Technical Report

UCAM-CL-TR-937
ISSN 1476-2986

Number 937

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Latency-driven performance in data centres

Diana Andreea Popescu

June 2019

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted December 2018 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Churchill College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Latency-driven performance in data centres

Diana Andreea Popescu

Summary

Data centre based cloud computing has revolutionised the way businesses use computing infrastructure. Instead of building their own data centres, companies rent computing resources and deploy their applications on cloud hardware. Providing customers with well-defined application performance guarantees is of paramount importance to ensure transparency and to build a lasting collaboration between users and cloud operators. A user's application performance is subject to the constraints of the resources it has been allocated and to the impact of the network conditions in the data centre.

In this dissertation, I argue that application performance in data centres can be improved through cluster scheduling of applications informed by predictions of application performance for given network latency, and measurements of current network latency in data centres between hosts.

Firstly, I show how to use the Precision Time Protocol (PTP), through an open-source software implementation PTPd, to measure network latency and packet loss in data centres. I propose PTPmesh, which uses PTPd, as a cloud network monitoring tool for tenants. Furthermore, I conduct a measurement study using PTPmesh in different cloud providers, finding that network latency variability in data centres is still common. Normal latency values in data centres are in the order of tens or hundreds of microseconds, while unexpected events, such as network congestion or packet loss, can lead to latency spikes in the order of milliseconds.

Secondly, I show that network latency matters for certain distributed applications even in small amounts of tens or hundreds of microseconds, significantly reducing their performance. I propose a methodology to determine the impact of network latency on distributed applications performance by injecting artificial delay into the network of an experimental setup. Based on the experimental results, I build functions that predict the performance of an application for a given network latency.

Given the network latency variability observed in data centres, applications' performance is determined by their placement within the data centre. Thirdly, I propose latency-driven, application performance-aware, cluster scheduling as a way to provide performance guarantees to applications. I introduce NoMora, a cluster scheduling architecture that leverages the predictions of application performance dependent upon network latency combined with dynamic network latency measurements taken between pairs of hosts in data centres to place applications. Moreover, I show that NoMora improves application performance by choosing better placements than other scheduling policies.

Acknowledgements

"Nu şovăi, nu te-ndoi, nu te-ntrista.
Purcede drept şi biruie-n furtună."

–Tudor Arghezi, *Inscripţie pe o uşe*

First, and foremost, I would like to thank my supervisor, Andrew W. Moore, for his help and support throughout the years. His guidance through ideas of experiments and discussions has been crucial. Andrew has taught me to look at the wider implications of my research, and has always had a positive outlook on what I have achieved. He has always encouraged me to attend conferences, and supported me whenever I wanted to travel home to recharge. Furthermore, Andrew's feedback on this dissertation has been invaluable.

I would like to thank the members of the Networking and Operating Systems Group, past and present: Jon Crowcroft, Richard Mortier, Malcolm Scott, Gianni Antichi, Salvator Galea, Neels Manihatty-Bojan, Marcin Wojcik, Nik Sultana, Murali Ramanujan, Jan Kucera, Matt Grosvenor, Eva Kalyvianaki, Noa Zilberman, for fruitful interactions and discussions; Ionel Gog and Malte Schwarzkopf for answering questions about Firmament. I would like to thank George Neville-Neil for answering questions and providing insights about PTPd. I would like to thank Jon Crowcroft and Peter Pietzuch for serving on my PhD viva committee.

I would like to thank the mentors and people I met during my internships at Google, for providing practical experience on real-world problems, and Jeff Mogul, whose paper about network latency requirements of cloud tenants I read early on in my PhD.

I would like to thank the researchers and fellow PhD students involved in the ITN METRICS project, who represented a great support network. I would also like to thank the Computer Laboratory and Graduate Students administration. I greatly enjoyed attending women@CL events and being part of its organising committee.

I would like to thank Churchill College, especially Rebecca Sawalmeh, who has always been prompt in helping me with whatever I needed. I am grateful to the many graduate students I met throughout the years in the Churchill MCR. I learnt something new from each one of them. I want to thank Gillian, Hansini, Marija, Shri, Anantha, Sam, Marco, Sai, Bill, Dominic and many others. Furthermore, I would like to thank Tiago, Kaspars, and Alex, for their cheerful support and encouragement, and with whom I shared excellent food from our home countries. I would also like to thank the friends I made during my internships, Konstantina, Maria and others, for brightening up my internship experience.

Lastly, I would like to thank my parents, Delia and Emil, my brother Marius, my sister-in-law Ruxandra, and all members of my family, who have been continuously with me throughout these years, especially my mother. My parents and my brother have accompanied me day and night, home or around the world, in this journey, with loving support and advice. This dissertation is dedicated to them.

I would like to express my gratitude for the financial support of EU FP7 ITN METRICS, EU Horizon 2020 SSCICLOPS, EPSRC EARL, the Women Techmakers Scholarship, Churchill College, and the Computer Laboratory, for funding my studies and supporting my attendance at conferences.

