], receivers play a more active role by subscribing to and unsubscribing from multicast groups, which represent different coding layers, according to the network congestion.

In Trevi, receivers have a choice of three flow control schemes.

1. a *push* communication scheme where the sender keeps sending codewords until all receivers have asked it to stop.
2. an active *pull* communication scheme where a sender sends one or more symbols only when explicitly requested by a receiver.
3. a hybrid *push/pull* communication scheme where the sender transmits sufficient codewords to ensure all receivers should receive the file and then sends additional codewords if requested.

4.4.3.1 The pull flow control scheme in detail

In order to facilitate the pull flow control scheme Trevi receivers include a statistically unique label when requesting an encoded symbol so that the RTT can be calculated upon receiving a symbol sent in response to that request (and therefore carrying the same identifier). No action is taken when symbols are lost. We use labels instead of sequence numbers since packets can be out-of-order. All symbols are useful whenever or from whichever path they arrive.

A receiver-driven approach for requesting symbols simplifies flow and congestion control and guarantees that no extra symbols are sent after the receiver decodes the initial data. More specifically, a receiver adjusts the number of pending symbols' requests (called the *window* of requests) to handle changes in:

1. The rate at which a storage server can store data. This is determined by the type of storage (spinning disk or SSD), the level of fragmentation of the disk and any bottleneck that may happen at the actual storage controller.

[1] see https://www.qualcomm.com/documents/why-raptor-codes-are-better-reed-solomon-codes-streaming-applications (accessed September 2017)

2. The rate at which a sender can send data. This is limited because one sender may be serving requests to many receivers. This may also be limited in a virtualised environment where the sender has to share physical compute and network resources with other VMs.

3. The congestion in the network.

The first point has not been addressed in past systems, especially for storage servers with spinning disks. In such cases the network bandwidth can be much higher than the disk array's throughput. Hence, there is no point in a storage server requesting more data than the amount it can actually store, sparing the extra bandwidth for other nodes in the network.

The data rate of a sender is subject to variability because it may serve multiple requests from receivers at the same time. Our approach ensures that senders will be requested to send encoded symbols at a rate that they can actually cope with. This rate can be achieved by adjusting the window of pending symbols' requests when the RTT increases. Note that the RTT also increases when symbols are buffered in switches, but in both cases the window of pending requests should be decreased.

Receivers also react when congestion occurs in the network by decreasing the number of pending symbol requests. Congestion can be inferred and avoided by actively monitoring the RTTs for each symbol request. Additionally, losses in the network can be estimated since a receiver can know for which requests respective symbols did not arrive. It is worth highlighting that the notion of the window, as introduced above, is different from the classic TCP flow and congestion windows. There are no timeouts and no retransmissions in Trevi, and instead some internal timeouts which are only necessary to remove stale requests from the current window and update the loss statistics. These timers are adjusted based on the monitored RTTs but do not trigger any retransmission requests. In the worst case, if such a timer expires and an encoded symbol arrives after the respective request was removed, the receiver just increases the timer value for the upcoming requests; there is no penalty for the early timer expiration because the encoded symbol will be used in the decoding process just like any other symbol (remember with fountain coding there is no concept of out-of-order symbols!).

This *pull* flow and congestion control scheme is by definition incast free. Packet losses are less important than in TCP; packets are not individually identified and nor does the receiver ask for specific "lost" packets and there is no notion of retransmissions because of timeouts. Hence, there is no need to extensively buffer packets and desperately try to deliver them to their destination. Smaller buffers means cheaper and more energy-efficient switches and/or more buffers for other TCP traffic which can be easily isolated from storage traffic using simple priority queuing mechanisms.

4.4.4 The hybrid push-pull flow control approach

A simple push-pull flow control scheme would help to minimise the overhead due to requesting encoded symbols separately. When transmitting data, the source knows how many symbols need to be transmitted in the absence of loss. This number depends on the statistical distribution that is used to calculate the degree of each symbol. Initially the source is set to send this much data and then pause. If no symbols have been lost, the source is notified by all receivers that the blob has been successfully decoded, and then it simply stops. If any symbols have been lost, receivers start issuing pull requests for additional required symbols, as described in Section 4.3.

4.4.5 Flow control refinements

The simple flow control approaches outlined above can be improved in several ways.

4.4.5.1 Priority and scavenging

Fountain coding is inherently resilient to loss. This makes it well suited to scavenger-type QoS mechanisms. In such mechanisms, scavenger traffic receives a very low QoS priority which means it will be preferentially dropped in the presence of any congestion. This in turn means that in the absence of congestion, such scavenger traffic can be sent at near-line rate. In its simplest form Trevi could use strict priority queuing mechanisms at the network switches. This would mean other traffic is served preferentially before the Trevi packets are forwarded. This idea has some similarities to the priority forwarding of headers in NDP[63].

Alternatively, if one is able to rapidly detect how much traffic there is currently in the network, a sender could actively control the rate at which it transmits Trevi packets. Pre-Congestion Notification [99] is a measurement based admission control system for real time traffic. Traffic traversing each path through the network is viewed as a single combined flow, called an ingress egress aggregate. Central to PCN [39] is the concept of a virtual queue—a simple mechanism that behaves like a queue with a slightly slower drain rate than the real queue.[2] As the virtual queue passes a lower threshold the queue is defined as being in a pre-congested state. If the virtual queue continues to grow it eventually passes a second threshold, which indicates that the real queue is about to start to fill. In PCN, crossing either of these thresholds causes arriving packets to be marked, and the aggregate rate of marks is used to decide whether to admit new flows or to drop existing flows.

[2]Since 2010, all Broadcom router chipsets have natively supported a form of virtual queue called a threshold marker, which is ideally suited to this purpose.

A similar virtual queue technique could be applied to our fountain storage system. This would allow the storage control nodes to assess how much other traffic is competing with the storage traffic. This can then be used to determine the safe rate at which to send data. This is particularly relevant for the case of multi-sourcing data from many replicas to one client machine. In this instance there is a very real risk of causing the final network queue nearest to the client to become congested. Simply running a virtual queue on this node, and using this as one of the parameters in the destination-driven flow control would significantly reduce this risk. It is interesting to note that the authors of HULL [7] have also looked to similar techniques to improve the latency performance of data centres using "phantom" queues that are a simplified form of virtual queue.

4.4.5.2 Optimising for slow writes

If one storage node is writing data much more slowly than the others in its group, then it will have a disproportionate effect on the rate at which all others can write. This is similar to the idea of stragglers in partition-aggregate systems. There are two potential solutions to this problem. Firstly, any node that is significantly slower could simply be removed from the multicast group. Secondly, the sender may choose to ignore the slow node and simply go faster than it can cope. If the node becomes overwhelmed, it can unsubscribe itself from that multicast group and mark that tract as unreadable. Both of these approaches have implications for the degree of replication within the system, but may significantly improve performance.

4.4.6 Multicasting data

As described in the FDS paper [107], write requests for (part of) a blob are first resolved utilising the hash of the blob's identifier to locate it in the Tract Locator Table (TLT) (step 1 in Figure 4.2). This gives the client the addresses of individual servers as well as the multicast group(s) to which it should send the tracts in the write request. Tract storage servers are assumed to be already subscribed to the right multicast groups, according to the information in the TLT (all entities in the storage network have the same view of the TLT [107]).

When a client needs to write a tract to a number of servers, it first sends a *prepare* notification to all replica points (step 2). This notification must be sent in a reliable way, either via a separate control TCP connection or by employing a retransmission mechanism for this packet, which includes the identifier of the specific tract (an encoded symbol can be piggy-backed in the notification). Upon receiving this notification, storage servers start requesting encoded symbols from the client. For write requests, servers (denoted as r in step 3) are the receivers of symbols.

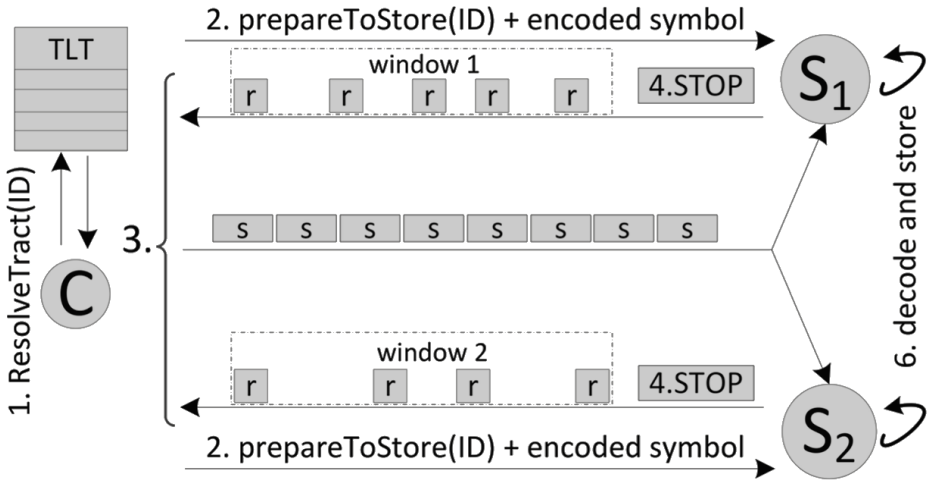

Figure 4.2: Trevi writes data using a pull-based transport API

It is important to note that although servers run their own flow and congestion control windows, the client always sends encoded symbols based on the requests coming from the slowest server. The rest of the servers slow down the rate at which they request symbols to match the incoming rate (the separate windows of requests converge to that of the slowest storage server). This way, the client is able to multicast encoded symbols, denoted as s in step 3, to all servers at a rate defined by the slowest server (this feature is beneficial for the network because the multicasting rate is smoothed by the slowest server). Replication is by definition a synchronised operation, which is completed only when all replicas acknowledge the reception of a tract. In Section 4.4.5.2, we describe a potential optimisation in case one storage server is straggling.

Finally, each server sends a *stop* notification containing the identifier of the stored tract, which must be reliably delivered to the client (step 4). The client stops sending encoded symbols after receiving such notifications from all storage servers that store the specific tract.

Reliable multicasting in our approach supports data replication with the minimum network overhead and increased efficiency by just multicasting replicas to the set of nodes that are deterministically chosen to store a given set of data. Existing systems [164, 107, 121] select the nodes for storing data deterministically, and therefore the only requirement is to have these nodes subscribing to a multicast group that is specific to the dataset assigned to them.

4.4.7 Multi-sourcing data

One of the powerful features Trevi provides is the ability to multi-source data from all replicas at once without the need to coordinate. As long as the data has been encoded differently at each store then any codewords received will allow the receiver to decode

more data. The process is similar to that used for writing. After resolving the nodes that store a specific tract (step 1 in Figure 4.3), a client sends a *get blob* request to all these nodes (step 2). All servers acknowledge the reception of the request (step 3). As illustrated in Figure 4.3, a symbol can be piggybacked in the acknowledgement packet.

After receiving acknowledgements from all servers, the client starts requesting encoded symbols from all storage servers that hold an updated version of the tract at the same time (for read requests, the client is the receiver of symbols). Some systems [107, 164, 121] support mechanisms which assure that nodes with outdated data will never be selected to fetch data. Trevi adopts a similar approach to ensure that such nodes will never be chosen to contribute symbols representing outdated data. As shown in Figure 4.3, the client keeps separate windows of pending symbol requests for each storage server.

Figure 4.3: Trevi reads data from multiple sources using a pull-based transport API

Each storage node creates and sends encoded symbols in an independent and uncoordinated way in response to requests from the client (step 4). The only requirement here is that there should be a reasonable probability that symbols generated by each client are different. To ensure that, some randomness must be added in the way storage nodes select the seeds used when they calculate the degree of each symbol. Two different seeds will statistically produce two different encoded symbols. This could be achieved by simply using the address of the storage node to generate the initial random seed. Note that there is no need for any kind of synchronisation or coordination for this scheme to work. Furthermore, even if the symbols aren't different, the system will still perform as well as any current approach. Servers transmit symbols with different rates (defined by the client's flow control mechanism) and each server contributes as many symbols as it is able to produce and transmit. Servers will never send an encoded symbol unless they are requested to do so.

Finally, the client reliably sends a *stop* request (step 5) to (separately) let each server

know that it decoded the requested blob. The whole procedure ends when the client passes the decoded blob to the application or the file or block subsystem (step 6).

The multi-source transmission provided by Trevi allows storage resources to be fully utilised even when the I/O workload cannot be parallelised (e.g. for smaller read requests involving a single stripe). More specifically, all storage nodes that hold an updated version of some data can contribute to the transmission of the data to a client. This feature provides a second, inherent level of load balancing when fetching data, the first being the striping of blobs to multiple storage nodes.

4.5 The likely benefits of Trevi

Fountain coding is a powerful technique that gives Trevi many direct benefits over TCP. Referring back to the list of issues in Section 4.2, we can see how fountain coding avoids or removes the issue altogether:

TCP Incast Fountain coding [22] eliminates the need for retransmission upon packet loss. Instead, additional encoded symbols are transmitted until the receiver can decode the missing data. By definition, no incast can ever occur (§4.4.3).

Wasting network resources in exchange for resilience With fountain coding, write requests can be multicast to all replica points, thereby minimising network overhead and increasing energy efficiency (§4.4.6).

Expensive switches to prevent packet loss Packet losses are much less important when using fountain coding. Consequently we can reduce the size of network buffers, or treat storage traffic as a lower QoS class with limited buffer space. Hence, our approach requires less energy to power the memory required for buffering storage requests (§4.4.3).

Lack of parallelism when multiple replicas exist By contrast, fountain coding allows simultaneous multiple sources when reading data, leading to more efficient utilisation of storage resources even when the I/O workload cannot itself be parallelised (e.g. for smaller read requests involving a single stripe) (§4.4.7).

No (or basic) support for multipath transport Fountain coding schemes are much more forgiving of dynamic balancing, since all symbols are useful and there is no notion of out-of-order packets. Unlike TCP (where balancing happens on a per-flow basis to avoid out-of-order packets throughout a flow's lifetime), in our approach encoded symbols can be balanced independently. This provides a lot more flexibility in the design of in-network multipath mechanisms (§4.4.3).

In addition Trevi's multi-sourcing of data reads allows it to offer lower latency for storage traffic without needing to compromise the latency sensitive data.

4.6 The price to pay

In previous sections I discussed the problems with existing storage systems that are based on commodity hardware and which operate on top of TCP. I also explained how Trevi avoids these. In this section I discuss the potential downsides of Trevi, which are all related with the fountain coding technique. However, my co-authors and I believe that none of these issues is significant in a data centre storage context.

CPU Overhead. Fountain coding involves encoding and decoding of information on the sender and receiver side, respectively. Here, the overhead comes from generating random numbers according to the used statistical distribution (e.g. the Robust Soliton Distribution [93]), actually segmenting the data and, mainly, from XORing several pieces of the initial data to produce each encoded symbol to be transmitted. I am confident that this overhead will not be prohibitive with respect to Trevi's applicability. First, modern hardware in data centre networks consists of fast, multi-core CPUs that could easily cope with the encoding and decoding processes. Second, the process itself is highly parallelisable, thus one could take advantage of the multiple cores or even offload it to hardware (e.g. GPU or NetFPGA [105]—though as I say below, this could have a significant impact on energy use). Finally, an opportunistic approach, where a master replica decodes and stores the original blob while other servers serve the statistically-required number of symbols to decode the blob, can be used to minimise the overall CPU overhead of the storage system.

Network Overhead. As I mentioned in Section 4.3.1, fountain coding involves a constant penalty in terms of network overhead. This overhead is not significant taking into account that in Trevi there is no TCP incast, which can severely degrade the I/O performance, and that we save network resources by multicasting write requests and moving storage traffic to a scavenger class.

Memory Overhead. In Trevi, a sender needs to have fast access to a blob of data as long as it creates new symbols (in response to respective requests). This implies that a blob must be in memory until all receivers have successfully received and decoded the blob. This requirement could potentially have an impact on the required amount of memory to support multiple I/O requests in parallel. However, more control of the storage buffer cache (using direct I/O and a userspace cache, or even libOS techniques [94]) makes it possible to partially map larger blobs into memory to allow encoding to be suspended if the memory is required elsewhere. This helps to mitigate the tail of requests for a given blob, especially if there are stragglers in the storage cluster.

Energy Efficiency. The impact of Trevi on energy consumption has yet to be evaluated this. On the one hand, Trevi requires some energy intensive functionality at the servers. More processing power is required to segment the data, and significnant power is needed to encode and decode the data. Addiotionally, slightly more data needs to be transmitted compared to a regular TCP blob transfer. However, since the data can be multicast instead of multi-unicast and because symbol loss does not trigger any Incast, power-hungry buffers in the network switches can be reduced. Clearly there will be a trade-off between these two and it is hard to predict what the overall impact will be.

Unsuitability for Dynamic Storage. One clear drawback of Trevi is that it cannot be used for dynamic storage. Trevi relies on creating codewords across an entire storage blob, consequently it needs the whole blob to exist before it can be encodeed. This also means Trevi is unsuitable for real-time or incremental backup systems. This constraint is discussed in more detail in 4.8 below.

4.7 Exploring the impact of Trevi

This section seeks to answer the question "How well does Trevi perform compared with existing approaches?". Trevi was designed to achieve two aims: to reduce the impact of storage traffic on foreground flows and to improve the performance of storage flows by leveraging the benefits of multicast. As a minimum to to be viewed as successful, Trevi should perform better than TCP for both the storage and foreground traffic, and ideally it should perform as well as DCTCP or even outperform it.

As yet there is no code available to run Trevi in a real network[3]. This means there are two possible approaches to assessing how well it performs. A thought experiment can be constructed that makes assumptions about how Trevi will affect other traffic and uses these to predict the impact. The problem with this approach is it will quickly become extremely complex as you scale up the size of the network. The other approach is to create a simulation model of Trevi and then compare this against existing approaches.

4.7.1 A simple thought experiment

As a trivial starting point I constructed the following simple thought experiment. Picture a network where traffic is a mix of long-running storage flows and shorter foreground flows consist of short messages and longer responses. This is a reasonable assumption according to the literature on data centre traffic. In the following I will attempt to gauge at what point it becomes efficient to use a Trevi-style transport for the storage traffic. For now I will assume that it is a unicast variant of Trevi as this will make the comparison

[3]Dr Parisis and his group at the University of Sussex are working on related ideas

with TCP easier. Further I assume that Trevi requires a fixed overhead of 10%. My final simplifying assumption is that Trevi traffic is given extremely low priority at queues. I start by contrasting two extreme cases—a lightly loaded network where there is plenty of spare capacity on average and congestion only occurs as a result of TCP's congestion control and a heavily loaded network where there is barely any spare capacity.

Light Load In a lightly loaded network, if all the traffic runs TCP then the storage flows will quickly grow their congestion windows and will take a large share of the network until they finish. This is a simple consequence of the fact that TCP is optimised to favour longer-running flows. During this time any foreground flows will see an increase in delay due to queues building in the network. Once the storage flows finish then the queues will go and the foreground flows will complete much faster. In other words the likely outcome will be a large variability in flow completion time for foreground flows. This will potentially be exacerbated by incast.

If the storage traffic uses Trevi then this will change. Trevi traffic is preferentially dropped at queues. Therefore it will not impact foreground traffic. Equally, because there is relatively little foreground traffic it should be easy for sufficient Trevi packets to reach their destination and thus the Trevi storage flows should complete in a similar time to TCP. This suggests that in such a network using Trevi will lead to a significant performance improvement. However a lightly loaded network is an inefficient use of resources. It is hardly news that by reducing the load in a network you can significantly improve its performance—at a low-enough load, even TCP will perform extremely well for foreground traffic.

Heavy Load Now consider a network that is heavily loaded such that it is suffering frequent packet drops. If all the traffic runs TCP then the congestion in the network will lead to frequent packet drops and retransmissions. This will affect both the storage and foreground flows. For the storage flows it will act to limit the size of congestion window they can achieve and will increase the flow completion time. For foreground flows it will lead to even greater variation in flow completion time.

If the storage traffic uses Trevi then this will be preferentially dropped at the queues. In turn this will mean that the relative congestion seen by the foreground traffic will go down. This will have the positive effect of reducing the variability of flow completion times and improving the overall average. However, if too much Trevi traffic is being dropped then at some point you will reach a stage where insufficient packets get through to allow the Trevi flows to complete.

In other words when the load in the network is too high neither TCP alone nor TCP combined with Trevi is able to work effectively. Again, this is not really surprising - if you picture the network as a time and space switch then as you approach capacity you run out of free slots to move flows into. The result is that eventually you end up with congestion collapse and all flows suffer.

Table 4.1: Comparing the impact of increasing the ratio of Trevi traffic

Storage Traffic Ratio	0.7	0.7	0.8	0.8	0.9	0.9
Trevi Overhead	5%	10%	5%	10%	5%	10%
Avail. Foreground Capacity	265MB	230MB	160MB	120MB	55MB	10MB

The implication is that there must be a sweet spot where the network is able to run efficiently (at reasonable load) but with Trevi traffic still able to get sufficient bytes through to allow flows to complete. The following is a simple attempt to calculate where this sweet spot might lie.

Trevi traffic is preferentially dropped at queues. Consequently any time the network becomes congested Trevi will have to send significant amounts of extra traffic. As a rule of thumb a network running TCP starts to degrade rapidly once congestion approaches 10% (e.g. once more than 1 in 10 packets are being dropped). So to find the ideal spot we need to look at how many TCP drops we can trade for Trevi drops before Trevi stops working.

Trevi breaks down once it is no longer able to get enough bytes to the receiver to decode the data. Assume that in a data centre $S\%$ of bytes belong to storage flows. If Trevi needs an overhead of δ, then in the Trevi case you need $S + (S.\delta)$ bytes to reach the receiver. In a fully loaded network with capacity C this equates to:

$$(S + S\delta).C \; bytes \tag{4.1}$$

In turn that means if you lose more than:

$$(1 - (S + S\delta))C \; bytes \tag{4.2}$$

then Trevi no longer functions. Table 4.1 seeks to put this in context (this assumes a 1GB network link).