To my family

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Networks represent an important component of modern computing, shaping an interconnected world. While high-level applications, such as games, music and video streaming, office tools, play an important role in people's lifes, the importance of communication between computers cannot be overstated. Without a fast and reliable means of communication, these applications would be running only on the local computers with no external input, safe for the information obtained through compact disks or memory sticks. The Internet has evolved over the years into such a fast and reliable means of communication. At the heart of this evolution have been the strong need of people to communicate with each other, and the information that people want to disseminate to the world. As a result, a person can contact anyone over the Internet through electronic mail, voice or video call, and information on just about anything is nowadays easily available worldwide to anyone with an Internet connection. These two key needs have fueled the development of networks over the years, making the Internet ubiquitous.

Services such as web search, social networking, online shopping, content streaming, instant messaging, video calling, digitised newspapers, books, or research articles, form the online landscape today. Because of the huge number of people that access these services, enormous computing resources are needed. This demand has lead to the development of specialised warehouse-scale computers (WSCs) [BH18] that are housed in *data centres*. Data centres contain many hundreds of thousands of such computers, powering the above-mentioned services and many more. But designing and operating such complex systems require expert knowledge. As a result, only a few companies, such as Amazon, Google, IBM, Oracle or Microsoft, develop and manage their own data centres. This complexity has given rise to *cloud computing*: businesses rent compute, storage and network resources from specialised companies, *cloud providers*, to power their services, instead of developing and maintaining their own infrastructure. Naturally, businesses require and expect predictability from the rented resources, which translates into *predictable performance* for their applications. The customer, or *tenant*, expectations are encoded into contracts called *Service Level Agreements (SLAs)*, with specific objectives, *Service Level Objectives (SLOs)*, defined in collaboration with the cloud provider. The objectives refer to quantifiable metrics, such as availability, throughput, or response time.

The infrastructure in data centres is shared amongst different tenants, giving rise to possible interference between different applications, which in turn can lead to unpredictable application performance. Interference can appear in multiple places in the data centre: at the servers (*hosts* or *end-hosts*), where tenant applications that run inside *virtual machines (VMs)* share the underlying server hardware, or in the network, with some tenants sending traffic that causes packets to queue behind each other in switches, increasing the packets' *network latency*, or *network delay*, or *one-way delay (OWD)* (the time a packet takes to travel from the source to the destination across the network) [MK15; BMP+17]. Several approaches to minimise interference have been proposed, both at the end-host and in-network. Avoiding or reducing the interference at the end-host can be achieved through thoughtful *cluster scheduling*: avoiding the colocation of applications that compete for the same end-host resources by placing these applications on different hosts that meet their resource requirements. In the network, interference effects can be avoided or reduced through *flow scheduling* [AYS+13; POB+14; GNK+15; GSG+15; HRA+17] and traffic *load balancing* [ARR+10; BAA+11; AED+14]. Pinpointing the exact cause of the interference is as challenging as data centre systems are complex [RBB+18].

In my dissertation, I focus specifically on network latency in data centres, as an intrinsic property of the data centre network, and as a consequence of network interference. I focus on three aspects: how to best measure network latency, assessing and modelling its impact on typical cloud applications' performance, and how to mitigate its effects.

In Chapter 3, I show how the Precision Time Protocol (PTP), through its software implementation PTPd [PTP18], can measure network latency and estimate packet loss. In Chapter 4, I present *PTPmesh*, a cloud monitoring tool for tenants, which uses PTPd as a building block. PTPmesh offers *end-to-end* measurements (VM-to-VM measurements) in cloud environments. To demonstrate the practicality of PTPmesh, I conduct measurement campaigns using PTPmesh in several data centres from different cloud providers, identifying different latency magnitude, latency variance and packet loss characteristics. Under normal conditions, latency in data centres varies between tens to hundreds of microseconds. Unexpected events, such as network congestion or packet loss, can lead to network latencies in the order of milliseconds.

In Chapter 5, I conduct an analysis of the impact of *network latency* on typical cloud applications' performance. I quantify experimentally the relationship between application performance and increasing network latency through controlled experiments conducted on custom testbeds where I artificially inject network latency. I show that even small amounts of network latency in the order of tens or hundreds microseconds can cause application performance loss for certain applications. I then build functions that predict application performance under different network latency values using the experimental results.

Previous work on providing network guarantees has sought to provide *bandwidth and-or tail latency guarantees* (§2.7.2.2), and, as a result, provide applications with *predictable performance*. In my work, I take the opposite approach. In Chapter 6, I use the predictions for application performance built in Chapter 5, and current measured network latency, as inputs to

a cluster scheduling policy for data centres. The policy places or migrates applications in order to achieve the best performance under the current network conditions. I call this type of cluster scheduling *latency-driven, application performance-aware, cluster scheduling*. To demonstrate the practicality of my policy, I implement the *NoMora* cluster scheduling architecture as an extension of the Firmament cluster scheduler [GSG+16].

The thesis of this dissertation is that application performance in data centres can be improved through cluster scheduling of applications informed by measurement-based application performance predictions combined with measurements of current network latencies.