So this simple calculation shows that the Trevi overhead has a big impact on how much foreground traffic can be supported. Taking potentially realistic figures of 90% storage traffic and 10% overhead you would only be able to send 0.1% foreground traffic before the storage traffic suffers. Rearranging 4.2 we can see that with 10% Trevi overhead your storage traffic cannot exceed 82% network capacity.

This thought experiment suggests that Trevi might be useful in networks where the utilisation is such that TCP would start to trigger too much congestion. Of course, even with the low congestion that the preferential dropping of Trevi packets gives, there will still be queues building at network switches which will have an impact on TCP throughput.

4.7.2 ns2 Simulations

In order to better assess the behaviour of Trevi, I created a simplified ns2 model of a Trevi-style storage transport protocol. This is described below.

Basic Operation

In my model, each Trevi source sends out packets at a constant rate that is somewhat less than line-rate. This is to prevent the first queue from overflowing—in the real system this rate would be controlled by a combination of receiver feedback and knowledge of the state of the NIC queue. However ns2's queue models are more simplistic than this. Packets are all 1500 Bytes long, although clearly in many modern data centres with fast fabrics they would actually use jumbo frames (9000+ Bytes). Packets are marked to indicate their origin, destination and details about the flow. Once the receiver has seen enough packets to account for the size of the flow plus the overhead it tells the source to stop sending.

Metadata Header

Each Trevi packet carries a small metadata header that indicates the flow it belongs to and the total number of bytes that need to be received for that flow (e.g. the number of packets needed to re-assemble the fountain-coded object). The number of bytes sent for each flow is increased by a fixed percentage to allow for the overhead seen in fountain coding. Currently, I have chosen 10% overhead, although in a real system the overhead might be less than this. As with all additional headers in ns2, this creates simulation overhead, but the size of the header is not counted against the size of the packet. In a real Trevi system it is envisaged that there would be a small control channel that would send the metadata.

Queues

Every queue has been modified to impose a strict drop policy on Trevi packets. This means that if the queue is dropping or CE marking packets, then any Trevi packets in the queue will be preferentially dropped. Originally, I tried to use the built-in DiffServ models within ns2. However I was unable to achieve the strict drop behaviour I wanted. Consequently, I ended up directly modifying the underlying C++ queue classes. Unfortunately, this has a notable performance impact on ns2 as it can require the entire queue to be traversed to see if there is a Trevi packet available[4]. However, this impact is manageable (Trevi simulations end up taking about 4 times longer than the equivalent TCP simulations). In

[4]It would be possible to add a counter at each queue to record the number of Trevi packets actually in the queue. This would save the need to traverse the queue if no Trevi packets are actually present.

Table 4.2: Summary of the simulation matrix

Name	Foreground type	Storage type
TCP-TCP	TCP-SACK	TCP-SACK
DCTCP-TCP	DCTCP	DCTCP
TCP-TREVI	TCP-SACK	Trevi
DCTCP-TREVI	DCTCP	Trevi

a real switch queue there is already logic present to do this sort of prioritised dropping more efficiently.

The Receiver

The Trevi receiver sees packets as they come in. It keeps a count of the total bytes received for each flow as well as storing the arrival time for the first and most recent packets in that flow. Using ns2's global overview, the simulation script periodically checks each Trevi flow, and if it sees that the correct total of bytes has been exceeded it will stop that flow. This is equivalent to a message being sent back to the source, but is simpler to implement in ns2.

4.7.3 Simulation setup

In order to compare the relative performance of Trevi against existing transports I used a **k=6** Fat Tree topology, modifying the ns2 code used by the DCTCP researchers and others[6]. I used separate traffic generators for storage and foreground traffic. Initially I attempted to set up multicast connections for the Trevi traffic. However, this proved impossible at anything above a small scale. The overhead added by the need to have multicast traffic classifiers and packet copying at every queue slowed down simulations to such an extent that the simulations became unworkable. The lack of multicast means I am unable to see the full potential benefits of Trevi in these results, but if there are benefits without it then it is reasonable to assume that adding multicast would increase these benefits.

4.7.3.1 Simulation matrix

In order to properly compare the impact of Trevi, I ran 4 sets of simulations. These are designed to test the full set of combinations of DCTCP, TCP-SACK and Trevi. These are shown in Table 4.2.

I repeated each set of simulations with a light and a heavy storage load to explore the impact of this on the performance. For the DCTCP simulations I used the same settings

as described in[6]. These are $B = 10Gbps$, $K = 65$ packets, and $g = 1/16$. The RED parameters are also set such that the queue is measured in packets, with all packets marked once the threshold, K, is passed. For TCP-SACK I used the following settings: DropTail queue with a 1.5MB buffer, minimum RTO of 1ms and a segment size of 1460 bytes. In both cases I used delayed ACKs (ackRatio=2).

Trevi traffic was paced at the sender with 1 packet sent every 1 µs. This rate equates to a fairly slow 1.2Gbps. In a real network it might be that Trevi could send significantly faster. However, I found early on in my simulations that if I set the rate too high it would quickly swamp the foreground traffic (especially when running TCP). This is because at any given moment, 3-5 Trevi flows will be active. This causes overwhelming congestion at the ns2 source node and has a significant impact on performance. To find the ideal rate I performed several short simulations (10s simulated time), varying the rate each time. I then found the rate that balanced the impact on FCTs for both the foreground and background traffic, while still allowing sufficient storage flows to complete. As explained in section 4.4.3, a real-world implementation of Trevi would use dynamic rate adaptation to find the safe sending rate.

4.7.3.2 Topology and Traffic

Each simulation was run on a K=6 Fat-Tree topology. That gives a total of 54 hosts, 36 aggregation switches and 9 core switches. The bandwidth at each level was 10Gbps, and the transmission latency was set to 1 µs per hop. The traffic matrix was generated as follows:

Foreground traffic: Each host opens a flow to each other host. I used my modified traffic generator class described in section **??**. Flow sizes are between 1kB and 10MB, drawn from a Pareto distribution. Flow inter-arrival times are Exponential, with an average of 3ms.

Storage traffic: Every pair of nodes also runs a storage flow, but with a much longer time between flows. This roughly replicates cases where regular backups are being made with multiple copies distributed across the network. Originally I intended to use an exponential distribution for storage sizes. However this did not tally well with the various DC traces described in Chapter 2. After trying complex combinations of distributions, I ended up creating a static file with 10,000 flow sizes, distributed roughly exponentially between 800kB and 1GB. This file was generated by fitting to the flow size distribution in Figure 4 of the DCTCP paper[5][5]. The file is randomised at the start of each simulation, and then flow sizes are drawn in turn. The interarrival time for flows is roughly Exponential, with

[5]In the DCTCP paper[5], the authors define background traffic as a combination of "update flows" and "short message traffic". In these simulations I have combined the short message traffic into the foreground traffic since it is unsuitable for Trevi.

a mean of 2s for the heavy traffic and 10s for the light. The light load gives approximately 25% storage bytes while the heavy load stresses the network significantly by increasing this to 95%.

Each simulation lasts for 100s of simulated time which gives an average of 1.5 million foreground and 22,000 storage flows (e.g. the storage distribution repeats twice on average). To give some idea of the complexity of the simulations, each simulation took between 12 and 36 hours to complete.

For every flow I collected details of flow size and flow completion time. For Trevi flows I also collected data relating to how many bytes were dropped in the network. I plotted the FCTs as a box-whisker plot with flow sizes split into bins using a log scale. The results are discussed below.

4.7.4 Results

This section presents the results from two sets of simulations. The only thing that changes between them is the interarrival time for the storage traffic. In the first set the storage interarrival rate is chosen to roughly equate to that used in the DCTCP paper[5]. Consequently you would expect DCTCP to perform well in these simulations. The second set used an interarrival rate 5x higher for storage traffic. This is designed to significantly stress any transport protocol that is seeking to favour foreground traffic. These are respectively described as the low and the high storage traffic matrices.

4.7.4.1 Low storage traffic matrix

To allow an easy comparison between the different transports I have plotted the flow completion times as box-whisker plots rotated by 90°for clarity. Outliers are plotted with a very low alpha which means the density of the line of outliers roughly reflects their distribution. The first graph (4.4) shows the FCTs for the foreground traffic across the 4 scenarios.

As can be seen these results are somewhat unexpected. With the lower levels of storage traffic, TCP actually performs surprisingly well on average for foreground flows with both TCP and Trevi storage. However, Trevi introduces enormous variability in FCTs (with significantly more outliers). By contrast, DCTCP performs slightly less well than TCP for small and medium flows, but significacntly outperforms for longer foreground flows. Trevi has a slight negative impact on DCTCP across all flow sizes.

To try and better understand the reason behind this I also examined the FCTs for the storage traffic. These are shown in Figure 4.5.

These results give a possible explanation for the results seen. The results show that the FCTs for the storage traffic with TCP foreground and Trevi storage are far better than

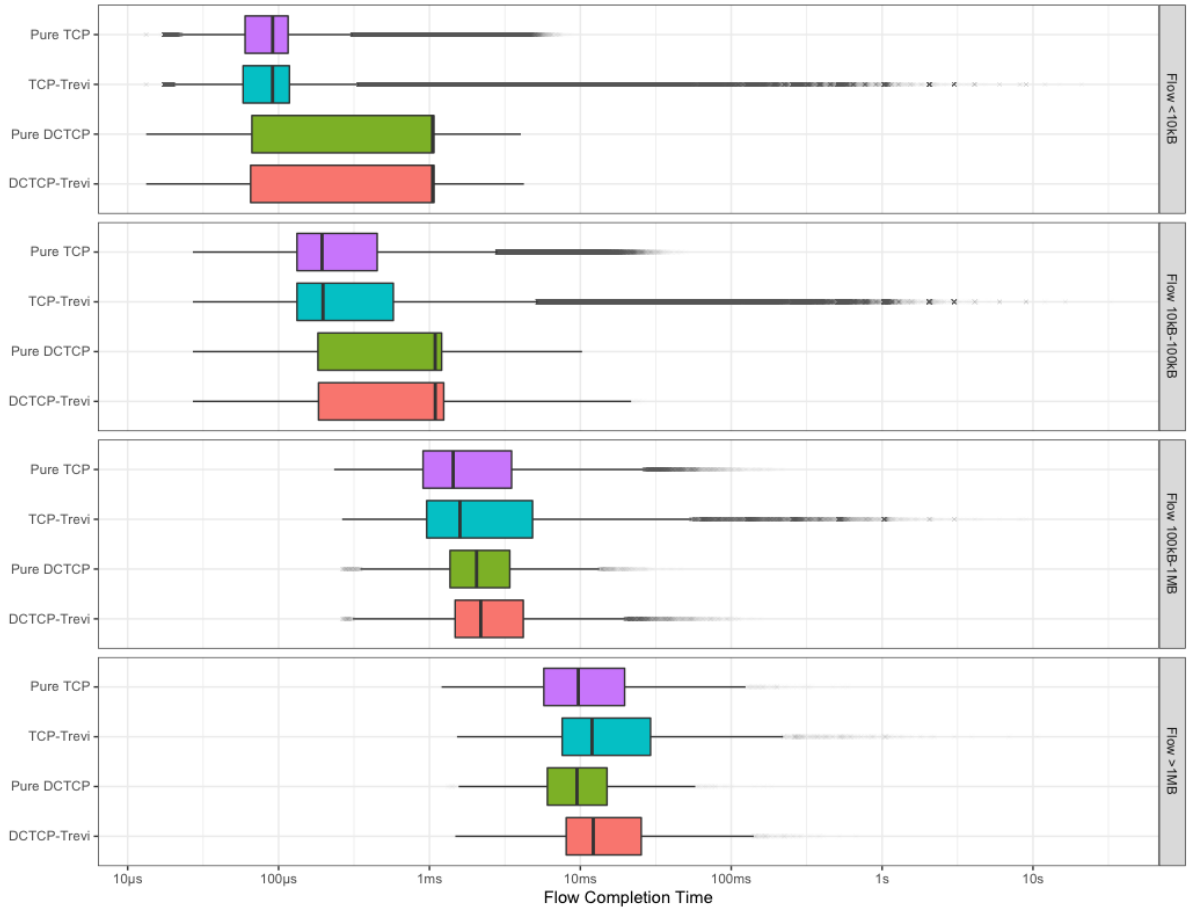

Figure 4.4: Comparing the foreground FCTs for the 4 scenarios with longer inter-arrival times for storage traffic (low storage traffic matrix).

the results for both the pure TCP traffic and the pure DCTCP traffic. Likewise the results for the DCTCP-Trevi simulations are better than all the other results. It is likely that some of this result is down to Trevi being especially effective in the absence of significant congestion. In this case Trevi effectively transfers traffic at >1Gbps immediately, whilst both TCP and DCTCP need significant time to ramp up their sending rates. In turn, the fact that the storage traffic completes so quickly, coupled with the relatively long gaps between new storage flows, means that there is less traffic in the network. Even with the preferential drop of Trevi traffic, if a Trevi packet is actually at the head of the queue it will always be transmitted, hence slightly increasing the FCT for the other traffic.

This still does not explain why a handful of <100kB foreground flows take over 10s to complete. Further research would be needed to explain this properly. One possibility that springs to mind is that this is actually a false result caused by sequence wrapping leading the simulator to think that packets belong to an older flow than they actually do. Another real possibility is that these flows are actually showing incast happening—the arrival rate of foreground flows is extremely high, and it is very possible that large numbers of ACK packets are being lost as a result.

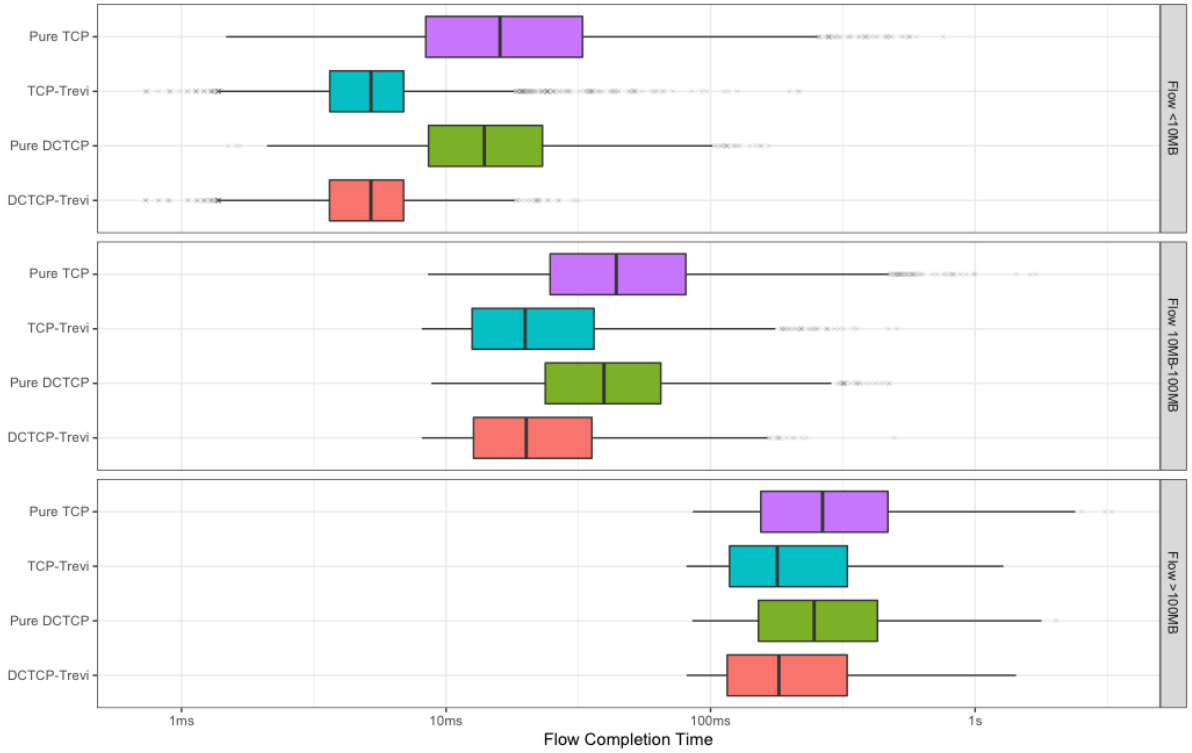

Figure 4.5: Comparing the Storage FCTs for the 4 scenarios with longer inter-arrival times for storage traffic (low storage traffic matrix).

4.7.4.2 High storage traffic matrix

Now I want to explore the impact of increasing the storage traffic. As explained above, the increased interarrival rate means storage traffic now makes up 95% of the bytes in the network. For the pure TCP case one would expect that the FCTs for the foreground traffic would suffer badly (as TCP strongly favours longer-running flows). You'd also expect that the storage flows would exhibit high variance in flow completion time. For the pure DCTCP case, the results of the microbenchmarks in Chapter 3 suggest that storage throughput will be sacrificed to try and keep FCTs low for small flows. In theory because Trevi is preferentially dropped at queues, the performance of TCP foreground flows should improve significantly. What is less clear is how DCTCP will react with Trevi traffic.

The pure TCP case (shown in teal in the graphs) shows the expected behaviour. At all flow sizes the FCT shows a significantly increased spread and there are a lot of outliers.

The pure DCTCP case (purple) is interesting as it shows fewer outliers than TCP, but shows that on average it actually performs slightly worse than TCP for the smallest flows (<10kB). However, with larger flows it consistently outperforms TCP as expected.

DCTCP with Trevi (pink) shows a mixed picture as you might expect. The increased space in the queues due to preferential drop allows the minimum FCT to reduce and slightly improves overall performance. However, this is at the expense of increased variation in

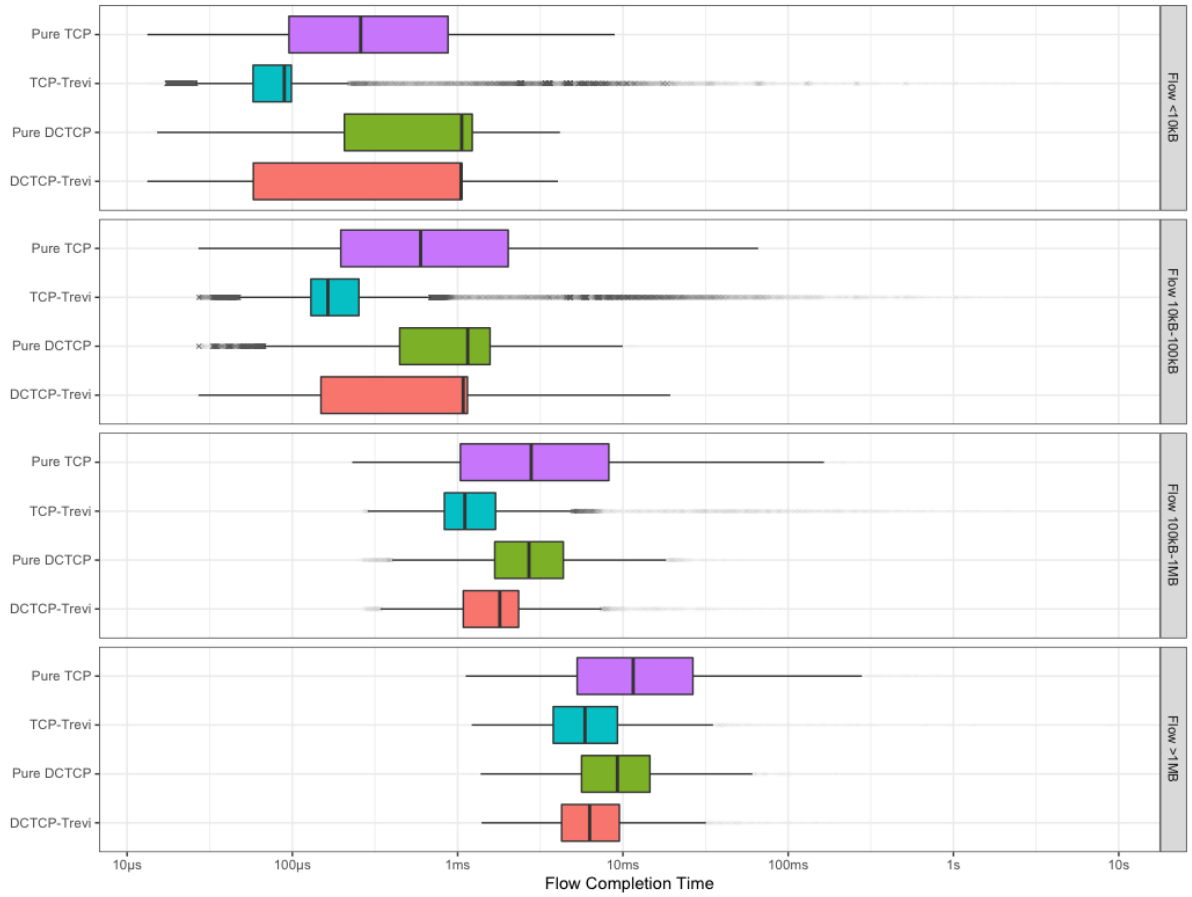

Figure 4.6: Comparing the foreground FCTs for the 4 scenarios with much shorter inter-arrival times for storage traffic (high storage traffic matrix).

FCT.

The really intriguing results are for TCP with Trevi. Here, we see that the foreground traffic for TCP generally performs much better. However, it has vastly more outliers with some flows taking over a second to complete. If time allowed this would be an interesting result to analyse in more detail. I suspect this may be partially due to the limitations of ns2's ingress queue model, which could be causing delays before traffic even enters the network. Alternatively, it could be that by allowing preferential drop, TCP flows within each pod are getting much better completion times at the expense of the flows that have to traverse the core switches. That would explain the contradiction of seeing large numbers of outliers with much improved average FCTs. Finally, it may even be that the increased network capacity for the TCP traffic, coupled with the all-to-all traffic pattern is causing TCP-incast for some flows.

4.7.5 Discussion

In section 4.5, I made several claims about the likely benefits of Trevi. In particular I claimed that it addressed several perceived issues that exist with TCP-based storage

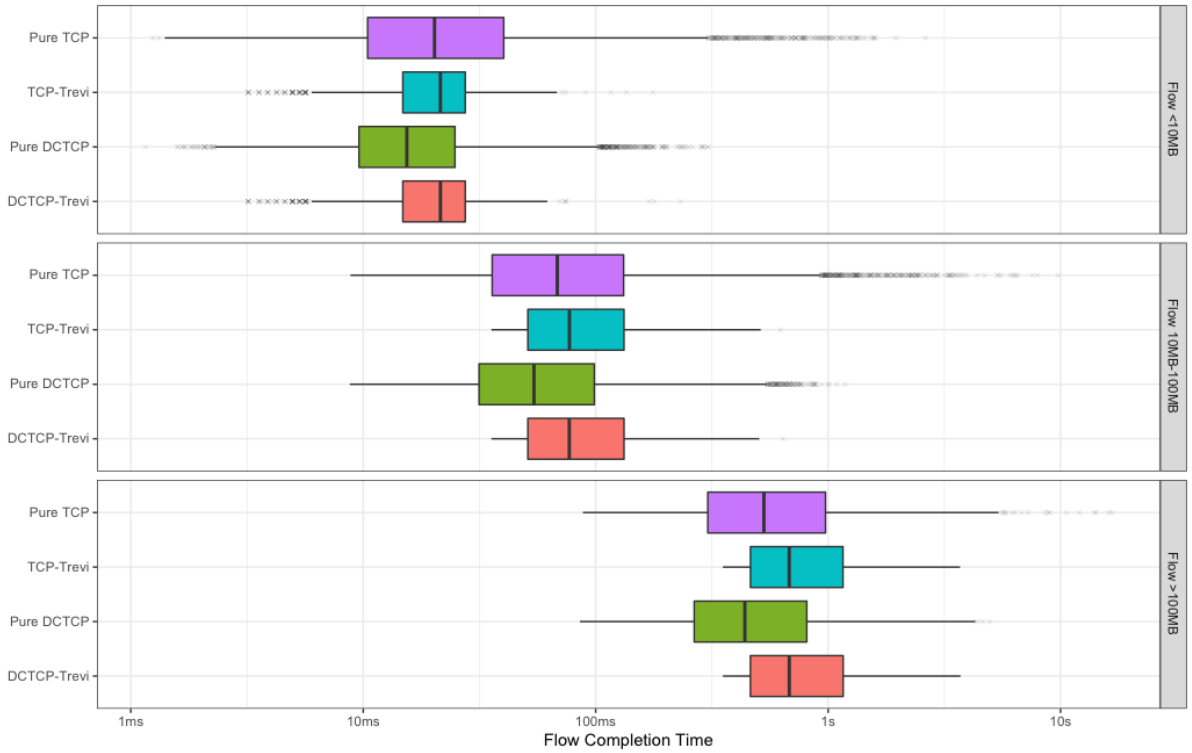

Figure 4.7: Comparing the storage FCTs for the 4 scenarios with much shorter inter-arrival times for storage traffic (high storage traffic matrix).

systems. I can now examine these claims with reference to the results from my simulations. Many of the claims are specific to the actual architecure of Trevi—for instance its use of multicast to give improved use of the multiple available paths and to multisource data and its reliance on standard cheap switch hardware. Two specific claims that I made were relating to TCP incast and Trevi's ability to improve the performance of foreground flows without too adversely affecting storage traffic.

4.7.5.1 Trevi and TCP incast

TCP incast has always proved elusive in real world data sets. Undoubtedly there are occasions when it happens. However, it can be hard to prove its existence due to the limitations on capturing packet-level network data. Within the simulation setup above, incast would manifest itself by a sudden increase in the number of flows that take significantly longer to complete. In other words it would show up as a spike in the number of outliers in the box-whisker plots. While it is annoying for storage flows, incast can be hugely damaging for foreground flows. Therefore what we are looking for is cases where Trevi reduces the number and severity of the outliers for the foreground traffic.

4.7.5.2 Improved performance for foreground traffic

My results show that Trevi has a marked impact on the performance of foreground traffic, both with TCP and DCTCP. As mentioned above, the average performance of TCP is almost a full order of magnitude better, and the majority of flows complete in a much shorter time. However, there are a large number of outliers which I posit may be a result of incast or may be a result of an exacerbated unfairness penalty relating to where in the network the flow origin is. DCTCP shows a much clearer general improvement, especially for very short flows, although the IQR widens somewhat. This is further evidence that relying on a single transport protocol for all data centre traffic can lead to inefficiency.

4.7.5.3 Impact on storage flows

In section 4.5, I made several claims about the likely impact of Trevi on storage traffic. Here I re-examine those claims and see if they stand up in light of my simulations.

TCP Incast The results are ambiguous about the impact of Trevi on incast. Storage traffic itself will no longer be a source of incast, but the results for the TCP foreground traffic suggest that under some conditions Trevi could exacerbate incast by being too efficient at leaving space in network queues.

Wasting network resources in exchange for resilience The strict priorority queueing and lack of explicit retransmission allow Trevi traffic to act as a scavenger class. This means that you no longer need to provide extra network capacity for the storage traffic.

Expensive switches to prevent packet loss These results show that simple priority queueing is all that Trevi requires in order to provide improved performance for foreground traffic. Coupled with running short queues this means cheap Layer 2 switches with small amounts of memory will suffice.

Lack of parallelism when multiple replicas exist. As mentioned above, I was unable to get multicast to work and so these results are not sufficient to draw any conclusions about the impact on parallelism.

No (or basic) support for multipath transport These results use ECMP to spread the traffic across the multiple available routes. However, as discussed in 2.2, ECMP is only effective where flows are roughly comparable. In this simulation, the number, distribution, and size of flows is such that ECMP probably does suffice, and so these results show the impact of using multiple paths. However, they do not explore what happens if the foreground traffic is using a multipath transport such as MPTCP.

Overall I believe my results show that Trevi has the potential to work really well as a storage transport. However I think full scale tests of a real-world system are needed before one can say that for certain.

4.8 Realistic use cases for Trevi

As can be seen from the discussions above, Trevi works best with larger files. With a small file not only is it harder to generate an efficient set of codewords, but also the overhead is likely to be much higher. With small files there is also a relativey greater sensitivity to packet loss. Conseqently, Trevi will be less effective for small files. Heuristically it is hard to say exactly how large a file needs to be before it will benefit from Trevi, but it is likely to be once it exceeds a few Megabytes.

Also Trevi relies on being able to create codewords across an entire storage blob prior to sending. Thus Trevi can only work with static files—that is, files that do not alter during the life of the file transfer. By contrast, block-based storage systems, be they TCP-based or not, can cope with dynamically changing storage files.

The ideal use case for Trevi is where you have items in stable storage that you wish to distribute across the data centre. A good example of this is static databases that need to be distributed to worker nodes and virtual machine images that will only change occasionally, and which are only needed when nodes are re-purposed.

4.8.1 Using Trevi for distributing images

In any data centre there is a frequent need to distribute virtual machine images to Hypervisors. Whilst sometimes these images may be stored locally on a hypervisor, more often they will be stored remotely. In the OnApp architecture, such images are stored centrally. In Openstack such images are stored in the Glance service which is also a central service. This will espcially be the case when there are large numbers of different images, for instance in a multi-tenant public cloud where there could be hundreds of different images which need to be distributed[6]. These images are completely static and are also quite large (often hudreds of Megabytes, or even Gigabytes for Windows images). These two features make them ideally suited for Trevi. In large multi-tenant data centres, such VM image transfers are extremely common, although empirical data on exactly how common is hard to find.

[6]As a case in point, OnApp currently offers a library of over well over 500 standard Virtual Server templates to customers [112].

4.8.2 Using Trevi for Map-Reduce clusters

Map-Reduce relies on sending queries to multiple nodes, and receiving small responses back from all of them. However, in parallel with this, there is an ongoing need to refresh and maintain the data that Map-Reduce works with. Referring back to the traffic traces in the DCTCP paper [5], one of the key components of the background traffic was "large, 1MB to 50MB, update flows that copy fresh data to the workers." This sort of flow is perfect for Trevi, and is likely to be a significant proportion of the storage traffic in any data centre running a partition-aggregate workload.

4.8.3 Cases where Trevi is unsuitable

Some forms of storage traffic would be less suitable for Trevi. For instance real-time or incremental backup would only work if there were a large buffer that would need to be filled, encoded, transmitted and then decoded. Of course, it is possible to consider some form of hybrid system. When a backup image is initially created then it could be sent via Trevi. Incremental changes to that image would then be sent using a more conventional form of transport. Trevi will also work less well when the files are too small—in such cases the overhead of encoding and decoding the file might well outweigh any benefits.

4.9 Conclusions

In this chapter I presented my work on Trevi, a new storage architecture based on fountain coding. Trevi overcomes the limitations present in all storage systems that are based on TCP. I described our strawman design which highlighted the main features of our approach, and also presented an initial design for a receiver-driven flow and congestion control mechanism that can better utilise storage and network resources in a data centre storage network. I used ns2 simulations to show that Trevi seems to perform well for storage traffic while also improving the performance of the foreground traffic. Further work is needed to properly explore the impact of Trevi's sending rate and also to explore the claim that multicast will further improve performance, especially for storage reads.

Chapter 5

Multi-tenant data centres

Much of the research on data centre networking has concentrated on large single tenant data centres such as those operated by Facebook and Google. However, these only represent a small proportion of data centres. Most data centres are much smaller and are owned/operated by companies and universities. Others are multi-tenant data centres ranging from general purpose ones, such as Amazon Web Services and Microsoft Azure, to more application-oriented ones, such as Rackspace or Google Compute Engine. This chapter concentrates on latency control in multi-tenant, general purpose data centres (often also referred to as "cloud" data centres).

This chapter will show how the requirements of multi-tenant data centres distinguish them from single tenant data centres. It will show that tenants can be allowed to use any transport protocol they like and still receive guaranteed network performance by the use of simple policers to shape their network traffic. It will also give results showing that in a multi-tenant environment such an approach:

- Gives better latency performance than dedicated transport layer techniques like DCTCP [5] and HULL [7].
- Provides appropriate network performance for a variety of realistic workloads.
- Makes better use of resources within the data centre, maximising the number of future tenants that can be accepted.

I was involved in the design and testing of Silo while doing an internship at Microsoft Research, Cambridge in late 2012. My major contribution was in the design of the policer mechanisms and the discrete event simulations (both the micro benchmarks and the large-scale ns2 simulations) as well as in identifying network calculus as a solution for calculating the impact of VM placements on the network. This work was published in SIGCOMM 2015 [79] and I was heavily involved in writing that paper. I include details of the full system to put the work in the proper context. Some of the figures in the chapter were taken from our Microsoft Research Technical Report [78].

5.1 Latency sensitive applications in the cloud

Multi-tenant data centres differ significantly from single tenant data centres. For a start you can no longer trust the end users—not only might they be competing for a greater share of the resources, they might also be direct business rivals of other tenants. As such all tenants should be isolated from one another to prevent unwanted interference. Multi tenant data centres are usually virtualised with multiple VMs residing on a single server. This can also be the case in a single tenant data centre, but in a cloud data centre each VM may be running its own custom OS and application set. Consequently, you cannot rely on the end-host transport protocol to guarantee low latency.

As discussed in Chapter 3, predictable message latency is important for many web applications and essential for OLDI applications. Achieving predictable latency can be done in several ways [53, 7]. However, these techniques do not translate well to the world of multi-tenant data centres. Currently operators offer flexible options for computation and storage, but networking is something of a poor relative with most operators offering no guarantees on performance. This limits the nature of the applications that can be run on such data centres unless you are a large enough client that the operator can provide you with dedicated racks and pods, becoming more like a traditional server farm than a cloud data centre. Various solutions have been proposed that treat the network as a virtualised resource [10, 12]. However, these mechanisms are not able to give any latency guarantees.

As explained in Section 3.3.1, OLDI applications have particularly strict latency requirements. Running such an application on a multi-tenant data centre is nearly impossible today because tenants have no control over network performance. Even if a tenant uses classic tricks to choose an optimal set of nodes [118] they still have no control over network bandwidth and latency.

Silo is designed to address exactly this problem. The starting point for the design was the question "What would be needed to allow a Google-like web search application to run on a multi-tenant data centre?". Silo extends Oktos [12] and allows data centre operators to give tenants guarantees for maximum delay as well as minimum bandwidth.

The key insight behind Silo's design is that the VM bandwidth guarantees needed for isolating tenants also make it easier to bound end-to-end packet delay. In turn that bounds the message delay and hence makes it easier to offer latency guarantees. Silo uses this insight when admitting tenants and placing their VMs across the data centre such that their guarantees can be met. The VM placement relies on strictly pacing traffic at the guaranteed rate. This then allows us to use network calculus to yield a deterministic upper bound for network queuing even across multiple network hops [84, 33, 34]. In order to make efficient use of network resources, Silo also allows tenants to request a burst allowance. This allows tenants with highly variable traffic patterns to still get strict latency guarantees with a lower bandwidth guarantee.

Silo's main contributions are:

- Identifying the network guarantees necessary for predictable message latency and identifying mechanisms that can support these.
- The design of a novel admission control and VM placement algorithm[1] that uses network calculus to efficiently map tenants' multi-dimensional network guarantees to two simple constraints regarding switch queues.
- The design of an efficient software packet pacer for fine-grained rate limiting of VM traffic[2]. It couples I/O batching with "void" packets which are forwarded by the NIC but dropped by the first hop switch that are used to precisely space out actual packets. The pacer achieves sub-microsecond pacing with extremely low CPU overhead.

An important feature of Silo's design is ease of deployment. It uses standard features of network switches, requires no modification of tenant applications and allows the use of standard guest OSes. Testbed experiments and packet level simulations show that Silo gives predictable low latency even in the presence of competing traffic. It improves on DCTCP and HULL by a factor of 22 at the 99th percentile. Silo's placement algorithm can even improve network utilisation and overall cloud utilisation compared to a naïve system.

5.2 Network requirements

In the rest of this chapter, a message is defined as one or more packets of application data sent across the network making up a single information exchange. Message latency is the time to send a complete message whilst delay is used to talk about packet delay. As explained in section 3.2, the latency for a message comprises the time to transmit its packets into the network and the time for the last packet to propagate to the destination. This simple model excludes end host stack delay.

$$\begin{aligned} Msg.Latency &\approx Transmission\ delay + In\text{-}network\ delay \\ &\approx (Message\ size/Capacity) + In\text{-}network\ delay \end{aligned} \tag{5.1}$$

Thus to guarantee maximum message latency we need:

[1]This aspect of the work was done mainly by one of my intern colleagues, Justine Sherry, a PhD student at Berkeley.

[2]This aspect of the work was performed by Keon Jang, then a Post Doc at Microsoft Research

Table 5.1: Showing how the percentage of late messages changes with burst size and bandwidth guarantee. The shading indicates the relative performance.

	Bandwidth Guarantee					
Burst Allowance	B	$1.4xB$	$1.8xB$	$2.2xB$	$2.6xB$	$3xB$
1	99%	77%	55%	45%	38%	33%
3	99%	22%	8%	3.6%	1.9%	1.1%
5	99%	6.1%	0.9%	0.2%	0%	0%
7	99%	1.4%	0%	0%	0%	0%
9	98%	0.4%	0%	0%	0%	0%

Requirement 1: Guaranteed network bandwidth. This bounds the transmission delay component of message latency.

Requirement 2: Guaranteed packet delay. This bounds the in-network delay component of message latency.

5.2.1 Handling bursty traffic

Many data centre applications have bursty workloads, i.e. their instantaneous bandwidth requirement significantly exceeds the average bandwidth. To illustrate this, imagine a simple application that sends messages from one VM to another across a data centre. Messages of size M are generated with exponential inter-arrival times. The average bandwidth of the network is B and the packet delay guarantee is d. Thus from 5.1 the maximum message delay D is $D = M/B + d$. In the first row in Table 5.1, you can see that even if the application is guaranteed to receive its average bandwidth, 99% of messages exceed the maximum message delay. This is because messages arrive in a non-uniform manner. Increasing the bandwidth guarantee helps, but even at 3x the average bandwidth 33% of messages fail to meet their guarantee.

To accommodate such burstiness, Silo optionally offers a burst allowance for tenant VMs. Specifically, a VM that has not been using its guaranteed bandwidth in the past is allowed to send a small number of messages at a higher rate. Table 5.1 shows that as we increase the sending VM's burst allowance, the percentage of late messages drops sharply; with a burst of 7 messages and 1.8x the average bandwidth, only 0.09% messages are late. Thus, the third requirement for guaranteed message latency is:

Requirement 3: Guaranteed burst allowance. Bursty workload applications need to be able to send short traffic bursts at a higher rate.

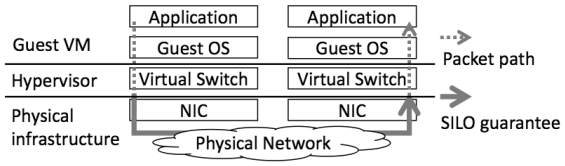

Figure 5.1: Silo only guarantees the network delay

5.3 Scope and design insights

Of the three network requirements, guaranteeing packet delay is particularly challenging because every node and link of the end-to-end path adds delay. The scope of Silo's guarantees is important.

5.3.1 Scope

In virtualised data centres, the end-to-end path of packets comprises many layers. Figure 5.1 shows this path; at the sender packets pass down through the guest OS network stack, the hypervisor and the NIC before being sent onto the wire. Broadly, a packet's total delay comprises two components: *end-host delay* (shown with dashed line), and *network delay* (shown with a solid line). The latter component includes delay at the NIC and at network switches. Since Silo targets IaaS cloud settings where tenants can deploy arbitrary OSes, we restricted our focus to the part of the end-to-end path controlled by the cloud provider. Thus, Silo guarantees network delay + virtualisation delay, i.e. the delay between source and destination hypervisors.

5.3.2 Guaranteeing network delay

Network delay comprises the propagation, forwarding and queuing delay across NICs and the network. In data centres, physical links have a short length and high capacity, so the propagation and forwarding delay is negligible, and queuing delay dominates. This holds even for full bisection networks[52, 3]. TCP-like protocols drive the network to congestion by filling up queues, leading to high and variable network delay. While recent proposals like HULL [7] and pFabric [8] reduce network queuing, they do not ensure guarantees for packet delay (or bandwidth).

In Silo we have adopted a different tactic to bound queuing delay. As discussed above and in Chapter 3, cloud applications require guaranteed network bandwidth for predictable message latency. Ensuring that a VM's traffic is paced at its guaranteed rate ensures a deterministic upper bound for network queuing [34, 84]. For example, consider n flows bottlenecked at a network link. Each flow is guaranteed some bandwidth and is allowed to burst one packet at a time. Assuming the total bandwidth guaranteed across all flows

is less than the link capacity, the maximum queue build up at the link is n packets. This happens when a packet for each flow arrives at the link at exactly the same time. Section 5.4.2 builds upon this simple observation, using network calculus to quantify the maximum queuing across a multi-hop network.

5.3.3 Fine-grained pacing

Our queuing delay analysis assumes that VM traffic is in strict conformance to its guarantees. Thus, end hosts need to control the rate and burstiness of their traffic at a packet-level timescale. Today's software pacers are typically inaccurate and do not scale with the number of flows [132]. The problem is exacerbated by the fact that, to achieve good forwarding performance, network stacks rely on aggressive batching of packets sent to the NIC. This further contravenes the fine-grained pacing requirement.

The obvious solution of pacing at the NIC itself is impractical because NICs only offer small number of rate limited queues, and flows that share the same hardware queue can suffer from head of line blocking. SENIC proposes a hardware and software hybrid approach to achieve scalable and accurate pacing [132]. Instead, we devise a software-only pacing mechanism that uses void packets to precisely control packet gap while retaining I/O batching to handle traffic at 10 Gbps (see Section 5.4.3.1).

5.4 Silo design

Silo places virtual machines (VMs) with guarantees for network bandwidth, packet delay and burst allowance. It relies on two components–a VM placement manager with visibility of the data centre topology and tenants' guarantees, and a packet pacer in the hypervisor at each server. The placement manager admits tenants and places their VMs across the data centre such that their guarantees can be met (Section 5.4.2), and configures the pacers with VM guarantees. The pacers coordinate with each other and dynamically determine the rate limit for individual VMs, thus ensuring that VM traffic conforms to their bandwidth and burst guarantees (see Section 5.4.3).

5.4.1 Silo's network guarantees

With Silo, tenants can imagine their VMs as being connected by a private virtual network, as shown in Figure 5.2. A virtual link of capacity B and propagation delay d connects each VM to a virtual switch. Each VM's traffic is shaped by a virtual token bucket with average bandwidth B and size S. The network capabilities of a VM are thus captured using three parameters, $\{B,\ S,\ d\}$:-

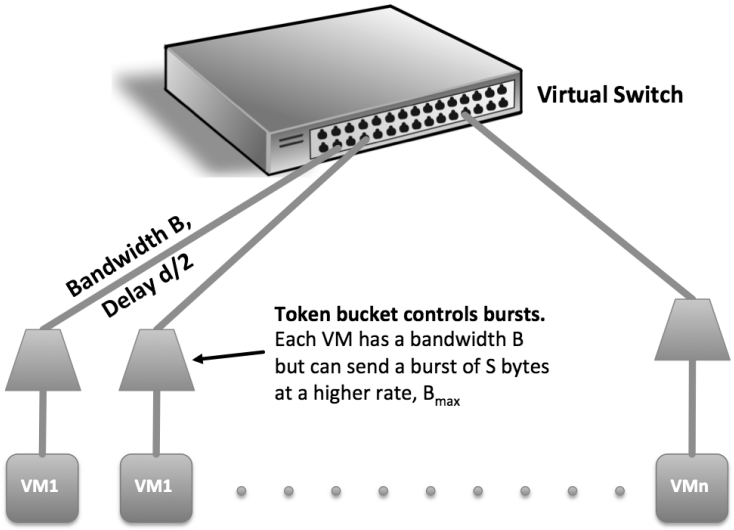

Figure 5.2: Each tenant sees a virtual network

i. a VM can send and receive traffic at a maximum rate of B Mbps,

ii. a VM that has under-utilised its bandwidth guarantee is allowed to send a burst of at most S bytes,

iii. a bandwidth-compliant packet is guaranteed to be delivered, from the sending to the receiving NIC, within d µs.