1.1 Contributions

In this dissertation, I make four principal contributions:

1. My first contribution is **showing how the Precision Time Protocol (PTP), through a software implementation PTPd [PTP18], can be used to measure network latency and estimate packet loss in data centres**. I propose *PTPmesh*, which uses PTPd, as a tool for cloud tenants to measure network conditions in data centres (see Chapter 3 and Chapter 4).

2. My second contribution is a **measurement study of network conditions in several data centres across different cloud providers** (Amazon Web Services [1], Google Cloud Platform [2], Microsoft Azure[3]) using PTPmesh. The study reveals different profiles in terms of latency magnitude, latency variance and packet loss across data centres and cloud providers (see Chapter 4).

3. My third contribution is **showing that small network delays in the order of tens or hundreds of microseconds can impact substantially application performance**. I study how network latency affects application performance for a wide range of applications, from a simple Domain Name System (DNS) client application to complex data processing and machine learning frameworks (Spark [Spa], STRADS [KHL+16], Tensorflow [ABC+16]). I do this through custom experiments where I inject artificial delay in the networked system. Furthermore, I model the relationship between application performance and network latency, building **functions that predict application performance dependent upon network latency** (see Chapter 5).

4. My fourth contribution is a **latency-driven, application performance-aware, cluster scheduling policy** that exploits dynamic network latency measurements between pairs of hosts combined with application performance predictions dependent upon network

[1] `https://aws.amazon.com/`
[2] `https://cloud.google.com/`
[3] `https://azure.microsoft.com/en-gb/`

latency to place or migrate applications in the data centre, with the goal of providing improved application performance. I implemented the NoMora cluster scheduling architecture as an extension of the Firmament cluster scheduler framework [GSG+16]. I show that the policy improves overall average application performance (see Chapter 6).

All of the measurement experiments, algorithms, implementations and analysis described are results of my own work. The ideas, experiments and results presented in this dissertation have been discussed with my supervisor, Dr. Andrew Moore, who provided guidance throughout the work. Malcolm Scott provided invaluable assistance for setting up the experimental testbeds in the Computer Laboratory's model data centre. Salvator Galea helped me with setting up an experimental testbed used in Chapter 3. Noa Zilberman developed the NRG tool [NZM17] (§2.1.4), which I use in Section 2.5 and Chapter 5. Many of the plots presented in this dissertation were generated using the `matplotlib` library starting from scripts initially written by Ionel Gog and Malte Schwarzkopf.

1.2 Dissertation outline

This dissertation is structured as follows:

- **Chapter 2** gives an overview of the background related to the work presented in this dissertation, and describes my preliminary experiments supporting the work done in this dissertation. The chapter first introduces general notions about network measurement, classical network monitoring tools and techniques, and network emulation and simulation tools. Next, it describes mechanisms for time synchronisation between computers over the network. The chapter then reviews the most important data centre network architectures, data centre applications, data centre network traffic characteristics, and data centre network monitoring systems. Further, the chapter presents my preliminary study of network latencies in data centres, and my experiments showing that network latency impacts application performance. Additionally, it presents a principled analysis of the baseline latency contributors at the end-host and in the network. Lastly, the chapter discusses the characteristics of cluster and cloud workloads, and summarises the traits of cluster schedulers.

- **Chapter 3** presents my investigation of PTP's ability to measure network latency and packet loss through a set of experiments conducted on local testbeds. First, it presents the validation of the use of one-way delay as a measure of network latency. It then presents my definition for a metric for computing packet loss ratio over a defined period of time depending on the number of messages exchanged between the PTP master and the PTP client.

- **Chapter 4** introduces *PTPmesh*, which uses PTPd as a building block, as a monitoring tool for cloud tenants. PTPmesh is deployed in data centres of three cloud providers (Amazon AWS, Google Cloud Platform, Microsoft Azure). An analysis of the collected data is carried out, offering insights into the characteristics of network latency magnitude, latency variance and packet loss in different data centres.
- **Chapter 5** describes the experimental setup and the set of experiments carried out for different cloud-based applications to see how they react to increased network latency. Further, the chapter presents functions that predict application performance depending on network latency, which are constructed based on the experimental results.
- **Chapter 6** describes the cluster scheduling architecture *NoMora*, which combines dynamic network latency measurements between hosts and application performance predictions dependent upon network latency constructed in the previous chapter, to place or migrate applications in a data centre. It then evaluates the cluster scheduling policy, showing that it improves overall average application performance for a well-known cluster workload [RTG+12].
- **Chapter 7** draws conclusions arising from the work of this dissertation, and presents directions for future work.

1.3 Related publications

Parts of the work described in this dissertation are part of the following peer-reviewed publications:

[PM18] Diana Andreea Popescu, Andrew W. Moore, "A First Look At Data Center Network Conditions Through The Eyes of PTPmesh", In: *Proceedings of the 2018 IFIP/IEEE 2nd Network Traffic Measurement and Analysis Conference*, Vienna, Austria, 26-29 June 2018.