Just as today's cloud providers offer a few classes of VMs (small, medium, large, etc.), we expect providers will offer a few classes of network guarantees. Tenants can also leverage tools like Cicada [86] to automatically determine their guarantees. Some tenants may only need bandwidth guarantees; for example, a tenant running a data-analytics job. In Section 5.4.4, we show that Silo can also accommodate tenants without any network guarantees and ensure they co-exist with tenants with guarantees.

5.4.1.1 Guarantee semantics

The precise semantics of the network guarantees represent a trade-off between how useful they are for tenants and how practical they are for providers. We have chosen our guarantees to balance this trade-off. As with past proposal [12, 128, 38], our VM bandwidth guarantee follows the hose model, i.e. the bandwidth for a flow is limited by the guarantee of both the sender and receiver VM. So if a tenant's VMs are guaranteed bandwidth B, and N VMs send traffic to the same destination VM, each sender would achieve a bandwidth of B/N (since the destination VM becomes the bottleneck). By contrast, a VM's burst guarantee is not limited by the destination; all N VMs are allowed to send

a simultaneous burst to the same destination. This is motivated by the fact that applications that need to burst (like OLDI) often employ a partition aggregate workflow that results in an all-to-one traffic pattern [5].

However, allowing VMs to send traffic bursts can result in high and variable packet delay for VMs of other tenants. Synchronised bursts can even overflow switch buffers and cause packet loss. While Silo carefully places VMs to ensure switch buffers can absorb the bursts, we also control the maximum bandwidth (B_{max}) at which a burst is sent.

5.4.1.2 Calculating the latency guarantee

Silo's tuneable network settings allow tenants to determine their maximum message latency. Consider a VM, that has not used up its burst allowance sending a message of size $M(\leq S)$ bytes. The message is guaranteed to be delivered to its destination in less than $M/B_{max} + d$ seconds. If $M > S$ then the latency is less than $S/B_{max} + (M - S)/B + d$ seconds.

5.4.2 VM placement

Given a tenant request, Silo's placement manager performs admission control, and places its VMs at servers in the data centre such their network guarantees can be met. If the guarantees cannot be met, the request is rejected.

5.4.2.1 Placement overview

Placement of VMs in today's data centres typically focuses on non-network resources like CPU cores and memory. Recent efforts propose algorithms to place VMs such that their bandwidth guarantees can also be met [56, 12]. Silo expands VM network guarantees to include packet delay and burst allowance. With only bandwidth guarantees, the placement constraint at a switch port only involves flows traversing the port—the sum of the bandwidth guarantees for these flows should not exceed the port's capacity. However, queuing delay at a switch port is determined not only by the flows traversing the port, but also by other flows that these flows interact with along their respective paths.

The main insight behind our approach is that each VM's bandwidth guarantee yields an upper bound for the rate at which it can send traffic. This allows us to quantify the queue bound for any switch port, i.e. the maximum queuing delay that can occur at the port. Further, we can also determine a port's queue capacity, the maximum possible queue delay before packets are dropped. For example, a 10Gbps port with a 312KB buffer has a $\approx$250 µs queue capacity. The key novelty in the placement algorithm is mapping multi-dimensional network guarantees to two simple queuing constraints at switches. To

ensure the network has enough capacity to accommodate the bandwidth guarantees of VMs and absorb all bursts, we need to ensure that at all switch ports, the queue bound does not exceed queue capacity. This is the first constraint. As we explain later, packet delay guarantees lead to the second queuing constraint. These constraints then dictate the placement of VMs.

In the following sections, we detail our placement algorithm. We assume a multi-rooted tree-like network topology prevalent in many of today's data centres (see Background Section 2.1.1). Such topologies are hierarchical; servers are arranged in racks that are in turn, grouped into pods. Each server has a number of slots where VMs can be placed.

5.4.2.2 Queue bounds

We begin by describing how we use basic network calculus concepts [84, 89] to determine the queue bounds for network switches. This serves as a building block for Silo's placement algorithm.

Source Characterisation.
Traffic from a VM with bandwidth guarantee B and burst size S is described by a arrival curve $A(t) = Bt + S$, which provides an upper bound for traffic generated over a period of time. We will refer to this curve as $A_{B,S}$. This arrival curve is shown in Figure 5.3(a) and assumes that the VM can send a burst of S bytes instantaneously. While we use this simple function for exposition, our implementation uses a more involved arrival curve (labelled $A(t)$ in the figure) that captures the fact that a VM's burst rate is limited to B_{max}.

Calculating queue bounds.
Arrival curves can be used to determine queue bounds for network switches. Just as traffic arriving at a switch is characterised by its arrival curve, each switch port is associated with a service curve that characterises the rate at which it can serve traffic. Figure 5.3(b) illustrates how these two functions can be used to calculate the maximum queuing at the port or its queue bound. At time $t = p$, the aggregate traffic that the switch can serve exceeds the aggregate traffic that can arrive. This means that at some point during the interval $(0, p]$ the queue must have emptied at least once. The horizontal distance between the curves is the time for which packets are queued. Hence, the port's queue bound is q, the maximum horizontal distance between the curves (i.e., the largest q such that $S(t) = A(tq)$).

This allows us to calculate the queue bound at a switch directly receiving traffic from a VM. Below we describe how arrival curves can be added (when traffic from different VMs merges at a switch) and propagated across switches to determine the queuing at any network switch.

Adding arrival curves.

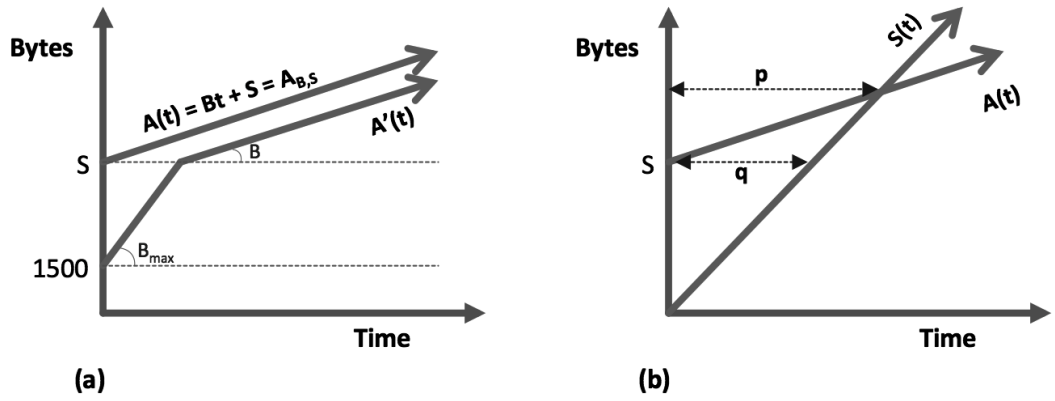

Figure 5.3: Network Calculus: (a) shows two arrival curves (b) shows an arrival curve and a queue's service curve

Arrival curves for VMs can be combined to generate an aggregate arrival curve. For example, adding arrival curves $A_{B1,S1}$ and $A_{B2,S2}$ yields $A_{B1+B2,S1+S2}$. However, as explained below, the semantics of our guarantees allow us to generate a tighter arrival curve when adding curves for VMs belonging to the same tenant.

Consider a tenant with N VMs, each with an average bandwidth B and burst allowance S. The arrival curve for each VM's traffic is $A_{B,S}$. Imagine a network link that connects the tenant's VMs such that m VMs are on the left of the link and the remaining $(N-m)$ are on the right. We want to add the m arrival curves for the VMs on the left to generate an aggregate curve for all traffic traversing the link from left to right. Our choice of hose-model bandwidth guarantees implies that the total bandwidth guaranteed for the tenant across the link is

$min(m, N-m) * B$ [12].

By contrast, burst allowances are not destination limited, so the maximum traffic burst across the link from left to right is $m * S$ bytes. Thus, instead of $A_{mB,mS}$, the aggregate arrival curve is actually $A_{min(m,Nm)*B,mS}$.

Propagating arrival curves.

After traffic egresses a switch, it may no longer be shaped according to the properties it arrived at the switch with. For example, consider Figure 5.4: flow f_1 has a sending rate of $C/2$ and flow f_2 has a sending rate of $C/4$ (link capacity is C). Both have a burst size of one packet so f_1's arrival function is $A_{C/2,1}$ and f2's is $A_{C/4,1}$. At switch $S1$, the first packet of both f_1 and f_2 arrive simultaneously; the packet from f_2 is served first followed by the packet from f_1. Immediately after this, a packet from f_1 arrives and is served. This sequence then repeats itself. Thus, f_1's packets are bunched due to queuing at switch $S1$ such that after leaving the switch, f_1's arrival function is $A_{C/2,2}$. Note that a flow's average bandwidth cannot change with queuing, only the burst size is impacted.

Kurose [84] proved an upper bound for the burst size of traffic egressing a switch. Consider

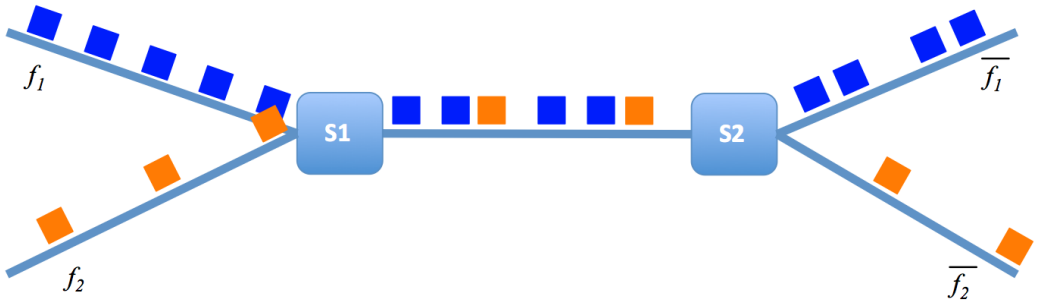

Figure 5.4: Switch S1 causes the packets in flow f_1 to bunch

the value p from Figure 5.3—the maximum interval over which the queue must be emptied at least once. In the worst case, every packet sent by a VM over the interval $[0, p]$ may be bunched together and forwarded as one burst. However, this analysis means that the arrival curve for egress traffic depends on a port's p value which, in turn, depends on other flows using the port. To bound the impact of competing traffic on a given VM's traffic, we ensure that the p value on a port can never exceed its queue capacity c.[3] In the worst case, every packet sent by a VM over the interval $[0, c]$ may be forwarded as one burst. Since a VM with arrival curve $A_{B,S}$ can send at most $B * c + S$ bytes in time c, the egress traffic's arrival curve is $A_{B,(B*c+S)}$.

5.4.2.3 Placement algorithm

We have designed a placement algorithm that uses a greedy first-fit heuristic to place VMs on servers. Initially a new tenant's network guarantees are mapped to two simple queuing constraints at switches. These constraints characterise a valid VM placement and guide the design of the algorithm.

Valid placement.
For the tenant's bandwidth guarantees to be met, we must ensure that network links carrying its traffic have sufficient capacity. Further, VMs can send traffic bursts that may temporarily exceed link capacities. Switch buffers need to absorb this excess traffic, and we must ensure that switch buffers never overflow. In combination, these restrictions imply that for each switch port between the tenant's VMs, the maximum queue buildup (queue bound) should be less than the buffer size (queue capacity). Formally, if V is the set of VMs being placed and $Path(i, j)$ is the set of ports between VMs i and j, the first constraint is:-

[3] A port's queue capacity is a static value and is dictated by the size of the port's packet buffer, but can be set to a lower value too.

$$\textit{Queue-bound}_p \leq \textit{Queue-capacity}_p \qquad \forall p \in Path(i,j);\ i,j \in V \tag{5.2}$$

For packet delay guarantees, we must ensure that for each pair of VMs belonging to the new tenant, the sum of queue bounds across the path between them should be less than the delay guarantee. However, a port's queue bound changes as tenants are added and removed which complicates the placement. Instead, we use a port's queue capacity, which always exceeds its queue bound, to check delay guarantees. Thus, for a tenant whose network delay guarantee is d, the second constraint is:-

$$\sum_{p \in Path(i,j)} \textit{Queue-capacity}_p \leq d \qquad \forall i,j\ \in V \tag{5.3}$$

Finding valid placements.
A request can have many valid placements. Given the oversubscribed nature of typical data centre networks, we adopted the following optimisation goal–find the placement that minimises the "level" of network links that may carry the tenant's traffic, thus preserving network capacity for future tenants. Servers represent the lowest level of network hierarchy, followed by racks and pods.

Our algorithm places a tenant's VMs while greedily optimizing this goal. First, we attempt to place all requested VMs in the same server. If the number of VMs exceeds the empty VM slots on the server, we attempt to place all VMs in the same rack. To do this, for each server inside the rack, we use the queuing constraints on the server's uplink switch port to determine the number of VMs that can be placed at the server. If all requested VMs can be accommodated across servers within the rack, the request is accepted. Otherwise we consider the next rack and so on. If the request cannot be placed in a single rack, we attempt to place it in a pod and finally across pods. Pseudocode for the algorithm is shown in Appendix B.

Other constraints.
An important concern when placing VMs in today's data centres is fault tolerance. Our placement algorithm can ensure that a tenant's VMs are placed across some number of fault domains. For example, if each server is treated as a fault domain, we will place the VMs across two or more servers. Beyond this, VM placement may need to account for other goals such as ease of maintenance, reducing VM migrations, etc. Commercial placement managers like Microsoft's Virtual Machine Manager model these as constraints and use multi-dimensional bin packing heuristics to place VMs [90]. Our queuing constraints could be added to these systems reasonably simply.

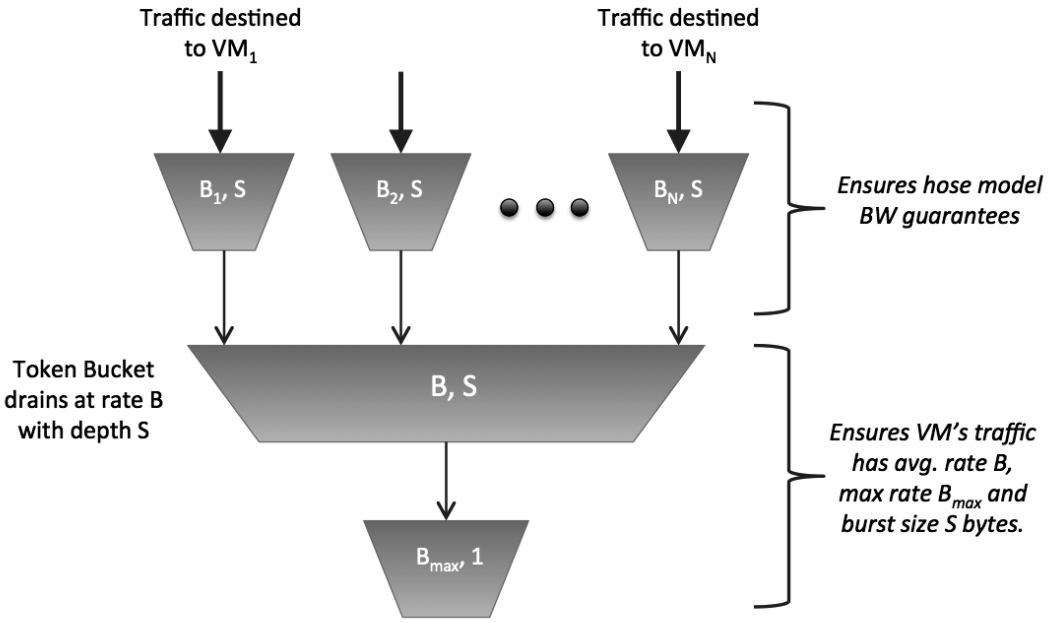

Figure 5.5: Silo uses a hierarchy of token buckets to ensure tenants conform to their traffic specification

5.4.3 End host pacing

Silo's VM placement relies on every tenant's traffic conforming to their bandwidth and burstiness specifications. To achieve this, a pacer at the end host hypervisor paces traffic sent by each VM. Figure 5.5 shows the hierarchy of token buckets used by the pacer to enforce traffic conformance. The bottom-most token bucket ensures a VM's traffic rate can never exceed B_{max}, even when sending a burst. The middle token bucket ensures the average traffic rate is limited to B and the maximum burst size is S bytes. At the top is a set of token buckets, one each for traffic destined to each of the other VMs belonging to the same tenant. These are needed to enforce the hose model semantics of guaranteed bandwidth; i.e. the actual bandwidth guaranteed for traffic between a pair of VMs is constrained by both the sender and the destination. To enforce the hose model, the pacers at the source and destination hypervisor communicate with each other like EyeQ [80]. This coordination determines the rate B_i for the top token buckets in F such that $\sum B_i \leq B$.

5.4.3.1 Packet level pacing

Ideally, the token buckets should be serviced at a per-packet granularity. This precludes the use of I/O batching techniques since today's NICs transmit an entire batch of packets back-to-back [7]. However, disabling I/O batching results in high CPU overhead and

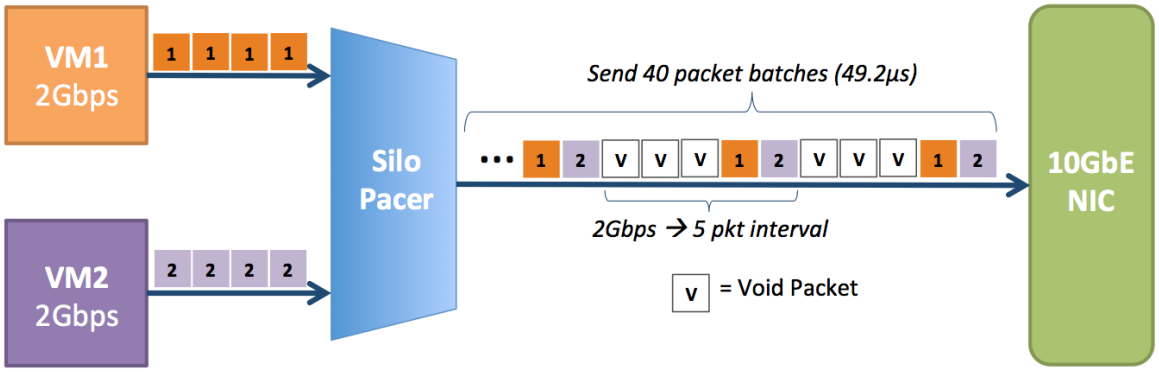

Figure 5.6: Silo uses void packets to pace traffic sent by the NIC

reduces throughput; in experiments with batching disabled (LSO), we cannot even saturate a 10Gbps link (see Figure 3.1). One solution is to implement pacing at the NIC itself [133, 7]. However, this requires hardware support. For ease of deployment, we design a software solution.

In order to retain the high throughput and low overhead offered by I/O batching while still pacing packets at sub-microsecond timescales, we use a novel technique called "void packets" to control the spacing between data packets forwarded by the NIC. A void packet is a packet that will be forwarded by the NIC but discarded by the first switch it encounters. This can be achieved, for example, by setting the packet's destination MAC address the same as the source MAC.

Figure 5.6 illustrates how we use void packets. The link capacity is 10Gbps and VM1 is guaranteed 2Gbps, so every fifth packet sent to the NIC belongs to VM1. In every batch of 40 packets sent to the NIC, 12 are actual data packets, while the other 28 are void packets. While the NIC forwards the entire batch of packets as is, all void packets are dropped by the first hop switch, thus generating a correctly-paced packet stream. The minimum size of a void packet, including the Ethernet frame, is 84 bytes. So, at 10Gbps, we can achieve an inter-packet spacing as low as 68ns.

5.4.4 Tenants without guarantees

Some cloud applications are not network limited, so they do not need any network guarantees. Silo leverages priority forwarding in switches to support tenants without any network guarantees. The majority of Ethernet switches implement IEE802.1p which offers a simple mechanism to mark packets with relative priorities. "Best effort" traffic from such tenants is marked by our pacer as low priority while traffic from tenants with guarantees is higher priority. Thus, such tenants share the residual network capacity. While high network utilisation is not a primary design goal for Silo, such best effort traffic can

improve utilisation without hurting tenants with guarantees.

5.5 Implementation

We have implemented a Silo prototype comprising a VM placement manager and a software pacer implemented as a Windows NDIS filter driver. The pacer driver sits between the virtual switch (vSwitch) and the NIC driver, so we do not require any modification to the NIC driver, applications or the guest OS.

The pacer driver implements token buckets and supports token bucket chaining. We use virtual token buckets, i.e. packets are not drained at an absolute time, rather we timestamp when each packet needs to be sent out. This requires an extra eight bytes on each packet's metadata. The overhead is negligible in comparison to the size of the packet buffer structure: 160 bytes in Windows `NET BUFFER`[4] and 208 bytes in Linux `skb`[5].

At high link rates, I/O batching is essential to keep the CPU overhead low. For accurate rate limiting with I/O batching, we need two key properties. The first is to keep the precise gap between packets within a batch, we achieve this using void packets as described above. The second is to schedule the next batch of packets before the NIC starts to idle. This is essential to guarantee burst allowance but challenging since we want to keep the batch size small so that NIC queuing delay is limited. We borrow the idea of soft-timers [11] and reuse existing interrupts as a timer source. Our pacer does not use a separate timer, but triggers sending the next batch of packets upon receiving a DMA (Direct Memory Access) completion interrupt for transmit packets. We use a batch size of 50 µs when pulling out packets from the token buckets.

5.5.1 Pacer microbenchmarks

We evaluate our pacer implementation in terms of throughput and the CPU overhead. We use physical servers equipped with one Intel X520 10GbE NIC, and two Intel Xeon E5-2665 CPUs (8 cores, 2.4Ghz). Overall, we find that the pacer is able to saturate 10Gbps links with low CPU overhead.

Figure 5.7 shows the CPU usage of the entire system by varying the rate limit imposed by the pacer. The right most bar is CPU usage when the pacer is disabled. LSO is disabled in all cases. The orange solid line represents the number of transmitted packets per second, including void packets. The pacer consumes 0.6 cores to generate only void packets at 10 Gbps. As the actual data rate increases, the overall CPU utilisation goes up to 2.1 cores

[4]See `http://msdn.microsoft.com/en-us/library/windows/hardware/ff556030(v=vs.85).aspx` (accessed February 2018)

[5]See `http://lxr.free-electrons.com/source/include/linux/skbuff.h.` (accessed February 2018)

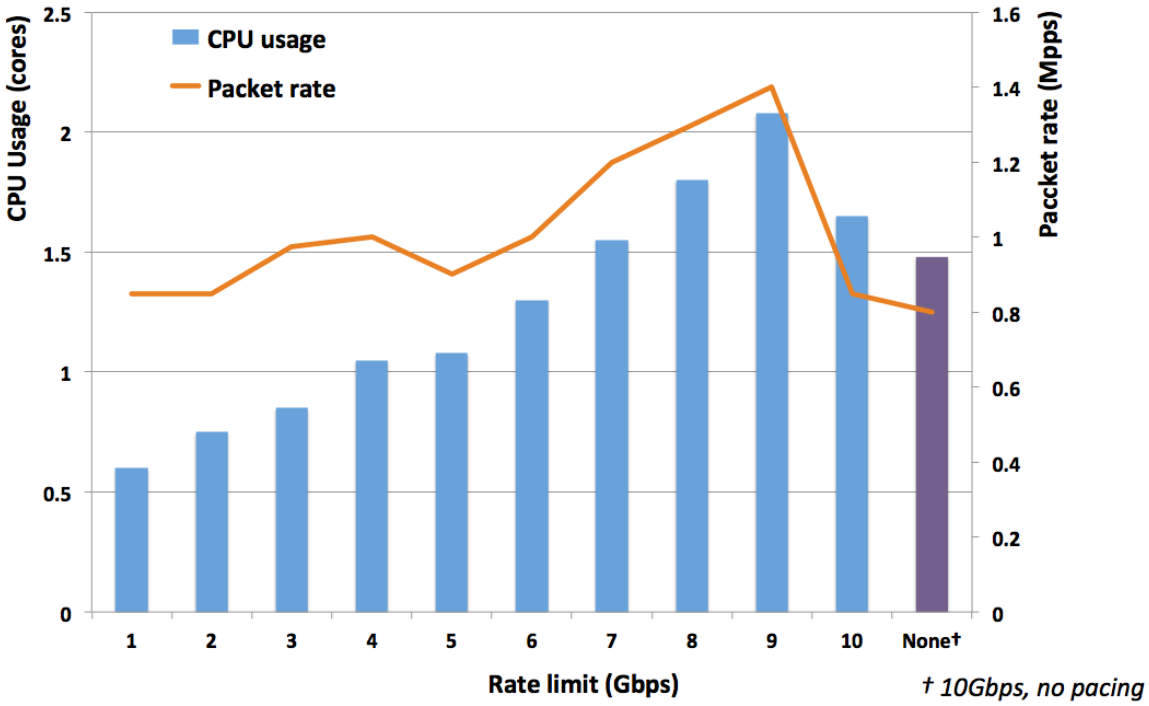

Figure 5.7: Packet rate and CPU usage for the Silo software pacer

worth of CPU cycles at 9 Gbps. The reason is that at 9 Gbps, the pacer needs to put 1/10th of MTU sized packets (150 bytes) between all the data packets, which results in a high packet rate. The graph shows that the overall CPU usage is proportional to the packet rate shown in the orange line. At the full line-rate of 10 Gbps, our pacer incurs a penalty of less than 0.2 cores of extra CPU cycles compared to no pacing. Since void packets are generated only when there is another packet waiting to be sent, the pacer does not incur any extra CPU over-head when the network is idle. Furthermore, void packets do not increase the power consumption at the NIC and switch because most of power is consumed by keeping the link active[6].

In Figure 5.8, we show the throughput for both void packets and data packets. Except at 9 Gbps, the pacer sustains 100% of the link capacity, and achieves an actual data rate of more than 98% of the ideal rate.

5.6 Evaluation

We evaluate Silo across three platforms: a small scale prototype deployment, a medium scale packet-level simulator, and a data centre scale flow-level simulator. The key findings are as follows:-

[6]See `http://www.cisco.com/c/dam/en/us/products/collateral/switches/catalyst-2960-series-switches/cisco_catalyst_switches_green.pdf` (accessed February 2018)

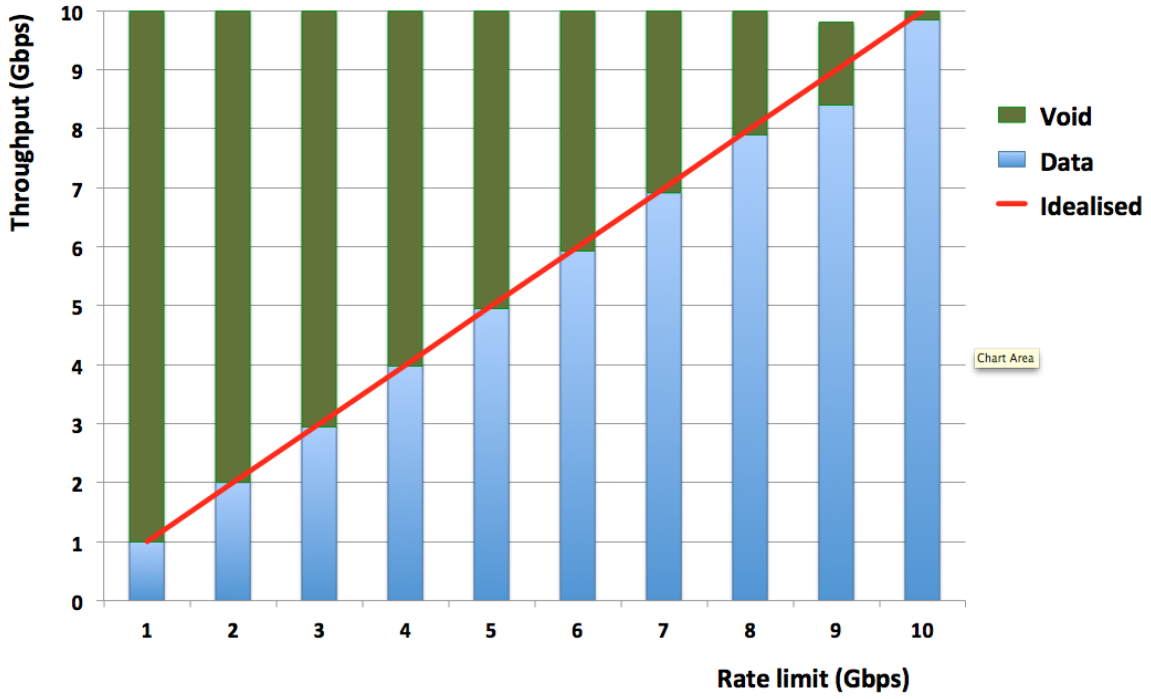

Figure 5.8: Silo's software policer performs well compared with the ideal

i. By using a testbed deployment with a web search workload, we verify that our prototype can offer bandwidth, delay and burstiness guarantees which, in turn, ensures predictable tail message latency.

ii. Through ns2 simulations, we show that Silo improves message latency as compared to state-of-the-art solutions like DCTCP [5], HULL [7], and Oktopus [12]. Unlike Silo, none of these solutions can ensure predictable message latency.

iii. With our flow-level simulator, we characterise the performance of our VM placement algorithm and show that, as compared to locality-aware VM placement, it can actually improve both network and overall cloud utilisation.

5.6.1 Testbed experiments

We deployed our prototype across five physical servers connected to a 10GbE switch. To avoid virtualisation overhead, we emulate each VM as a process with an IP address. We model 16 VMs per server.

Our testbed experiments comprises two tenants, A and B, each with 40 VMs. The tenants are running a delay-sensitive and a bandwidth-sensitive application respectively. We model tenants' workload based on Bing traffic patterns [14]. For Tenant A, one aggregator VM generates messages to all other worker VMs who send 2KB response messages. Average interval between messages is 13 ms. For Tenant B, we generate large messages

with size of 10MB between all pairs of workers at a target rate of 1Gbps per VM. We run the experiments for 10,000 messages per worker for Tenant A, with tenant B running concurrently. All traffic uses TCP sessions. The tenants' guarantees are shown in Table 5.2.

Table 5.2: Tenant network guarantees for the testbed experiments

	Tenant A	Tenant B
Bandwidth (B)	0.051Gbps	1Gbps
Burst length (S)	2kB	1.5kB
Delay guarantee (d)	1	N/A
Burst rate (B_max)	1Gbps	N/A

5.6.1.1 Baseline comparison with Oktopus and TCP

We compare Silo against baseline TCP and Oktopus [12]. TCP provides no guarantees while Oktopus provides bandwidth guarantee but no latency guarantee or burst allowance. For Silo, the tenants' guarantees can be used to estimate the maximum message latency. In this experiment, the estimated maximum message latency for tenant A is 2.1ms.

5.6.1.2 Uniform message arrivals

We begin with a simple experiment where Tenant A generates messages that are uniformly spaced in time. We measure latency at the application which includes end-host stack delay. Since Silo does not control stack delay and only guarantees NIC-to-NIC delay the worst-case latency at the application-level is not bounded. Consequently, we focus on the 99th percentile message latency.

Figure 5.9 shows the 99th-percentile message latency for Tenant A with and without Tenant B. It shows that with Silo, the message latency is within the estimate, even when there is competing traffic from Tenant B. With Oktopus, Tenant A gets a bandwidth guarantee but is not allowed to burst, so the message latency is 3x the Silo estimate. TCP performs well when Tenant A runs in isolation. However, when there is competing traffic from Tenant B, Tenant A's latency suffers due to queuing at the switch. Tenant A's message latency is as high as 5.2 ms, 2.5x the estimate. This shows that our prototype can ensure predictable tail message latency.

5.6.1.3 Bursty message arrivals

We now consider a more realistic scenario with a bursty workload for tenant A; the inter-arrival time between its messages is based on the DCTCP workload [5][7]. As shown in

[7]We fit the inter arrival time distribution in [5] to an exponential distribution.

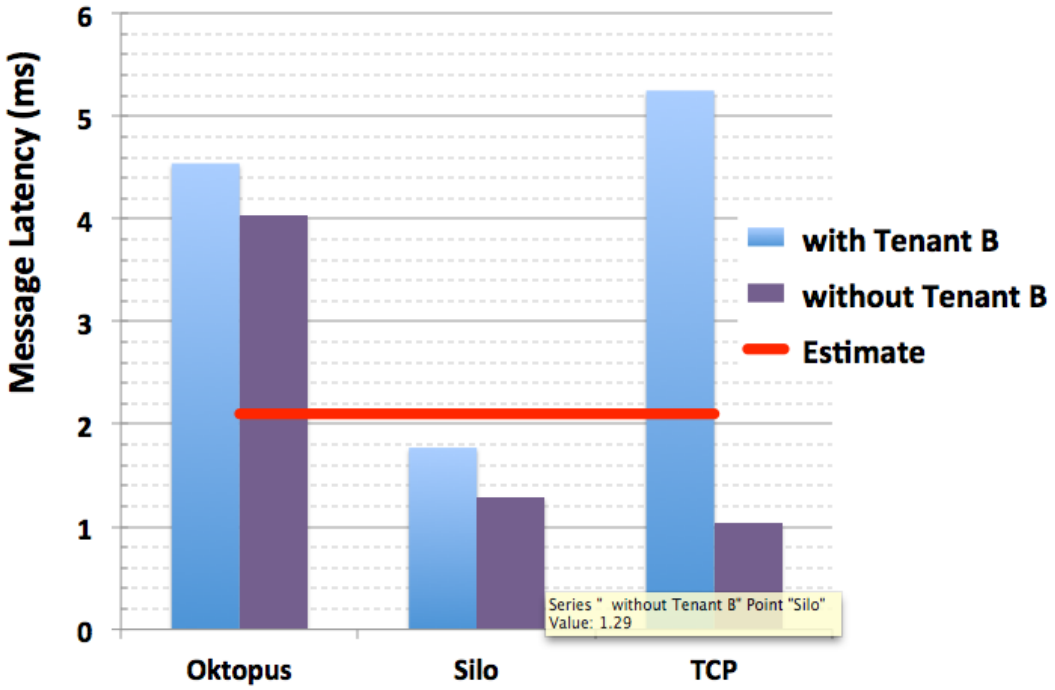

Figure 5.9: 99th-percentile message latency for delay sensitive application (with and without competing bandwidth-sensitive application)

5.2, tenants with non-uniform workloads can still ensure guaranteed message latency by asking for overprovisioned bandwidth (i.e. the tenant's bandwidth guarantee is higher than its average bandwidth requirement) and a burst allowance.

To evaluate this, we vary tenant A's bandwidth over-provisioning ratio and burst allowance, and show its 99th-percentile message latency in Figure 5.10. The dashed line is the estimate for the maximum message latency. With no bandwidth overprovisioning, the message latency is much higher than the estimate. This is because of the exponential arrival pattern; when messages arrive too close to each other, later messages have to wait. When messages arrive too far from each other, reserved bandwidth is wasted. However, when bandwidth is overprovisioned by 2x and with a burst allowance of 8KB, latency is well under the estimate. We also verified that the same over-provisioning ratio leads to good performance across different bandwidth requirements or message sizes.

Overprovisioning of bandwidth naturally leads to network under utilisation. However, overprovisioning is only necessary for bursty small-message applications like OLDI. Such applications have low average bandwidth requirements, so some overprovisioning seems affordable. Applications that use large messages do not need overprovisioning.

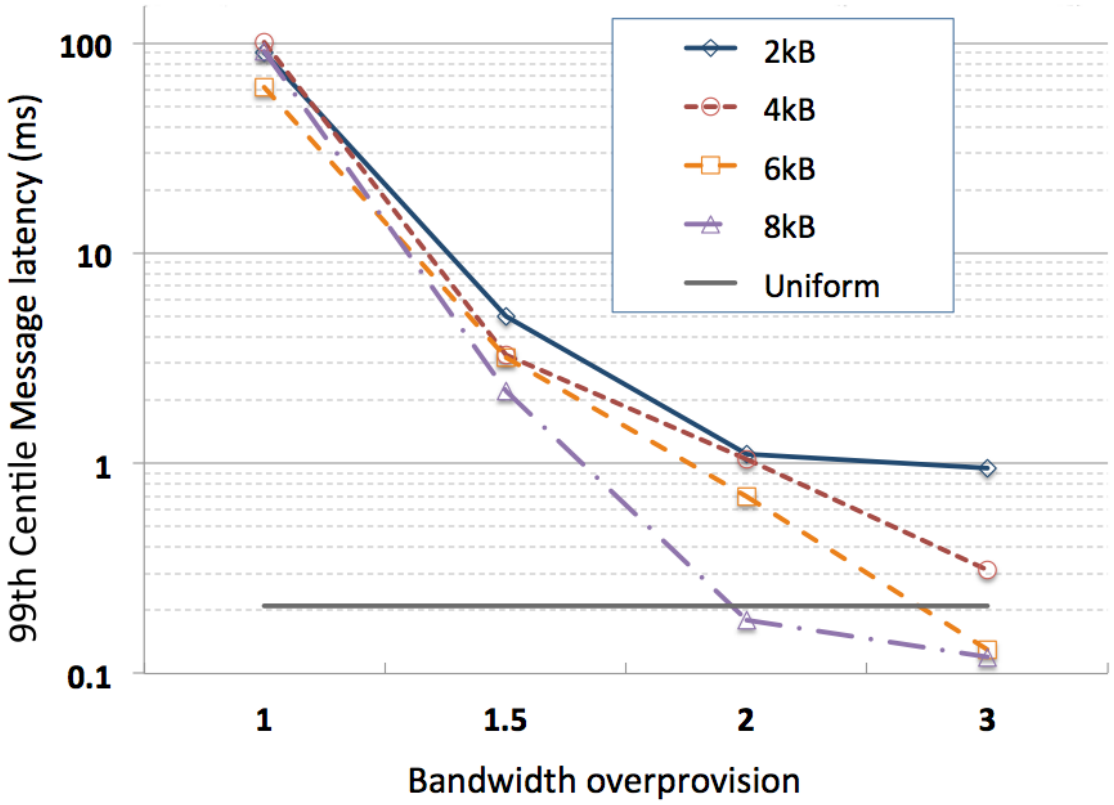

Figure 5.10: 99th-percentile of the message latency for bursty message arrivals. Overprovisioning of bandwidth and burst allowance ensures predictable latency.

5.6.2 Packet level simulations

We use ns2 to compare Silo against state-of-the-art solutions. Instead of using two specific tenants, we model two classes of tenants. Class A contains delay-sensitive tenants that run a small message application, and require bandwidth, delay and burst guarantees. Each class A tenant has an all-to-one communication pattern such that all VMs simultaneously send a message to the same receiver. This coarsely models the workload for OLDI applications [5]. Class B contains bandwidth-sensitive tenants that run a large message application and only require bandwidth guarantees. Such tenants have an all-to-all communication pattern, as is common for data parallel applications. The bandwidth and burst requirements of tenants in these classes are generated from an exponential distribution with the parameters in Table 5.3.

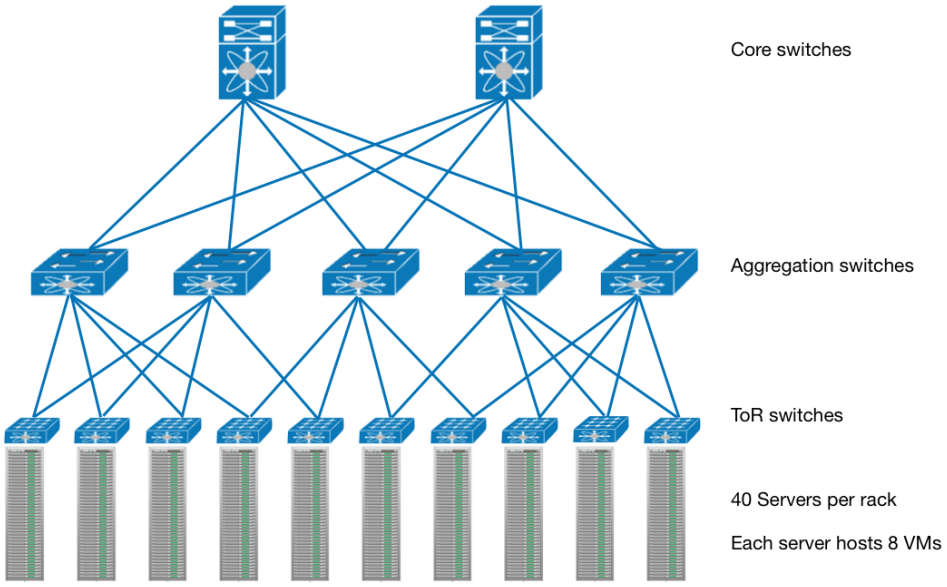

Figure 5.11: Topology used for ns2 simulations

Table 5.3: Tenant classes and their guarantees for the ns2 experiments

	Class A	Class B
Traffic Pattern	All-to-one	All-to-all
Bandwidth (B)	0.25Gbps	2Gbps
Burst length (S)	15kB	1.5kB
Delay guarantee (d)	1	N/A
Burst rate (B_{max})	1Gbps	N/A

5.6.2.1 Simulation setup

We use the ns2 simulator setup described in Section 2.6.2 to model 10 racks, each with 40 servers and 8 VMs per server, resulting in 3200 VMs. We use a multi-rooted tree topology for the cloud network as shown in Figure 5.11. The capacity of each network link is 10Gbps, the network has an oversubscription ratio of 1:5. We model commonly used shallow buffered switches with 312KB buffering per port (queue capacity is 250 µs). The number of tenants is such that 90% of VM slots are occupied. For Silo, VMs are placed using its placement algorithm. For Oktopus, VMs are placed using its bandwidth-aware algorithm [12]. For other solutions, we use a locality-aware algorithm that greedily places VMs close to each other.

5.6.2.2 Class A tenants

Figure 5.12 shows the latency for all small messages across 50 runs. Silo ensures low message latency even at the 99th percentile while all other approaches have high tail

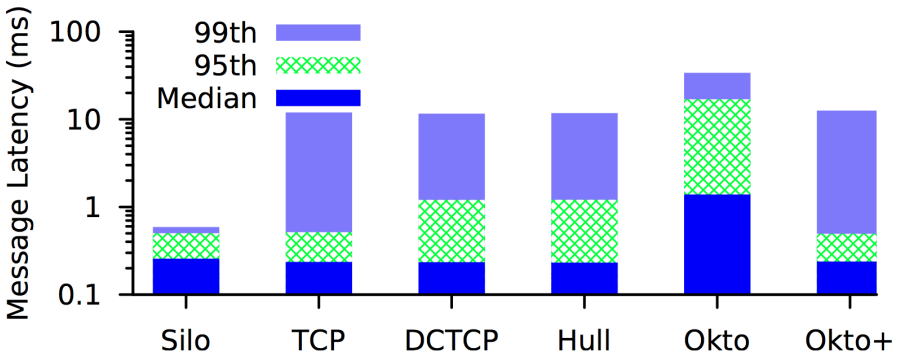

Figure 5.12: Message latency for class A tenants

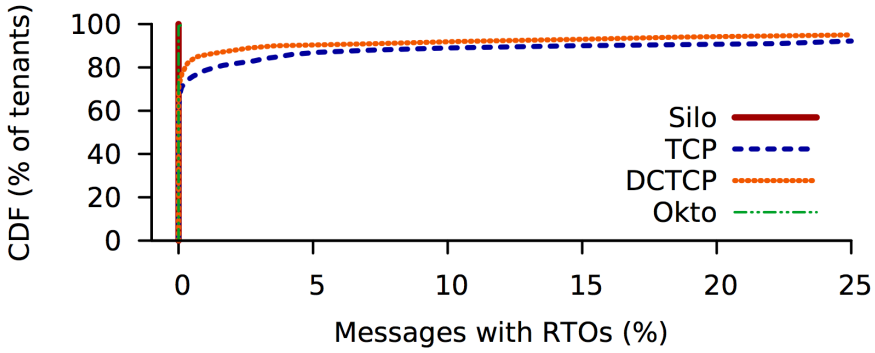

Figure 5.13: Class A tenants that suffer RTOs

latency. With Oktopus, VMs cannot burst, so the message latency is high, both at the average and at the tail. At 99th percentile, message latency is 60x higher with Oktopus compared to Silo. Okto+ is an Oktopus extension that couples bandwidth guarantees with burst allowance. It reduces the average latency but still suffers at the tail. This is because it does not account for VM bursts when placing VMs which, in turn, can lead to switch buffer overflows.