[PM17] Diana Andreea Popescu, Andrew W. Moore, "PTPmesh: Data Center Network Latency Measurements Using PTP", In: *Proceedings of the 2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Banff, Canada, 20-22 September 2017.

[ZGP+17] Noa Zilberman, Matthew Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wójcik and Andrew W. Moore, "Where Has My Time Gone? ", In: *Proceedings of the Passive and Active Measurement: 18th International Conference, PAM 2017*, Sydney, NSW, Australia, March 30-31, 2017.

Parts of the work described in this dissertation are part of the following technical report:

[PZM17] Diana Andreea Popescu, Noa Zilberman, Andrew W. Moore, "Characterizing the impact of network latency on cloud-based applications' performance", In: *Technical Report, Number 914, UCAM-CL-TR-914, ISSN 1476-2986*, November 2017, Computer Laboratory, University of Cambridge, UK.

Parts of the related work described in the Background chapter (Chapter 2) of this dissertation are part of the following peer-reviwed publication and in submission publication:

[PM16] Diana Andreea Popescu, Andrew W. Moore, "Reproducing Network Experiments in a Time-controlled Emulation Environment", In: *Proceedings of The 8th International Workshop on Traffic Monitoring and Analysis (TMA 2016)*, 7-8 April 2016, Louvain-La-Neuve, Belgium

[KPA+18] Jan Kucera, Diana Andreea Popescu, Andrew W. Moore, Jan Korenek, Gianni Antichi, "Seek and Push: Detecting Large Traffic Aggregates in the Dataplane", 2018

During the course of my PhD, I have also co-authored the following publications that do not contribute directly to the work described in this dissertation:

[PAM17] Diana Andreea Popescu, Gianni Antichi and Andrew W. Moore, "Enabling Fast Hierarchical Heavy Hitter Detection using Programmable Data Planes", In: *Proceedings of ACM SOSR '17 Proceedings of the Symposium on SDN Research*, Santa Clara, CA, USA, April 3-4, 2017 (extended abstract)

[PM15] Diana Andreea Popescu and Andrew W. Moore, "Omniscient: Towards realizing near real-time data center network traffic maps", In: *Proceedings of ACM CoNEXT Student Workshop'15*, Heidelberg, Germany, December 1, 2015 (extended abstract)

[PG16] Diana Andreea Popescu, Rogelio Tomas Garcia, "Multivariate Polynomial Multiplication on GPU", In: *Proceedings of the International Conference on Computational Science 2016, ICCS 2016*, 6-8 June 2016, San Diego, California, USA

Chapter 2

Background

In this chapter, I present the background relevant to my dissertation. Firstly, I present the basic network measurement definitions and network monitoring techniques and tools, and then network emulation and simulation tools (§2.1). Secondly, I describe mechanisms for timekeeping on computers (§2.2). Thirdly, I present the network architecture of today's data centres, the typical applications that run in data centres, the characteristics of network traffic in data centres, and data centre network monitoring systems (§2.3). Next, I present a preliminary measurement study of latencies in data centres (§2.4), and my preliminary experiments of the impact of network latency on application performance (§2.5). Furthermore, I discuss the latency values observed in today's networked systems (end-host and in-network) (§2.6). Lastly, I present the characteristics of cluster workloads, and describe the state-of-the-art for cluster scheduling (§2.7).

2.1 Network measurement

2.1.1 Network measurement definitions

In my dissertation, I look at Internet Procotol (IP) [Pos81] networks, with the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) [Bra89] as transmission protocols. Before looking into measurement techniques, I give a brief overview of traffic properties which are usually measured, as presented in [CK06].

Packet delay is an additive metric and is the sum of routing delay (time spent inside a router), transmission delay (time needed to put a packet onto a link) and propagation delay (time needed by a packet to traverse the link from one end to another). Further, the routing delay can be decomposed into packet processing delay (time to determine the output port for the packet), queueing delay (time spent waiting in router's output queues) and other additional delay, such as marshalling and unmarshalling. The queueing delay can be seen as a measure of congestion on the output link. Another property is the rate of *packet loss*: the phenomenon which occurs

when a network device drops the packet due to congestion, or because the packet is identified corrupted and then dropped. *Throughput* is the rate at which traffic flows through the network and is measured in bits per time unit. *Packet jitter* represents the variability of packet interarrival time. An important property for application performance is *goodput*, which is the rate at which the application endpoint successfully receives data. There are other properties that can be used to characterise the traffic: time series of byte counts (to quantify the workload represented by traffic), and the distribution of packet sizes encountered. Another property is the *ON/OFF activity* in network traffic (ON state represents activity, OFF state means silence). The activity can be viewed at three different levels: the packets themselves are an ON/OFF process, packets form *trains* from a source to a destination (defined by a given interarrival threshold), and a collection of trains form a *session*.

A flow can be defined as a set of packets having the same predefined properties, and that are exchanged between two endpoints.

There are two types of measurement methods: *active* and *passive* methods [CK06]. Active measurement methods imply injecting additional packets into the network and observing their behaviour. The best known active measurement tools are *ping* and *traceroute* (see Table 2.1). Network latency from a source host to a destination host is often measured as Round-Trip Time (RTT) using *ping*. A disadvantage of active methods is that they create additional network load and can thus bias the measurements. On the other hand, passive measurement methods rely solely on observing the traffic without generating additional traffic.