With DCTCP and HULL, message latency is higher by 22x at the 99th percentile (and 2.5x at the 95th). Two factors lead to poor tail latency for TCP, DCTCP and HULL. First, class A tenants have an all-to-one traffic pattern that leads to contention at the destination. Second, none of these approaches isolate performance across tenants by guaranteeing bandwidth, so class A small messages compete with large messages from class B tenants. This leads to high tail latency and losses for small messages. Figure 5.13 shows that when using TCP over 21% of class A tenants suffer more than 1% retransmission timeout events (RTOs). Such events are usually caused by TCP incast and are known to have a particularly bad impact on message completion time. With DCTCP and HULL, this happens for 14% of tenants. Thus, by itself, neither low queuing (ensured by DCTP and HULL) nor guaranteed bandwidth (ensured by Oktopus) is sufficient to ensure predictable message latency.

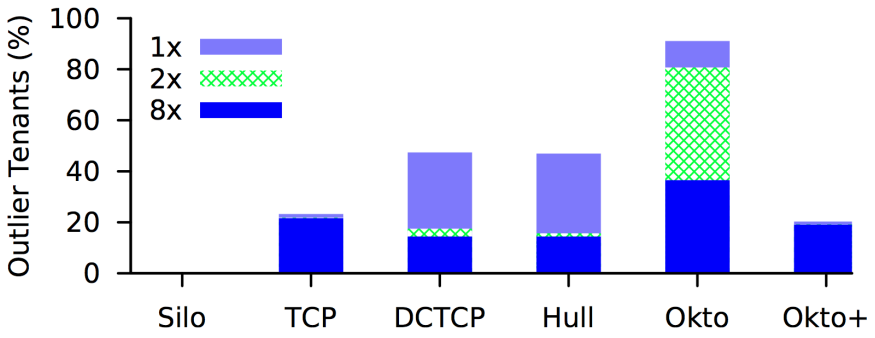

Figure 5.14: Class A tenants with outliers

We also look at outlier tenants, i.e. class A tenants whose 99th percentile message latency is more than the latency estimate. In Figure 5.14 we mark the fraction of outliers whose latency exceeds the estimate by 1x, 2x or 8x. Silo has no outliers while both DCTCP and HULL have 15% of tenants experiencing more than 8x the latency estimate.

5.6.2.3 Class B tenants

Silo does not let tenants exceed their bandwidth guarantee and thus it might impact the performance of class B tenants that have large messages where the completion time is dictated by the bandwidth they obtain. Figure 5.15 shows the average message latency for class B tenants, normalised to the message latency estimate. For clarity, we omit the results for HULL (similar to DCTCP) and Okto+. With both Silo and Oktopus, tenant bandwidth is guaranteed, so all large messages finish by the estimated time. With TCP and DCTCP, the message latency varies. 65% of tenants achieve higher bandwidth with DCTCP as compared to Silo but there is a long tail with many tenants getting very poor network bandwidth. Overall, this shows how Silo trades off best-case performance for predictability.

5.6.3 Large-scale flow-based simulations

To evaluate the performance of Silo's VM placement algorithm, we developed a flow-level simulator that models a public cloud data centre. The data centre has 32k servers with a three tier network topology. Tenant requests arrive according to a Poisson process. Poisson processes are often used to model job arrival times at queues and other service centres, and as they have been used to model tenant arrivals in other papers[12, 92, 150] choosing it helps with future comparisons. By varying the average arrival rate, we can control the average data centre occupancy. Each tenant runs a job with an all-to-all traffic between its VMs. Each job also has a minimum compute time. A job is said to finish when all its flows finish and the compute time has expired. We compare Silo's placement against two other approaches: Oktopus placement that guarantees VM bandwidth only

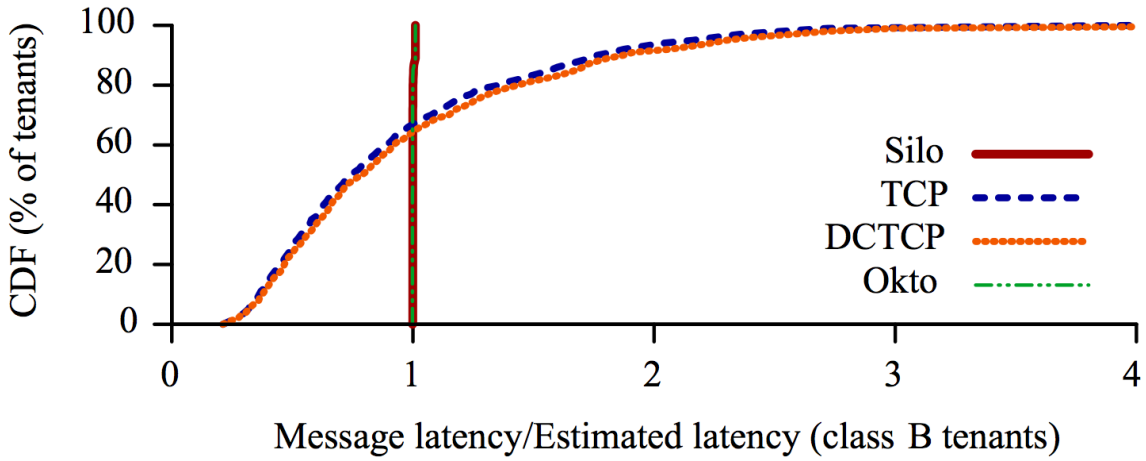

Figure 5.15: Message latency for class B tenants

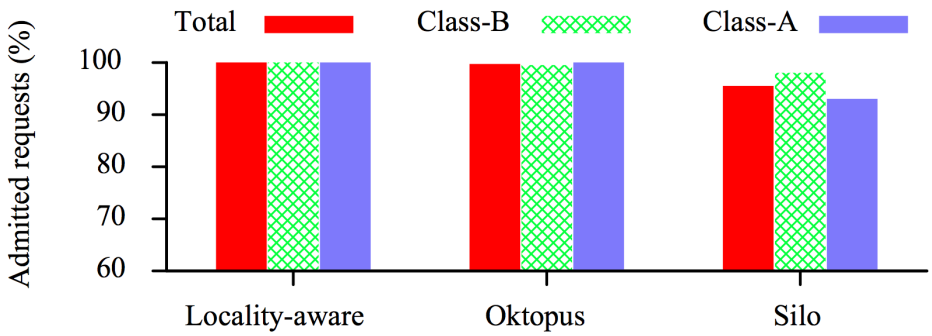

Figure 5.16: Number of requests admitted with 75% occupancy rate

and a locality-aware placement that greedily places VMs of a tenant close to each other. With the last approach, we emulate idealised TCP behaviour by sharing bandwidth fairly between flows. In this section we focus on the fraction of tenants that can be admitted by these approaches, and the impact on average network utilisation.

5.6.3.1 Admittance ratio

Figure 5.16 shows the fraction of tenants admitted with 75% data centre occupancy. Silo rejects 4.5% of tenants while the locality-aware placement accepts all of them and Oktopus rejects 0.3%. This is because Silo ensures that both the delay and bandwidth requirements of tenants are met, and may reject tenants even if there are empty VM slots. With Silo, the rejection ratio is higher for Class A tenants as their delay requirements are harder to meet.

However, as the data centre occupancy increases and tenants arrive faster, the admittance ratio of the locality-aware placement drops. Figure 5.17 shows that at 90% occupancy, it

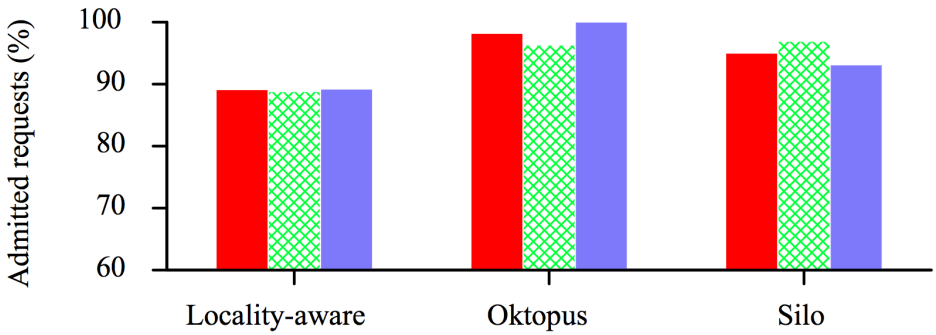

Figure 5.17: Number of requests admitted with 90% occupancy rate

rejects 11% of tenants as compared to 5.1% rejected by Silo. This result is counter-intuitive but can be explained as follows. Locality-aware placement will only reject requests if there are insufficient VM slots. By contrast, Silo can reject a request, even when there are empty VM slots, if the request's network guarantees cannot be met. The root cause is that locality-aware placement does not account for the bandwidth demands of tenants. So it can place VMs of tenants with high bandwidth requirements far apart. Such tenants get poor network performance and their jobs get delayed. These outlier tenants reduce the overall cloud throughput, delaying the time for the tenant to complete their jobs and causing subsequent requests to be rejected. With Silo, tenants get guaranteed bandwidth, so tenants do not suffer from poor network performance.

5.6.3.2 Network utilisation

Silo does not let tenants exceed their bandwidth guarantee, so it can result in network under-utilisation. However, the actual impact depends on tenants' traffic patterns. With the all-to-all communication modelled here, Silo's placement actually ends up improving utilisation as compared to the status quo, i.e. a TCP-like transport with locality-aware placement.

Figure 5.18 shows the average network utilisation in our experiments with varying data centre occupancy. As low occupancy, the data centre is lightly loaded, and there is not much difference between the approaches. At an occupancy of 75%, network utilisation with Silo is actually 6% higher than with locality-aware placement (which uses an idealised TCP for bandwidth sharing). As mentioned above, this is due to a small fraction of outlier tenants that get poor network performance and hence drag down average network utilisation. This is also the reason why network utilisation drops with increasing occupancy for the locality approach. Compared to Oktopus, Silo's network utilisation is lower by 10-13% at high occupancy. This is the price we pay for accommodating the strict delay requirements of class A tenants as it causes Silo to accept fewer requests than Oktopus and thus reduces network utilisation. Future work might look at whether we can

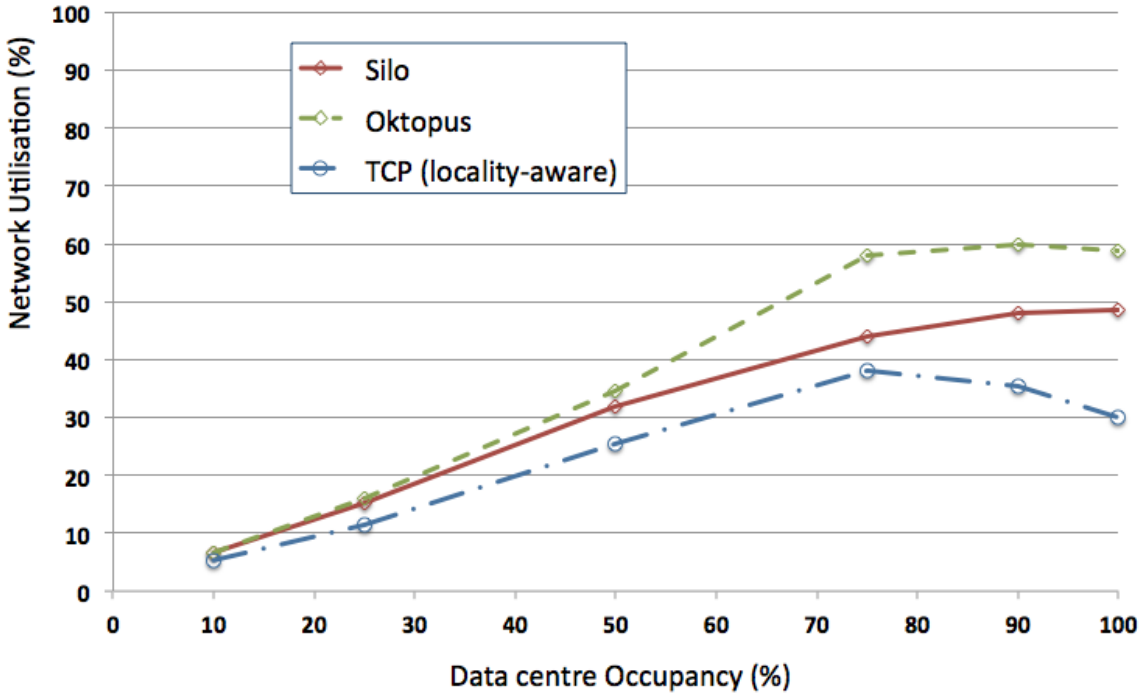

Figure 5.18: Average network utilisation for different data centre occupancy ratios

relax our bounds for queue occupancy without breaking the delay guarantees.

5.6.3.3 Placement scalability

We evaluate placement algorithm scalability by measuring the time to place tenants in a data centre with 100K hosts. Over 100K representative requests, the maximum time to place VMs is less than a second. Given the expected rate of churn in new tenants this suggests that our algorithm is scalable. We are sure that our algorithm is not the most efficient and future work might look at alternative algorithms that would scale even better.

5.6.3.4 Other simulation parameters

Figure 5.19 shows that the percentage of requests accepted by Silo reduces as we increase the average burst size requested by tenants. With an average burst size of 25KB, which is larger than messages for typical OLDI applications like web search [5, 167], Silo accepts 93.3% of requests. We also repeated the experiments while varying other simulation parameters and found the results to be qualitatively similar. For example, the admittance ratio results when data centre occupancy is lower than 75% are similar to the ones shown in Figure 5.16. As we increase the size of switch buffers or reduce network oversubscription, Silo's ability to accept tenants increases since the network is better provisioned.

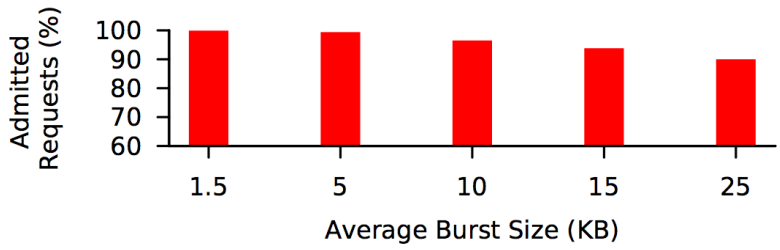

Figure 5.19: Comparison of requests admitted against average burst size

5.7 Summary and conclusions

This chapter described Silo, a tenant admission system designed to offer predictable message latency in multi-tenant data centres. We argue that to achieve such predictable latency, a general cloud application needs guarantees for its network bandwidth, packet delay and burstiness. We have shown how guaranteed network bandwidth makes it easier to guarantee packet delay.

Leveraging this idea, Silo enables these guarantees without any network or application changes, relying only on VM placement and end host packet pacing. We developed a placement algorithm that uses the ideas of network calculus to ensure that the requested guarantees can be met. Silo is able to be ported to any multi-tenant data centre that uses a placement algorithm to choose where to place new tenants and their virtual machines.

Silo was evaluated using a small-scale testbed and large-scale simulations. This evaluation shows that Silo can ensure predictable message completion time for both small and large messages in multi-tenant data centres. The testbed achieved fine grained packet pacing with low CPU overhead. The ns2 simulations show that Silo improves message latency compared to other state-of-the-art solutions like DCTCP [5], HULL [7], and Oktopus [12]. Finally the flow-level simulations were able to show that the new VM placement algorithm can actually improve both network and overall cloud utilisation.

Other systems such as openstack [115][8] also offer network isolation. However, this isolation is at a much coarser granularity than that offered by Silo. For instance, openstack network isolation relies on using VLANs to assign specific tenants to specific network interfaces or bonds. While this will isolate them from other tenants on the hypervisor, they can still be affected by congestion within the network. You are also limited by the number of physical network ports or bonds available. Even if you assign a fixed capacity to each VLAN you get face the same issues highlighted in Table 5.1. By contrast, Silo works at much finer granularity and is able to allow bursty traffic.

[8]Openstack has been gaining traction as an open-source architecture for creating data centre and cloud infrastuctures.

Chapter 6

Conclusions

This dissertation set out to prove my thesis that allowing data centre applications to choose a suitable transport protocol will give them improved performance compared to just using TCP.

I have shown how the use of TCP, a transport protocol originally designed for use in the Internet, has limited the performance of data centres. TCP has been optimised to maximise throughput, usually by filling up queues at the bottleneck. However, for most applications within a data centre network, latency is more critical than throughput. Consequently, the choice of transport protocol has become a bottleneck for performance. In itself this is not a new observation, but the usual approach to solving this problem has either been to use a TCP-like protocol such as DCTCP or to propose radical changes to the hardware or stack which moves too far from the core idea of building data centres from commodity off-the-shelf hardware.

I have explored alternatives that seek to minimise latency for applications that care about it, while still allowing throughput-intensive applications to receive a good level of service. Key contributions to this are Silo, a system designed to give tenants of a multi-tenant data centre guaranteed low latency network performance and Trevi, a novel transport system for storage traffic that utilises fountain coding to maximise throughput and minimise latency while being agnostic to drop, thus allowing storage traffic to be pushed out of the way when latency sensitive traffic is present in the network.

In Chapter 3 I demonstrated that within data centres, OLDI applications care more about overall flow completion times than packet latency. I explored the sources of latency within networks, and showed that for data centres, queues are the dominant source of latency. I then went on to explore how DCTCP performs when presented with traffic mixes that differ markedly from those it was designed for. This work suggests that no single transport protocol will always give optimum results. I also demonstrated that simply using DCTCP's modified form of the RED AQM mechanism gives you better performance at the tail than either DCTCP or TCP manage, as well as being almost as

good as DCTCP across all flows.

Chapter 4 described the Trevi storage protocol. Trevi is a new multicast storage architecture based on fountain coding. Trevi overcomes the limitations present in all storage systems that are based on TCP. The initial design for Trevi builds on the flat datacenter storage system [107]. It uses multicasting of reads and writes, along with a receiver-driven flow and congestion control mechanism that can better utilise storage and network resources in a data centre network. By using sparse erasure coding, storage traffic can be treated as a scavenger class, able to be dropped at any congested switch. Multicast then adds the ability to write to multiple replicas at once as well as read from multiple replicas in parallel, improving performance, reducing storage access times and reducing the need for complex tracking of metadata.

My simulations show that Trevi improves average flow completion times across a range of scenarios even without using multicast. Not only does it generally improve the performance of TCP, it also seems to help DCTCP perform better due to the strict priority imposed at all network queues. However, there were some unusual results that need further work to understand better. Chief among these was how Silo impacted the tail of the flow completion times for foreground TCP flows.

The limitations of the ns2 simulations do leave some questions unanswered that can only be properly explored in a large-scale real-life trial of a Trevi-like storage transport protocol. The key unanswered question is whether the claimed benefits of multicast would really be delivered as expected.

In Chapter 5 I discussed Silo, a tenant admission control system designed to provide tenants of multi-tenant data centres with guaranteed maximum latency and minimum bandwidth, while also allowing them to burst at a much higher rate if they need. The idea is to allow OLDI applications to run in public cloud-style data centres where individual tenants may not be trusted and must be isolated from each other.

Silo's use of network calculus allows it to place tenants such that even if they all send at their maximum rate they still achieve their guarantees, while its fine-grained packet pacing ensures tenants cannot exceed their guaranteed rates. Silo makes efficient use of network resources and outperforms approaches like Oktopus [12]. It also performs better than approaches like DCTCP [5] and HULL [7] that are designed to keep network queues short.

However, while it achieves the aim of allowing tenants to run OLDI applications, there is a definite price to pay in lost network utilisation. The main reason for this is the need to accomodate strict delay requirements imposed by OLDI applications. Future work would be needed to see whether the delay guarantees need to be so strictly applied. It might also be that a scavenger-style transport, such as Trevi, could make use of that unused capacity.

The above work supports my thesis. My analysis of DCTCP showed that, while a single protocol can be designed to give good performance for a given traffic matrix, it can't work as well when the traffic matrix changes. At one extreme, DCTCP will end up performing worse than TCP due to having insufficient flows to absorb congestion. At the other extreme, the Trevi results show that with low levels of storage traffic, even TCP performs well for the shortest flows. The Trevi results demonstrate how using a mix different protocols leads to an improved performance relative to either TCP or DCTCP alone. Finally, Silo gives tenants the freedom to use novel transport protocols, safe in the knowledge that they will always receive guaranteed performance from the network.

6.1 Next steps

The work presented in this dissertation has been largely driven by the belief that data centre transport protocols should concentrate on controlling latency.

In Appendix A, I introduce the idea of Transport Services. This is the concept that transport protocols should be seen as combinations of specific services rather than as a complete package. This would allow an application developer to exert fine-graided control over the performance they receive from the network. Currently the IETF is working on standardising this approach under the TAPS Working Group[71]. I was heavily involved in the process of chartering this working group and the work coming out of it promises to be extremely useful within the data centre context. Rather than just default to using TCP (or DCTCP), developers will be able to specify, for instance, that map-reduce traffic needs extremely tight latency bounds, but can afford to lose stragglers.

Storage traffic will always be a significant issue for data centres. As explained in Chapter 4, storage traffic fundamentally requires high throughput. This is at odds with the requirement for so much other traffic to receive guaranteed low latency. The obvious solution is to allow storage traffic to behave as a scavenger class, receiving the maximum available bandwidth but being preferentially dropped at any congested switch. Trevi proposes one possible solution that leverages fountain coding and multicast transmission to offer a storage system that is resilient to loss and offers the possibility of improved performance. The key next steps for Trevi are to better understand the impact of our data coding scheme (which uses XOR, an operation that is extremely efficient in hardware, but not in software) and to explore the impact of the various refinements discussed in section 4.4.5. George Parisis is pursuing this work at The University of Sussex, and is actively seeking research funding to take this forward.