2.1.2 Network monitoring techniques and tools

Table 2.1 presents classical network monitoring techniques and tools, and the measurements they offer. One of the most well known method to get network statistics is through the Simple Network Management Protocol (SNMP) counters [CFS+90]. The information that can be obtained by polling these counters regularly is the number of bytes received and sent on each interface of the network device or the number of packets received and sent on each interface. Although maintaining these counters does not have a significant performance overhead on the network device, they offer only a course-grained view of the network traffic, because they are limited by the polling interval (typically 300s), and by imprecise timestamps. The timestamp is supplied externally, and it is limited in numerical range, leading to the need to detect roll over. SNMP's resolution is sufficient for low throughput traffic.

Packet sampling, as used in NetFlow [Cisc], sFlow [sFl], PSAMP [JQC09], entails capturing only a subset of packets to reduce the number of records [CK06]. Packet sampling can be done using a constant or variable sampling rate. In the case of constant sampling, there are several possibilities: i) random sampling (packets are sampled with a fixed probability $0 < p < 1$); ii) deterministic sampling (packets are sampled periodically, meaning every Nth packet is sampled); iii) stratified sampling (packets are first divided into subsets, and then sampling is

Tool	Measurement
Ping	RTT; packet loss ratio
Traceroute	route 'guess'; RTT
SNMP [CFS+90]	switch port counters
NetFlow [Cisc]	flow counters
sFlow [sFl]	packet sampling
iperf [ESn]	throughput
Cisco IP SLA [Cisb]	RTT (average); one-way delay; packet loss
Port mirroring [Cisa]	copies of all packets from a port
NTP [MBK+10]	round-trip delay
PTP [IEE08]	master-to-slave delay; slave-to-master delay; estimated one-way delay

Table 2.1: Classical network monitoring tools.

applied within each subset). Another type of sampling is *trajectory sampling* [DG01; CK06]. In this technique, if a packet is selected for sampling at some device in the network, then it will be selected at all the other devices in the network. Amongst the uses of trajectory sampling are obtaining packet delays (important for SLAs), or tracing denial of service attacks.

NetFlow [Cisc] is a standard introduced by Cisco to monitor IP flows, which are usually identified by the 5 tuple (source and destination IP address, source and destination port and protocol number). The active TCP and UDP flows are kept in a cache. When a packet is received at the switch, NetFlow checks to see if the packet pertains to a cached flow by matching on the header fields. If it does, then the associated flow counters are incremented, and if it does not, a new flow entry is created in the cache. For deciding when to send flow records to the collector for analysis, several policies can be configured: i) on a TCP flow completion, which can be detected when seeing a packet with a FIN or RST flag; ii) when a flow has been idle for a configured timeout; iii) if a hard timeout is configured; iv) when the cache is full and an entry must be evicted. Timeouts can be specified at granularity of seconds. Sampled NetFlow [Cisd] can sample 1 in N consecutive packets that traverse the switch.

sFlow [sFl] is a standard implemented in most new switches to provide packet sampling and port counter sampling. For packet sampling, a switch selects 1 in N packets on each input port. The sampled packets' headers are forwarded to a collector along with metadata that includes the sampling rate, the switch ID, the timestamp of capture, the input and output port numbers.

Port mirroring [Cisa] involves copying all packets seen on a switch port to a different switch port for further analysis.

2.1.3 Network monitoring in SDN-enabled networks

Software Defined Networking (SDN) is a paradigm in which the control plane is separated from the data forwarding plane, enabling the centralisation of network control and offering the possibility of programming the network. The control plane is represented by a *controller* and the

data plane consists of networking devices, such as switches and routers. This separation is made possible by a programming interface, which allows the controller to communicate with the forwarding devices, for example to install forwarding rules at switches. OpenFlow [MAB+08] is the most popular such application programming interface (API). In an OpenFlow network, the controller can collect statistics about the flows (duration, number of packets, number of bytes) by polling the switches. These statistics can be either per-flow values or aggregates across multiple flows that match a rule. SDN monitoring tools use flow events that the controller receives (new flow - `PacketIn` message; termination of a flow - `FlowRemoved` message) and the statistics collected by the controller. I discuss several such tools in the following paragraphs.

OpenTM [TGG10] is a traffic matrix (TM) estimator for OpenFlow networks. It determines the current active flows in the network based on flow initiation and termination events. OpenTM uses routing information from the OpenFlow controller to discover the flow paths, and then it periodically polls switches on the flow path, obtaining byte and packet counters. OpenTM assumes that all the packets of a flow follow the same path in the network.

FlowSense [YLZ+13] is a monitoring tool that uses OpenFlow control messages to determine average link utilisation in OpenFlow-enabled networks. When a flow expires, the controller receives information about the duration and size of that flow. The link utilisation is computed only at certain times, *e.g.*, when all the flows on a link have expired. In the case of long flows, or if rules have large timeouts, the link utilisation is rarely computed. While the FlowSense approach does not present any overhead in terms of additional messages injected in the network, the delay between obtaining average link utilisation estimates can be 10 seconds with 90% accuracy. For proactive rules, *PacketIn* messages are not triggered. Wildcard rules result in a smaller number of *FlowRemoved* messages. As a result, these two types of rules limit the frequency of the link utilisation computation. Given these limitations, an adaptive monitoring method was proposed in PayLess [CBA+14], which uses the *PacketIn* and *FlowRemoved* messages in a similar fashion to FlowSense, but it additionally polls the switches using `FlowStatisticsRequest` messages. In doing so, it obtains flow statistics more often than FlowSense, which gathered statistics only when a flow terminated. The switches are polled at an interval determined by the byte count of the flow. If the flow does not change much, the polling timeout is increased, while if there is a significant difference between the byte count at two polling times, the polling timeout is decreased. This method can be used to determine link utilisation, and is more accurate than FlowSense. It is less accurate than polling at a fixed interval, but the total number of OpenFlow messages used is lower.

DREAM [MYG+14] is a network-wide measurement architecture that uses OpenFlow to coordinate the measurement devices. DREAM implements an algorithm for allocating switch memory resources depending on the measurement task needs, traffic and expected accuracy, multiplexing the resources both temporally and spatially.

OpenSketch [YJM13] is a measurement architecture inspired by the software-defined networking paradigm, dubbed software-defined traffic measurement. In the switch data plane, a packet

goes through 3 basic blocks: hashing (certain packet fields are hashed), classification (matching on fields according to predefined rules), and counting (gathering statistics). The control plane manages the data plane, and configures the measurement tasks according to available resources. OpenSketch uses sketches, data structures from streaming algorithms, to store information about packets. Sketches have two main advantages compared to flow-based counters: low memory usage and the possibility of setting the desired accuracy in relation to the memory used.

The information and the granularity provided by OpenFlow counters is limited, hence other approaches sought to use known ways of acquiring statistics from the network (*e.g.*, sampling [SKD+14], port mirroring [RSD+14]). Still, SDN monitoring has some benefits. It offers the possibility of greater control and reduced human overhead for configuring each switch individually to collect network statistics. Another benefit of SDN for monitoring is the centralised view of the network, which allows for a better allocation of network resources [MYG+14] to gather data. Also, the centralised view can help in reducing the collection of redundant data (*e.g.*, the same flow being sampled at several places in the network).

The monitoring approaches presented do not have the granularity necessary for data centres. This drawback lead to the increasing use of programmable switches for monitoring tasks [LMK+16a; LMK+16b; LMV+16; HW16; PAM17; SNR+17; YJL+18; HLB18; GHC+18]. FlowRadar [LMK+16a] keeps track of all the flows in the network with their associated counters, and exports this information periodically to a remote collector, which ultimately uses them for different monitoring applications targeted to datacenters. UnivMon [LMV+16], ElasticSketch [YJL+18] and SketchLearn [HLB18] use sketch-based data structures in the dataplane to record network traffic statistics that are exported at fixed time intervals to the control plane that processes them to perform different measurement tasks. HashPipe [SNR+17] and [PAM17] focus on determining the largest flows (heavy hitters). Sonata [GHC+18] proposes a query interface for network telemetry, uses sketches in the dataplane, and the controller zooms-in the network traffic of interest by refining the network query.

2.1.4 Network simulation and emulation frameworks

To understand the impact of network latency on application performance, one has to have the ability to recreate diverse network conditions. In this context, network simulation and emulation frameworks represent powerful tools for researchers, as they offer the possibility of replicating controlled and diverse network conditions, and of setting up experimental environments similar to real testbeds. However, the experimental results obtained on such frameworks are not always accurate due to different factors.

Simulation based frameworks, such as ns-2 [IH08], ns-3 [HRF+06] or OMNET++ [VH08], are often employed by researchers in order to evaluate their prototypes. These popular examples employ an event-driven simulation clock, and simplified models for hardware and network pro-

tocols. Simulations are usually lengthy in time, because they have to simulate every event, leading to serialisation of events for the case in which events were run in parallel in the real run. Also, the more realistic the simulation is, the longer the simulation runtime will be, with more events to run. However, not all simulators are event-driven, having deterministic behaviour. Some are stochastic, meaning that the simulation needs to be run repeatedly to obtain sound results. Most often, the simulator's experimental fidelity is simplified due to the use of network models.

Emulation brings more realism to the experimental results by replicating one or more parts of a system under study. This allows the use of unmodified applications and operating systems. Network link emulators, such as NetEm [Hem] and DummyNet [CR10], are essential tools used in network emulations. They vary different properties such as bandwidth, delay, jitter, packet loss, packet duplication, or packet reordering, on the outgoing interface. These tools also serve as building blocks for large-scale network emulators (DieCast [GVM+11], Mininet [HHJ+12], or SELENA [PRM14]), or in testbeds to change the network conditions. As such, their accuracy in replicating various network conditions is very important.