Multi-tenant data centres are a particularly rich topic of research. Unlike single tenant data centres, in a multi-tenant DC you can't trust individual tenants to act for the common good so it is important to use systems that seek to isolate tenants from one another. Silo presents one approach for this, offering tenants guaranteed network performance at

the cost of admitting fewer tenants into the data centre. More work is needed to see whether the current placement algorithm can be improved in order to admit more tenants. The work done on Silo has also influenced more recent work from the same group within Microsoft Research. In particular their work on isolating tenants through the use of a virtualised data centre abstraction [10]. This work includes resources other than the network. However, it adopts a very simple approach to estimating the current demands on the network. Hitesh Ballani is already looking at whether the specific approach used in Silo can be used to improve this.

One key idea I didn't have time to explore is the use of explicit admission control systems within the data centre context. The work on Silo is close, with tenants being admitted based on whether they will be able to always receive their full network latency and throughput requirements. However, this approach is extremely conservative, and will often lead to low utilisation within the network (see Figure 5.18). I believe that an admission control system that actively monitors the state of the network offers the chance to dynamically adapt to conditions while ensuring no queues can ever build. HULL [7] goes some way towards this, combining "phantom queues" with the DCTCP [5] protocol. But I believe a better approach would be to use a system based on Pre-Congestion Notification [99]. This would use the current state of queues in the network to decide whether to allow an end-host to transmit. By combining this with a QoS system offering different traffic priorities it would be possible to ensure that all traffic always receives an appropriate latency and throughput, with latency sensitive traffic treated as if it were realtime traffic, storage traffic as best effort and background maintenance traffic receiving some less-than-best-effort service.

Bibliography

[1] H. Abu-Libdeh, P. Costa, A. Rowstron, G. O'Shea, and A. Donnelly. Symbiotic routing in future data centers. *SIGCOMM Comput. Commun. Rev.*, 41(4), Aug. 2010.

[2] M. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *Proc. of DSN 2005*, 2005.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of SIGCOMM*, 2008.

[4] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic flow scheduling for data center networks. USENIX Association, 2010.

[5] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *ACM SIGCOMM CCR*, volume 40. ACM, 2010.

[6] M. Alizadeh, A. Javanmard, and B. Prabhakar. Analysis of DCTCP: stability, convergence, and fairness. In *Proceedings of the ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 73–84. ACM, 2011.

[7] 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*, pages 19–19. USENIX Association, 2012.

[8] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. *ACM SIGCOMM Computer Communication Review*, 43(4):435–446, 2013.

[9] R. J. Anderson. The Eternity service. In *Pragocrypt*, 1996.

[10] S. Angel, H. Ballani, T. Karagiannis, G. OShea, and E. Thereska. End-to-end performance isolation through virtual datacenters. In *Proceedings of the 11th*

USENIX conference on Operating Systems Design and Implementation, pages 233–248. USENIX Association, 2014.

[11] M. Aron and P. Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems (TOCS)*, 18(3):197–228, 2000.

[12] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 242–253. ACM, 2011.

[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, 2003.

[14] L. Barroso and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis Lectures on Comp. Arch.*, 4(1), 2009.

[15] M. Belshe and R. Peon. SPDY Protocol (now part of HTTP/2. RFC7540). 2012. See IETF draft: draft-mbelshe-httpbis-spdy-00.

[16] S. Bensley, D. Thaler, L. Eggert, P. Balasubramanian, and G. Judd. Datacenter tcp (DCTCP): TCP congestion control for datacenters. *Final Draft, IETF, Internet-Draft draft-ietf-tcpm-dctcp*, 2017.

[17] T. Benson, A. Akella, and D. Maltz. Network traffic characteristics of data centers in the wild. In *ACM SIGCOMM IMC'10*, 2010.

[18] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. WREN '09. ACM, 2009.

[19] L. Brakmo and L. Peterson. TCP Vegas: End to end congestion avoidance on a global Internet. *Selected Areas in Communications, IEEE Journal on*, 13(8), 1995.

[20] P. Breuer, A. Lopez, and A. Ares. The Network Block Device. *Linux Journal*, March 2000.

[21] B. Briscoe. Flow rate fairness: Dismantling a religion. *ACM SIGCOMM CCR*, 37(2), 2007.

[22] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *Proc. of SIGCOMM*, 1998.

[23] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification (RFC1813). Technical Report 1813, IETF Secretariat, June 1995.

[24] G. Carlucci, L. De Cicco, and S. Mascolo. Http over udp: an experimental investigation of quic. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, pages 609–614. ACM, 2015.

[25] P. H. Carns, W. B. Ligon, III, R. B. Ross, and R. Thakur. PVFS: a parallel file system for Linux clusters. In *Proc. of USENIX ALS*, 2000.

[26] P. Cataldi, M. Shatarski, M. Grangetto, and E. Magli. Implementation and performance evaluation of LT and Raptor Codes for multimedia applications. In *Proc. of IIH-MSP*, 2006.

[27] Y. Chen, R. Griffith, J. Liu, R. Katz, and A. Joseph. Understanding TCP incast throughput collapse in datacenter networks. ACM, 2009.

[28] Y. Chen, R. Griffith, D. Zats, A. Joseph, and R. Katz. Understanding TCP incast and its implications for big data workloads. Technical Report UCB/EECS-2012-40, University of California, Berkeley, 2012.

[29] K. Christensen, P. Reviriego, B. Nordman, M. Bennett, M. Mostowfi, and J. Maestro. IEEE 802.3 az: the road to energy efficient ethernet. *Communications Magazine, IEEE*, 48(11), 2010.

[30] Cisco Systems Inc. *Data Center Infrastructure Design—IP Network Infrastructure*, 2011. `http://www.cisco.com/en/US/docs/solutions/Enterprise/Data_Center/DC_3_0/DC-3_0_IPInfra.html`.

[31] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, 1953.

[32] A. Cooper, R. Woundy, and B. Briscoe. Congestion exposure (ConEx) concepts and use cases (RFC6789). 2012.

[33] R. L. Cruz. A calculus for network delay. I. Network elements in isolation. *Information Theory, IEEE Transactions on*, 37(1):114–131, 1991.

[34] R. L. Cruz. A calculus for network delay. II. Network analysis. *Information Theory, IEEE Transactions on*, 37(1):132–141, 1991.

[35] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1), 2008.

[36] A. G. Dimakis, V. Prabhakaran, and K. Ramchandran. Decentralized erasure codes for distributed networked storage. *IEEE Transactions on Information Theory*, 52:2809–2816, 2006.

[37] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI*, volume 14, 2014.

[38] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive. A flexible model for resource management in virtual private networks. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 95–108. ACM, 1999.

[39] P. Eardley. Pre-congestion notification (PCN) architecture. *RFC 5559*, 2009.

[40] L. Ellenberg. DRBD 9 and device-mapper: Linux block level storage replication. In *Proc. of the Linux System Technology Conference*, 2009.

[41] G. Fairhurst, B. Trammell, and M. Kuehlewind. Services provided by IETF transport protocols and congestion control mechanisms (RFC8095). Technical report, 2017.

[42] A. Fikes. Storage architecture and challenges. Presentation to Google Faculty Summit 2010, 2010.

[43] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[44] S. Floyd. Highspeed TCP for large congestion windows RFC3649. 2003.

[45] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *Networking, IEEE/ACM Transactions on*, 1(4), 1993.

[46] S. Floyd, J. Padhye, and J. Widmer. TCP friendly rate control (TFRC): Protocol specification (RFC5348). 2008.

[47] A. Ford et al. Architectural guidelines for multipath TCP development (RFC6132). *RFC6132*, 2011.

[48] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure. TCP Extensions for Multipath Operation with Multiple Addresses (RFC6824). Technical Report 6824, IETF Secretariat, January 2013.

[49] L. Gan, A. Walid, and S. Low. Energy-efficient congestion control. ACM, 2012.

[50] GENI Project homepage. `http://www.geni.net/`.

[51] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of SOSP*, 2003.

[52] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: a scalable and flexible data center network. *ACM SIGCOMM CCR*, 39-4, 2009.

[53] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, Oakland, CA, May 2015. USENIX Association.

[54] C. Gunaratne, K. Christensen, B. Nordman, and S. Suen. Reducing the energy consumption of Ethernet with adaptive link rate (ALR). *Computers, IEEE Transactions on*, 57(4), 2008.

[55] 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. *SIGCOMM Comput. Commun. Rev.*, 39(4), Aug. 2009.

[56] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, page 15. ACM, 2010.

[57] S. Ha et al. CUBIC: a new TCP-friendly high-speed TCP variant. *SIGOPS Oper. Syst. Rev.*, 42(5), July 2008.

[58] The Apache Hadoop homepage. `http://hadoop.apache.org/`.

[59] D. Halperin, S. Kandula, J. Padhye, V. Bahl, and D. Wetherall. Augmenting data center networks with multi-gigabit wireless links. ACM, 2011.

[60] S. Hand and T. Roscoe. Mnemosyne: Peer-to-Peer steganographic storage. In *IPTPS*, 2002.

[61] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies (CoNEXT)*, pages 253–264. ACM, 2012.

[62] M. Handley. Why the Internet only just works. *BT Technology Journal*, 24(3), 2006.

[63] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of ACM SIGCOMM*, pages 29–42. ACM, 2017.

[64] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. ElasticTree: saving energy in data center networks. NSDI'10. USENIX Association, 2010.

[65] U. Hölzle. OpenFlow at Google. Presentation at Open Networking Summit, 2012.

[66] M. Honda, Y. Nishida, C. Raiciu, A. Greenhalgh, M. Handley, and H. Tokuda. Is it still possible to extend TCP? IMC 2011, 2011.

[67] C. Hopps. Analysis of an equal-cost multi-path algorithm. *RFC 2992*, 2000.

[68] S.-W. Huang, T.-C. Huang, S.-R. Lyu, C.-K. Shieh, and Y.-S. Chou. Improving speculative execution performance with coworker for cloud computing. pages 1004 –1009, dec. 2011.

[69] F. Huici, A. Greenhalgh, S. Bhatti, M. Handley, et al. HEN – Heterogeneous Experimental Network. Presentaiton to Multi-Service Networks Workshop, 2005.

[70] IEEE. IEEE 802.1 data center bridging task group. http://www.ieee802.org/1/pages/dcbridges.html.

[71] IETF. Transport services (taps) working group charter. https://datatracker.ietf.org/wg/taps/charter/.

[72] IRTF. Datacenter latency control DCLC research group. https://trac.tools.ietf.org/group/irtf/trac/wiki/dclc.

[73] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3), 2007.

[74] V. Jacobson. Congestion avoidance and control. ACM SIGCOMM '88, 1988.

[75] V. Jacobson. Congestion avoidance and control. In *ACM SIGCOMM Computer Communication Review*, volume 18, pages 314–329. ACM, 1988.

[76] V. Jacobson. Modified TCP congestion avoidance algorithm. *end2end-interest mailing list*, 1990.

[77] R. Jain. *The art of computer systems performance analysis - techniques for experimental design, measurement, simulation, and modeling.* Wiley professional computing. Wiley, 1991.

[78] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message completion time in the cloud. Technical report, Tech. Rep. MSR-TR-2013-95, 2013.

[79] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. Silo: Predictable message latency in the cloud. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 435–448. ACM, 2015.

[80] V. Jeyakumar, M. Alizadeh, D. Mazieres, B. Prabhakar, C. Kim, and A. Greenberg. EyeQ: practical network performance isolation at the edge. *REM*, 1005(A1):A2, 2013.

[81] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. IMC '09. ACM, 2009.

[82] M. Kodialam, T. Lakshman, and S. Sengupta. Efficient and robust routing of highly variable traffic. In *In Proceedings of ACM HotNets-III*. ACM, 2004.

[83] E. Kohler, M. Handley, and S. Floyd. Datagram Congestion Control Protocol (DCCP). (RFC4340). Technical Report 4340, IETF Secretariat, March 2006.

[84] J. Kurose. *On computing per-session performance bounds in high-speed multi-hop computer networks*, volume 20. ACM, 1992.

[85] The Linux KVM Hypervisor. `https://www.linux-kvm.org/page/Main_Page`.

[86] K. LaCurts, J. C. Mogul, H. Balakrishnan, and Y. Turner. Cicada: Introducing predictive guarantees for cloud networks. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2014.

[87] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[88] L.-A. Larzon, M. Degermark, S. Pink, L.-E. Jonsson, and G. Fairhurst. The Lightweight User Datagram Protocol (UDP-Lite). (RFC3828). Technical Report 3828, IETF Secretariat, July 2004.

[89] J.-Y. Le Boudec and P. Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*, volume 2050. Springer Science & Business Media, 2001.

[90] S. Lee, R. Panigrahy, V. Prabhakaran, V. Ramasubramanian, K. Talwar, L. Uyeda, and U. Wieder. Validating heuristics for virtual machines consolidation. *Microsoft Research, MSR-TR-2011-9*, 2011.

[91] B. Lin and P. A. Dinda. VSched: Mixing batch and interactive virtual machines using periodic real-time scheduling. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 8. IEEE Computer Society, 2005.

[92] Z. Liu, K. Chen, H. Wu, S. Hu, Y.-C. Hu, Y. Wang, and G. Zhang. Enabling work-conserving bandwidth guarantees for multi-tenant datacenters via dynamic tenant-eue binding. *arXiv preprint arXiv:1712.06766*, 2017.

[93] M. Luby. LT Codes. In *Proc. of FOCS*, 2002.

[94] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: library operating systems for the cloud. 2013.

[95] A. Madhavapeddy, R. Mortier, R. Sohan, T. Gazagnaire, S. Hand, T. Deegan, D. McAuley, and J. Crowcroft. Turning down the LAMP: software specialisation for the cloud. USENIX Association, 2010.

[96] T. Marill and L. G. Roberts. Toward a cooperative network of time-shared computers. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 425–431. ACM, 1966.

[97] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven layered multicast. In *SIGCOMM*, 1996.

[98] 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 CCR*, 38(2), 2008.

[99] M. Menth, F. Lehrieder, B. Briscoe, P. Eardley, T. Moncaster, J. Babiarz, A. Charny, X. Zhang, T. Taylor, K.-H. Chan, et al. A survey of PCN-based admission control and flow termination. *Communications Surveys & Tutorials, IEEE*, 12(3):357–375, 2010.

[100] J. Mickens, E. B. Nightingale, J. Elson, K. Nareddy, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, and O. Khan. Blizzard: fast, cloud-scale block storage for cloud-oblivious applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 257–273. USENIX Association, 2014.

[101] T. Moncaster, M. Menth, and B. Briscoe. Encoding three pre-congestion notification (PCN) states in the ip header using a single DiffServ codepoint (DSCP). (RFC6660). 2012.

[102] A. Munir, G. Baig, S. M. Irteza, I. A. Qazi, A. X. Liu, and F. R. Dogar. Friends, not foes: synthesizing existing transport strategies for data center networks. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 491–502. ACM, 2014.

[103] D. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. Ciel: a universal execution engine for distributed data-flow computing. 2011.

[104] Z. Nabi, T. Moncaster, A. Madhavapeddy, S. Hand, and J. Crowcroft. Evolving TCP.: how hard can it be? In *Proceedings of the 2012 ACM conference on CoNEXT student workshop*, pages 35–36. ACM, 2012.

[105] J. Naous, D. Erickson, G. Covington, G. Appenzeller, and N. McKeown. Implementing an openflow switch on the NetFPGA platform. In *Proceedings of ACM/IEEE ANCS'08*. ACM, 2008.

[106] K. Nichols and V. Jacobson. Controlling queue delay. *Communications of the ACM*, 55(7):42–50, 2012.

[107] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proc. of USENIX OSDI*, 2012.

[108] M. Nowlan et al. Fitting square pegs through round pipes. NSDI'12, 2012.

[109] The network simulator, ns-2. `http://www.isi.edu/nsnam/ns`.

[110] The ns-3 homepage. `http://www.nsnam.org`.

[111] The OMNet++ network simulation framework. `http://www.omnetpp.org`.

[112] OnApp. Use OnApp's cloud template library to save time and money!, 2018. `https://onapp.com/2018/02/05/use-onapp-cloud-template-library-save-time-money/`.

[113] OneLab homepage. `http://www.onelab.eu/`.

[114] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 29–41. ACM, 2011.

[115] openstack. `https://www.openstack.org/`.

[116] OPNET Modeler homepage. `http://www.opnet.com/solutions/network_rd/modeler.html`.

[117] Oracle. The Oracle Clustered File System. http://oss.oracle.com/projects/ocfs/.

[118] Z. Ou, H. Zhuang, J. K. Nurminen, A. Ylä-Jääski, and P. Hui. Exploiting hardware heterogeneity within the same instance type of amazon ec2. In *4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2012.

[119] R. Pan, P. Natarajan, C. Piglione, M. S. Prabhu, V. Subramanian, F. Baker, and B. VerSteeg. PIE: A lightweight control scheme to address the bufferbloat problem. In *High Performance Switching and Routing (HPSR), 2013 IEEE 14th International Conference on*, pages 148–155. IEEE, 2013.

[120] 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*, page 22. ACM, 2013.

[121] G. Parisis, G. Xylomenos, and T. Apostolopoulos. DHTbd: A reliable block-based storage system for high performance clusters. In *Proc. of CCGRID*, 2011.

[122] K. Pawlikowski, H. Jeong, and J. Lee. On credibility of simulation studies of telecommunication networks. *Communications Magazine, IEEE*, 40(1), 2002.

[123] B. Pawlowski, D. Noveck, D. Robinson, and R. Thurlow. The NFS version 4 protocol. In *Proc. of SANE 2000*, 2000.

[124] D. Pediaditakis, C. Rotsos, and A. W. Moore. Faithful reproduction of network experiments. 2014.

[125] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A centralized "zero-queue" datacenter network. 2014.

[126] A. Phanishayee, E. Krevat, V. Vasudevan, D. G. Andersen, G. R. Ganger, G. A. Gibson, and S. Seshan. Measurement and analysis of TCP throughput collapse in cluster-based storage systems. In *Proc. of USENIX FAST*, 2008.

[127] PlanetLab homepage. `http://planet-lab.org/`.

[128] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 187–198. ACM, 2012.

[129] J. Postel. User Datagram Protocol, (RFC768). Technical Report 768, IETF Secretariat, August 1980.

[130] J. Postel. Transmission Control Protocol (RFC793). Technical Report 793, IETF Secretariat, September 1981.

[131] P. Prakash, A. A. Dixit, Y. C. Hu, and R. R. Kompella. The TCP outcast problem: Exposing unfairness in data center networks. In *NSDI*, pages 413–426, 2012.

[132] S. Radhakrishnan, Y. Geng, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. SENIC: Scalable NIC for end-host rate limiting. In *Proc. USENIX NSDI*, 2014.

[133] S. Radhakrishnan, V. Jeyakumar, A. Kabbani, G. Porter, and A. Vahdat. NicPic: Scalable and accurate end-host rate limiting. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Cloud Computing*. USENIX, 2013.

[134] X. Ragiadakou, M. Alvanos, J. Chesterfield, J. Thomson, and M. Flouris. Microvisor: A scalable hypervisor architecture for microservers. 2016.

[135] C. Raiciu, S. Barre, C. Pluntke, A. Greenhalgh, D. Wischik, and M. Handley. Improving datacenter performance and robustness with multipath TCP. 2011.

[136] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley. How hard can it be? Designing and implementing a deployable multipath TCP. NSDI'12, 2012.

[137] K. Ramakrishnan and S. Floyd. The addition of explicit congestion notification (ECN) to IP (RFC3168). *3168*, 2001.

[138] J. Roskind. QUIC: Multiplexed stream transport over UDP. *Google working design document*, 2013.

[139] D. Rossi, C. Testa, S. Valenti, and L. Muscariello. LEDBAT: the new BitTorrent congestion control protocol. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–6. IEEE, 2010.

[140] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. Moore. OFLOPS: An open framework for OpenFlow switch evaluation. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012.

[141] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 123–137. ACM, 2015.

[142] Y. Saito, S. Frolund, A. C. Veitch, A. Merchant, and S. Spence. FAB: building distributed enterprise disk arrays from commodity components. In *Proc. of ASPLOS*, 2004.

[143] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proc. of USENIX FAST*, 2002.

[144] H. Schulzrinne. RTP: A transport protocol for real-time applications RFC1889. 1996.

[145] P. Schwan. Lustre: Building a file system for 1,000-node clusters. In *Proc. of the Linux Symposium*, 2003.

[146] M. Scott, A. Moore, and J. Crowcroft. Addressing the scalability of ethernet with MOOSE. In *Proc. DC CAVES Workshop*, 2009.

[147] Stanford experimental data center laboratory. `http://simula.stanford.edu/sedcl`.

[148] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol (RFC3530). Technical Report 3530, IETF Secretariat, April 2003.

[149] S. Shepler, M. Eisler, and D. Noveck. Network File System (NFS) Version 4 Minor Version 1 Protocol (RFC5661). Technical Report 5661, IETF Secretariat, January 2010.

[150] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *NSDI*, volume 11, pages 23–23, 2011.

[151] A. Shokrollahi. Raptor codes. *IEEE Transactions on Information Theory*, 52(6):2551–2567, 2006.

[152] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[153] S. Smith, A. Madhavapeddy, C. Smowton, M. Schwarzkopf, R. Mortier, R. Watson, and S. Hand. The case for reconfigurable I/O channels. RESoLVE workshop at ASPLOS'12, 2012.

[154] S. Smith, A. Madhavapeddy, C. Smowton, M. Schwarzkopf, R. Mortier, R. M. Watson, and S. Hand. The case for reconfigurable i/o channels. In *RESoLVE workshop at ASPLOS*, volume 12, 2012.

[155] K. T. J. Song, Q. Zhang, and M. Sridharan. Compound TCP: A scalable and TCP-friendly congestion control for high-speed networks. *Proceedings of PFLDnet 2006*, 2006.

[156] W. Stevens. TCP slow start, congestion avoidance, fast retransmit, and fast recovery algorithms RFC2001. 1997.

[157] R. Stewart. Stream Control Transmission Protocol (RFC4960). Technical Report 4960, IETF Secretariat, September 2007.

[158] L. Strigeus, G. Hazel, S. Shalunov, A. Norberg, and B. Cohen. uTorrent Transport Protocol, Jun 2009. `http://www.bittorrent.org/beps/bep_0029.html`.

[159] V. Vasudevan et al. Safe and effective fine-grained TCP retransmissions for datacenter communication. In *ACM SIGCOMM CCR*, volume 39-4. ACM, 2009.

[160] 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 SIGCOMM*, 2009.

[161] VMware homepage. `http://www.vmware.com/`.

[162] G. Wang, D. Andersen, M. Kaminsky, K. Papagiannaki, T. Ng, M. Kozuch, and M. Ryan. c-Through: Part-time optics in data centers. volume 40. ACM, 2010.

[163] D. X. Wei, C. Jin, S. H. Low, and S. Hegde. Fast TCP: motivation, architecture, algorithms, performance. *IEEE/ACM Transactions on Networking (ToN)*, 14(6):1246–1259, 2006.

[164] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proc. of USENIX SOSP*, 2006.

[165] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the Panasas parallel file system. In *Proc. of USENIX FAST*, 2008.

[166] M. Welzl, M. Tüxen, and N. Khademi. On the usage of transport service features provided by IETF transport protocols. *Internet Draft draft-ietf-taps-transports-usage, Approved as RFC*, 2017.

[167] C. Wilson et al. Better never than late: meeting deadlines in datacenter networks. SIGCOMM '11, 2011.

[168] D. Wing and A. Yourtchenko. Happy eyeballs: Success with dual-stack hosts (RFC6555). (6555), 2012.

[169] H. Wu, Z. Feng, C. Guo, and Y. Zhang. ICTCP: Incast congestion control for TCP in data center networks. In *Proceedings of CoNEXT*, 2010.

[170] Xen hypervisor homepage. `http://xen.org/`.

[171] C. Xu, S. Gamage, P. N. Rao, A. Kangarlou, R. R. Kompella, and D. Xu. vSlicer: latency-aware virtual machine scheduling via differentiated-frequency CPU slicing. In *Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing*, pages 3–14. ACM, 2012.

[172] L. Xu, K. Harfoush, and I. Rhee. Binary increase congestion control (BIC) for fast long-distance networks. In *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, volume 4, pages 2514–2524. IEEE, 2004.

[173] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. EuroSys '10. ACM, 2010.

[174] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. DeTail: reducing the flow completion time tail in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 42(4):139–150, 2012.

[175] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

[176] Y. Zhang and N. Ansari. On mitigating TCP incast in data center networks. In *Proc. of IEEE INFOCOM*, 2011.

Appendix A

The role of sender transport selection

As I explained in Section 1.1, TCP has in effect become the new narrow waist for the data centre network. Despite the large number of alternative transports that are available in modern operating systems, developers continue to use TCP (or transports that look like TCP on the wire) because these are then guaranteed to pass all middleboxes within the network. However as my results in Section 4.7 show, the use of TCP leads to huge variability in flow completion time for both foreground and background flows. This runs counter to the idea that latency control is one of the key requirements for data centre applications (3).

There are three broad approaches to improving latency control within a data centre:

1. **End-to-end approaches.** In the 4-layer TCP/IP model, the transport protocol is responsible for congestion control, flow control and session control. Consequently, it is logical for latency control to also be handled at this layer. A good example of this sort of approach is Datacenter TCP (DCTCP) [5], which helps short, latency sensitive flows by ensuring that long bulk transfer flows do not congest network switches.

2. **Fabric-based approaches.** Software Defined Networking (SDN) is a fairly new paradigm that allows much finer-grained control over network flows. It provides an abstraction that allows richer interaction with the data control plane. Google have reported that they make extensive use of SDN within their WAN that connects their data centres together [65]. Inside a data centre, SDN-enabled fabrics offer new ways to perform traffic engineering that focus on latency control. Other proposals to improve data centre fabrics include novel queue management algorithms like DeTail [174], layer-2 improvements like data center bridging (DCB) [70], centralised arbitration as used by Fastpass [125], and others.

3. **Hybrid approaches.** Some new approaches to data centre latency control are hybrid, i.e., they combine improved end-system algorithms with modifications to the network fabric. Such hybrid approaches are not new in networking, e.g., the original quality-of-service approaches for the Internet – DiffServ and IntServ – can be thought of as combining end system traffic marking and signalling with in-network reservations and policing. However, the design space for such approaches in data centres is much broader, and the deployment possibilities are much greater in environments that are under the control of a single entity. Examples include zero-queue traffic shaping used by HULL [7], deadline-aware explicit rate control [167] and composite mechanisms [102, 63].

This dissertation is primarily interested in end-host approaches that can be readily deployed in commodity data centres. The rest of this appendix explores the limitations on deploying new transport protocols, explains the concept of Transport Services and shows how this approach can lead to improved transport behaviours, particularly in virtualised data centres.

A.1 The role of transport protocols

Transport protocols are responsible for the end-to-end aspects of network communications. Over the past two decades a large range of transport protocols have been designed. Many of these have gone on to be standardised by the IETF including UDP [129], TCP [130], SCTP [157], UDP-Lite [88], DCCP [83] and MPTCP [48]. In most cases new protocols have been defined because the IETF has established that there is a need for a set of behaviours than cannot be offered by any existing transport protocol. However, for an application programmer, using protocols other than TCP or UDP can be hard: not all protocols are available everywhere, hence a fall-back solution to TCP or UDP must be implemented. This can hold true even in the relatively controlled environment of a data centre.

The main transport functions are:

- **Reliability and error control**—Ensuring the connection delivers the correct data reliably. This means any data sent by the application will get delivered as long as the connection itself survives.
- **Repairing packet loss**—Identifying and repairing any packet loss in the network, usually by spotting holes in the received sequence space and by retransmitting missing data.
- **Ordering**—Delivering data in the correct order to the application.

- **Timeliness**—Delivering the data within an appropriate time frame.
- **Congestion control**—Responding to congestion to ensure the transmission rate doesn't trigger congestion collapse in the network.
- **Flow control**—Ensuring the receiving system can cope with the data rate being sent.
- **Session control**—Maintaining and controlling the two way conversation between sending and receiving application. Sessions often outlive the actual underlying connection.
- **Security**—Ensuring data is delivered to the application without interference or interception.

The IETF has also added a de facto requirement for "TCP fairness", meaning that protocols should avoid being "unfair" to competing TCP streams. It is important to note that not all transport protocols need implement all these functions.

Within the data centre environment *Timeliness* is especially important. As discussed in Chapter 3, most data centre applications require the data to be delivered within a predictable time frame. Delays can lead to problems with stragglers and for OLDI applications there may even be a strict deadline to deliver the results (for instance a web search may have a deadline of 100ms to return search results to the customer). Equally, for intra data centre traffic within a single tenant data centre, *security* can be ignored as all end-hosts are trusted.

Different transport protocols may provide some or all of these services and may do so in different fashions. Layering decisions must be made e.g. should a protocol be used natively or over UDP [138]. Because of these complications, programmers often resort to either using TCP (even if there is a mismatch between the services provided by TCP and the services needed by the application) or implementing their own customised solution over UDP, thus losing the opportunity of benefiting from other transport protocols. Since all these protocols were developed to provide services that solve particular problems, the inability of applications to make use of them is in itself a problem. Implementing a new solution also means re-inventing the wheel (or, rather, re-implementing the code) for a number of general network functions such as methods to pass through NATs and path maximum transport unit discovery (PMTUD).

In 2013 I, along with several other members of the Internet Standards community, started a push to define a new Working Group at the Internet Engineering Task Force (IETF). This has now been approved and goes under the name of TAPS or Transport Services [71]. This Working Group is chartered to define a minimal set of Transport Services that an application should be able to choose from. It defines a Transport Service as an end-to-end

facility provided by the transport layer that can only be correctly provided by using information from the application. The idea is to split the required transport behaviour from the underlying protocol that provides that behaviour. As an example, TCP provides a number of transport services including in-order delivery, reliability and application multiplexing. However, applications that rely on live streaming would be happy to sacrifice the reliability and ordering in favour of maintaining consistent throughput.

A.2 Transport Services

The transport layer provides many services both to the end application (e.g. multiplexing, flow control, ordering, reliability) and to the network (e.g. congestion control). In the TAPS Working Group Charter the IETF defines Transport Services as follows: *A Transport Service is any end-to-end service provided by the transport layer that can only be correctly implemented with information from the application.*

This is quite a narrow definition and needs careful explanation. The key word here is "information"—many existing transport protocols function perfectly adequately because the choice of protocol implicitly includes information about the desired transport capabilities. For instance the choice of TCP implies a desire for reliable, in-order data delivery. "Correctly implemented" means implemented in exactly the way the application desires. Implicit information such as is used currently is not always sufficient. For instance TCP is often used as the "lowest common denominator" transport that is understood by all nodes in the network and passes the majority of middleboxes. However this imposes a set of decisions about which Transport Services the traffic will receive.

Transport Services are not the same as the list of transport functions given above. However there is clearly a link. A Transport Service is a specific choice made about how to achieve a given transport function. As a simple example consider reliability. Reliability can be thought of as a spectrum ranging from completely reliable transport protocols such as TCP through to unreliable protocols such as UDP. The function in all cases is that of reliability but the service is the desired degree of reliability. For instance a real-time video application may choose to use TCP because it is more likely to work across a range of middleboxes, but almost certainly it doesn't want such complete reliability since retransmitting missing frames makes no sense in a real-time application.

The rest of this section explains how to identify Transport Services and how those services might then be exposed to the application.

A.2.1 Identifying Transport Services

One of the key aspects of the IETF work is how to identify which Transport Services should actually be supported. They adopted a two stage approach. Initially they surveyed all

existing IETF transport protocols in order to identify the underlying Transport Services these provide [41]. Now they are in the process of identifying the specific transport primitives provided by each protocol and using these to construct a list of the overarching transport features that can be combined into transport services[166]. Subsequently they will explore API mechanisms to allow applications to request particular transport services and provide guidance on how a TAPS-capable transport layer might choose between available mechanisms. It is hoped that this approach to identifying the set of service primitives will allow them to be combined to offer a rich set of services to the application.

A.2.2 Transport Primitives

In [166] the TAPS WG has identified a set of transport features provided by current IETF transports. They have divided these into features relating to the end-to-end connection and features relating to the actual data transfer.

Connection features

- **Establishment**. These features relate to the creation of the connection, the negotiation of options, the authentication of the connection, setting up sockets to listen for incoming replies and handing over any data to be sent during establishment.
- **Maintenance**. These features relate to maintaining a stable end-to-end connection. This includes heartbeat messages, renegotiation of options, path maintenance for multipath transports (adding, removing, switching and recategorising), authentication and numerous control features relating to path MTU, TTL, checksums and underlying IP options.
- **Termination**. These features relate to tearing down the end-to-end connection. This can be done cleanly (with all data transmitted), by aborting the connection (with or without informing the other side) and through timeouts.

Data transfer features

- **Sending**. These features relate to how the data is sent, whether it is reliable and/or ordered and via what path it is sent (for multipath transports).
- **Receiving**. These features relate to how a receiver handles data it has received. The data may be delineated (SCTP, UDP, etc.) or it may be a stream (TCP, MPTCP, etc.).
- **Error control**. These features are mainly specific to SCTP and relate to explicit error messages that can be sent to the receiver.

A.2.3 Exposing Transport Services

Transport Services should be exposed to the application via an API. The definition of such an API and the functionality underneath the API are beyond the scope of this dissertation. However I describe two simple approaches below. The first is based on moving transport functionality up the stack into the operating system. The second is called PVTCP (polyversal TCP) [104] an approach I developed in my first year working in parallel with an MPhil student in the department. This is explored in section A.3 below.

A.2.4 Operating system transports

One approach could be to develop a transport system that fully operates inside the Operating System. This transport system would provide all the defined services for which it can use TCP as a fall-back at the expense of efficiency (e.g., TCP's reliable in-order delivery is a special case of reliable unordered delivery, but it may be less efficient). To test whether a particular transport is available it could take the Happy Eyeballs [168] approach proposed for SCTP—if the SCTP response arrives too late then the connection just uses TCP and the SCTP association information can be cached so that a future connection request to the same destination IP address can automatically use it.

A.3 Polyversal TCP

As noted above, application developers really only have two choices for sending data over the network—TCP, with its reliable, ordered, congestion-controlled byte stream model or UDP with its unordered, unreliable datagram model. Middleboxes such as firewalls and intrusion detection systems have effectively hard-wired TCP into the Internet and have made it increasingly hard for novel transport protocols to be deployed [66].

TCP and UDP support a remarkable variety of applications over a huge range of connection speeds and latencies, but are struggling to meet the demands of today's high-bandwidth, low latency applications. This has led developers to use the underlying transport as a substrate over which to run application layer transports. Examples of this are Minion [108], which uses TCP as its substrate but allows the application to trade reliability in favour of reduced latency, and μTP [158], which provides TCP-like reliability on top of UDP for BitTorrent and uses a custom approach to congestion control.

Multipath TCP (MPTCP) takes a different approach [136]. The transport is designed to work alongside TCP, and clever design choices result in a new protocol that looks like TCP on the wire, but which is able to make far better use of the available bandwidth resource pool across multiple interfaces.

MPTCP points to a new approach for evolving transport protocols. Rather than expecting a new protocol to survive in an Internet dominated by middleboxes, we suggest that it should adopt a form of camouflage. Raiciu *et al.* [136] identify three design goals that are applicable to any new transport: to be able to work with unmodified receivers and APIs, to work in all cases where TCP currently works and to offer performance at least as good as TCP in any circumstances.

Early in my PhD, I helped come up with Polyversal TCP[1] (PVTCP)[104]. This section describes the original design of PVTCP. PVTCP has since evolved into a complete new approach being worked on by Vsevolod Stakhov and other people here within the Computer Laboratory.

A.3.1 PVTCP design guidelines

The middleboxes deployed by most operators have effectively limited the choice of transport protocols to UDP or TCP. This is symptomatic of the ossification that has been evident for some years [62]. Even within a data centre there are real issues caused by middleboxes. In single tenant data centres middleboxes may be used for load balancing, monitoring and traffic conditioning. Within multi-tenant data centres they are also used for security (both encryption of user data and firewalling to isolate tenants from each other) and triple-A (authentication, authorisation and accounting).

However there is a simple solution—if a new transport looks like TCP on the wire, then it survives the first hurdle to adoption in the wider Internet. MPTCP is proof that this approach works. But in and of itself this isn't enough, there are additional guidelines that should be followed if it is to offer more functionality than simple TCP:

1. The new transport should offer real deployment benefits—the history of the IETF is littered with new transports that have never got traction because there was no realistic deployment model.

2. The new protocol should exhibit stability and resilience in the face of adverse network conditions. In particular the protocol must be aware of the risk of fighting with itself in cases where it causes self-congestion.

3. The protocol should fail gracefully in the presence of aggressive middleboxes, coping with transparent erasure of TCP options and falling back to vanilla TCP.

There is also an important non-goal that has hampered the adoption of many new proposals: TCP or flow-rate fairness. This is the flawed notion that at any bottleneck every

[1]Myself and Zubair Nabi, an MPhil student, came up with closely related concepts independently. PVTCP was born out of the combination of these. Credit has to also go to our co-authors, Anil Madhavapeddy, Steve Hand and Jon Crowcroft.

flow should receive an equal share of the resources. There are many objections to this idea [21], but among the most critical is that it fails to take account of applications that simply open multiple flows in order to get a greater share of the available bandwidth [15].

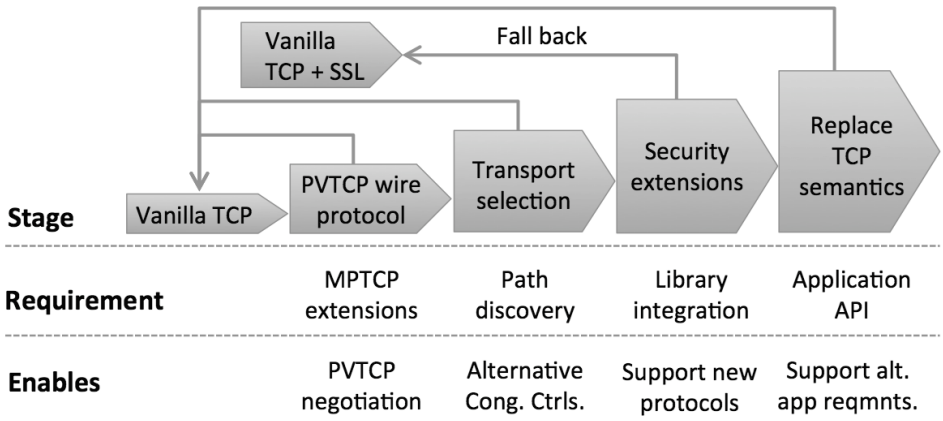

Figure A.1: The evolution of PVTCP. At every stage we offer enhanced performance over TCP and provide sensible fall-back strategies.

A.3.2 From universal to polyversal

The universality of TCP means it is the jack of all trades and master of none. By applying the guidelines above we can change that, creating a transport protocol that can offer any feature the application designer wants while still retaining the ability to fall back to vanilla TCP.

Polyversal TCP builds upon the MPTCP sub-flow mechanism by allowing the application to customise each sub-flow independently. These sub-flows can exhibit different *characteristics* (congestion control, reliability, ordering, security, etc.) depending on application requirements and the underlying network (including middleboxes). During setup, PVTCP performs path characterisation using mechanisms similar to path-MTU discovery. Using this information, PVTCP can either transparently choose the transport semantics for a particular sub-flow or if the application wants fine-grained control, then it can use `setsockopt()` to explicitly customize each sub-flow via a per-sub-flow socket. PVTCP maintains backwards compatibility with the traditional socket API by keeping the `socket()`, `bind()`, and `listen()` socket calls intact. If problems are found at any point during the lifetime of a connection, it can simply fall back to standard TCP for that connection or use alternatives such as MPTCP or SSL over MPTCP as shown in Figure A.1. As such PVTCP is an embodiment of the TAPS approach to transport design.

A.3.3 PVTCP in the data centre

TCP is designed to communicate between remote processes residing on different physical machines. Where the processes reside on the same machine, mechanisms such as direct memory access (DMA) serve the same purpose. However in a data centre it is hard to know where processes actually reside, and so TCP has become the default transport for all inter-process communications (IPC). TCP has a number of issues that make it unsuitable for this role including its lack of stability in the data centre, its requirement to push the network to congestion and issues with TCP incast[27] and outcast[131].

Data centre networks are an extreme case in which virtual hosts maintain multiple communication channels within and across physical machines. The underlying subnetworks of these channels can vary from on-chip multicore interconnects to inter-host Ethernet, optical or Infiniband links. They exhibit an order of magnitude difference in performance depending on the transport regime and the layout of the underlying network [153]. In such situations, applications can customize each sub-flow directly through the PVTCP socket API or allow PVTCP to do so on its behalf. For instance, PVTCP can choose the transport based on the size of the transfer and the location of the destination. In addition, for virtualized hosts, PVTCP can ensure robust live migration by temporarily switching to standard TCP to allow shared memory channels to be replaced.

A.4 Conclusions

This appendix introduced the idea of treating transport protocols as a set of specific services rather than a single combined protocol. It described the eight main roles of transport protocols and formally introduced the concept of Transport Services. I went on to describe the aims of the new IETF TAPS working group which I helped set up in 2013. Finally I introduced Polyversal TCP, an approach that is designed to embody the idea of Transport Services, allowing applications to make use of the most appropriate underlying transport protocol that is available.

Appendix B

Silo's Placement Algorithm

As explained in Section 5.4.2.3 of the main text two constraints are sufficient to describe a valid VM placement. These are that that any network links carrying a VM's traffic have sufficient capacity and that the sum of queue bounds across the path between pairs of VMs should be less than the delay guarantee.

Given the design of most data centres, a given network request can have many valid placements that meet these constraints. The algorithm below tries to find the placement that minimises the "level" of network links that carry a new tenant's traffic, thus preserving network capacity for future tenants.

Servers represent the lowest level of network hierarchy, followed by racks and pods. Our algorithm places a tenant's VMs while greedily optimizing this goal by progressively seeking to place all requested VMs on a single server, within the same rack, within a single pod or if all else fails, across pods. At each stage we use the queuing constraints on the uplink switch port to determine the number of VMs that can be placed at a given server.

Ensure: Placement for requests with N VMs with guarantees *B,S,d*

Require: Topology tree T consisting of pods, racks and hosts. Pre-calculated state includes *delayQuota[d,l]* (max link delay for a request with delay guarantee d allocated at level l) and *xxxUpdelay[d,l]* & *xxxDownDelay[d,l]* for xxx = server, rack or pod.

```
1: if N < VMSlotsPerServer then
2:     return AllocOnServer(request)
3: end if
4: for each l ∈ [0, T.height − 1] do
5:     for each p ∈ T.pods do
6:         vmsPerPod = 0
7:         for each r ∈ p.racks do
8:             vmsPerRack = 0
9:             for each s ∈ r.servers do
10:                v = CalcValidAlloc(s.emptySlots,N,requests.upLink,serverUpDelay[d,l],
           serverDownDelay[d,l], delayQuota[d,l])
11:                vmsPerRack += v
12:                if vmsPerRack >= N and l == 0 then
           return AllocOnRack(r,request)
13:                end if
14:            end for
15:            if l > 0 then
16:                v = CalcValidAlloc(vmsPerRack,N,request,r.UpLink,rackUpDelay[d,l],
           rackDownDelay[d,l], delayQuota[d,l])
17:                vmsPerPod += v
18:                if v >= N and l == 1 then
19:                    AllocOnPod(p,request)
20:                end if
21:            end if
22:        end for
23:        if l > 1 then
24:            v = CalcValidAlloc(vmsPerPod,N,request,p.UpLink,podUpDelay[d,l],
           podDownDelay[d,l],delayQuota[d,l])
25:            if v >= N and l == 2 then
26:                AllocOnCluster(request)
27:            end if
28:        end if
29:    end for
30: end for
31: function CalcValidAlloc(k,N,request,uplink,updelay,downdelay,delay)
32:    for each m ∈ [k, 1] do
33:        if (uplink.GetMaxDelay(request,(N-m),N,updelay) < delay
           and uplink.reverse.GetMaxDelay(request,m,N,downdelay) < delay) then
           return m
34:        end if
35:    end for
36: end function
```