NetEm NetEm [Hem] is an enhancement of the Linux traffic control facilities, built using the existing Quality of Service (QoS) and Differentiated Services (diffserv) facilities in the Linux kernel. An important feature of NetEm for the work in this dissertation is the ability to artificially delay packets. The delay can be constant, or it can follow a predefined distribution (uniform, normal, Pareto, Pareto-normal), or a user defined one. Additionally, NetEm can limit the bandwidth using the Token Bucket Filter from the Linux traffic control (TC). Several studies [NR09; SMA+10; JLH+11; HF15] have shown that NetEm does not accurately introduce the specified delay. The values introduced have high variance (because of *e.g.*, operating system (OS) scheduling, interrupts), impacting the accuracy of small delays less than several hundreds of microseconds, typical for data centres. NetEm by default can only delay the packets on the outgoing interface. As I am interested in artificially delaying packets that are sent or received to recreate diverse network conditions, this represents a limitation. To be able to delay packets also on the incoming interface, an emulated software device must be used, called the Intermediate Functional Block (ifb) [NR09]. All incoming packets are redirected to this interface, where the delay is applied. However, this introduces an additional overhead of at least 5μs [NR09], which is significant considering the current latencies in the cloud, as presented in Section 2.4 and Chapter 4.

An approach to overcome these issues is to use a hardware-based tool rather than a software one.

NRG The Network Research Gadget (NRG) [NZM17] is a latency appliance that implements traffic control mechanisms in hardware, replicating the functionality of NetEm [Hem]. The hardware implementation provides higher resolution and better control over latency than

software-based traffic control systems such as NetEm. The latency injection has a resolution of 5ns and can range from zero to a maximum value dependent on the configured line-rate. The appliance can add up to 700μs of latency at full 10Gb/s rate, and up to 7s at 100Mbps. In addition, NRG adds only 700ns of base latency, compared to several microseconds as is commonly the case with NetEm. NRG introduces both a constant latency and variable latency to recreate a predefined latency distribution. The supported latency distributions are: flat, user-defined, uniform, normal, Pareto, and Pareto-normal distributions. The definitions for the last three distributions are based upon NetEm's distributions [Hem]. The user defined distribution allows the specification of any distribution, and can be used to recreate the latencies measured in the cloud. Latencies are injected independently for each direction of each port, thus client to server and server to client added latencies are completely independent, replicating the direction-independent latency experienced by packets due to in-network congestion. There is no packet reordering within NRG. A packet P is delayed in a queue inside the appliance for a given amount of time d chosen randomly from a given delay distribution. The packet delay d within the queue is independent of the inter-packet gap. The per-packet added latency presumes independence of each arrival. This provides a simplification that permits practical implementation. NRG can be deployed on the NetFPGA SUME board [ZAC+14].

Given my focus in this dissertation on data centre latency scale, in the order of tens or hundreds of microseconds, I use NRG in Section 2.5 and Chapter 5.

2.2 Timekeeping on computers

Having the clocks of computers that communicate over the network synchronised has been of great importance since the beginning of computer networks. Clients and servers of distributed file systems, such as the Network File System, need a common time to have an ordering of the operations done on the files. Distributed databases (*e.g.*, Spanner [CDE+12]) leverage time-stamps to order transactions, with the transaction latency being bounded by the clock's uncertainty. Another example is the *make* utility's behaviour. When doing an incremental build, the *make* utility uses the system time and the time the object files were created to determine which source files have been modified in order to avoid recompiling all the files. In the case that a program is compiled on a computer, and later the source and object files are transferred to a different computer and recompiled there, if the system times of the two computers are different, *make* will detect the difference and report it as a warning to the user. The user then will have to remove the old files and build everything from scratch. Having the clocks of the two computers synchronised would have solved this issue. With the switches and hosts clocks synchronised, correlating the events detected at different locations in the network can be used to detect network-wide traffic events [MYG+14; MYG+16; HCG+18]. Network latency from a source host to a destination host is often measured as RTT (*e.g.*, the *ping* utility) [GYX+15; ALZ16]. If the hosts clocks are synchronised, then the one-way delay (OWD) can be measured

just by sending a packet from the source host to the destination host. These are all examples of situations that would benefit from having synchronised clocks.

2.2.1 Terminology

Clock synchronisation defines the process of adjusting a clock to a different clock's value. The *clock offset* defines by how much a clock's value is different from another clock's value.

The *accuracy* of a clock defines how close the clock's value is to the true time, while the *precision* defines the bound for the difference between the clocks' values.

A clock is driven by a quartz oscillator at a given frequency. The frequency of the clock can change, meaning it does not run at the same rate as another clock, desynchronising from each other. This phenomenon is called *clock drift*, and it can be caused by heat, poor quality materials, or vibration.

Clock synchronisation uses four timestamps: two from the client (clock to be adjusted) and two from the server (reference clock), determined through packet exchanges. These four timestamps can be processed in different ways [MBK+10; IEE08; LWS+16; GLY+18].

2.2.2 The Network Time Protocol (NTP)

The first protocol for timekeeping on computers was the *Time Protocol* [PH83]. It is a simple protocol, where a computer connects to a server over TCP or UDP to query the time. The server sends back its current time, and closes the connection. Unix-based operating systems used the *rdate* utility to synchronise their clocks, which can set the time of the computer to the time received from the server. The sudden change in system time can affect certain programs' behaviour. Because of this, the Network Time Protocol (NTP) [MBK+10] was introduced. In NTP, the clock is adjusted gradually. NTP offers a monotonic count, meaning that the clock does not go backwards when corrected.

The Network Time Protocol (NTP) [MBK+10] synchronises hosts and routers clocks over the network. NTP's best case accuracy is in the low tens of milliseconds over the Internet, and sub-millisecond over LANs. NTP uses UDP as a transport protocol. Commonly, NTP is implemented in software as a daemon process `ntpd`, so its precision is affected by different OS-related artefacts (*e.g.*, context switching, software timestamps).

An NTP client regularly polls several NTP servers. A client sends a packet at timestamp T_1 (*originate* timestamp). The server's *receive* timestamp is T_2. The server then sends a response packet at T_3 (*transmit* timestamp), which is received by the client at T_4 (*destination* timestamp). The protocol has two modes: in the one-step mode, the transmit timestamps are sent in the transmitted packets (Figure 2.1). In the two-step mode, they are sent in the next transmitted

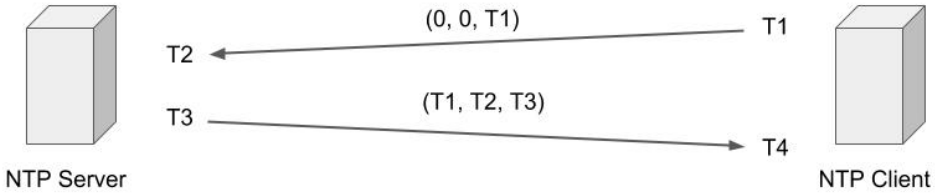


Figure 2.1: NTP protocol one-step mode.

packet. Based on these timestamps, the client computes the clock offset θ and the round-trip delay δ.

$$\theta = \frac{[(T_2 - T_1) + (T_3 - T_4)]}{2} \tag{2.1}$$

$$\delta = (T_4 - T_1) - (T_3 - T_2) \tag{2.2}$$

The offset and delay samples go through a filtering algorithm. The sample with the minimum delay amongst the last eight samples is selected. This sample remains selected until a different sample with lower delay appears.

The clock discipline algorithm adjusts the phase (coarse adjustment) and frequency of the clock, and it is described in detail in [Mil10]. The discipline is implemented as a feedback control whose inputs are the timestamp of a reference clock and the timestamp of the system clock. The difference between the two timestamps enters the clock filter (described in the previous paragraph), whose output is fed into the phase correction and frequency predictor stages. The frequency predictor stage uses a hybrid algorithm that combines a phase-lock loop (PLL) and frequency-lock loop (FLL). The frequency predictions of the two components are weighted differently depending on the conditions, with the PLL prediction being more important under increased network jitter (caused by network congestion), and the FLL prediction being more important under oscillator wander (caused by temperature variations). In PLL mode, the phase predictor is the offset amortised over time, while in FLL mode the phase predictor is not used. The phase error is upper bounded by half of the RTT between the client and the server [CK06]. In PLL mode, the frequency predictor is an integral of the offset over past updates, while in FLL mode the frequency predictor is a fraction of the current offset divided by the time since the last update. A client sends messages to each server with a poll interval of 2^{τ} seconds. The *poll exponent* τ is dynamically adjusted to maintain clock accuracy and to minimise network overhead, and ranges from 4 (16s) to 17 (131072s).

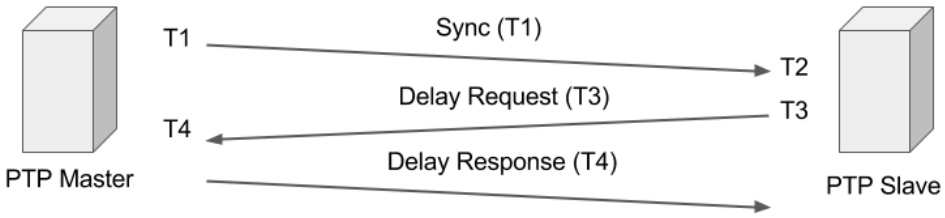


Figure 2.2: PTP protocol.

2.2.3 The Precision Time Protocol (PTP)

The IEEE 1588 Precision Time Protocol (PTP) [IEE08] is a standard protocol used to synchronise clocks over a network and it can achieve sub-microsecond precision. The master clock provides the reference time for the slave clocks. A *grandmaster* is chosen from the available clocks in the network. The grandmaster will be the root of a tree formed out of devices that are PTP-enabled. Each element of the tree is both a slave to its parent and a master for its children.

There are several types of PTP clocks. The simplest type is the *ordinary clock*, which is an end device that has only one network connection, and can act as a master or a slave. A *boundary clock* has a slave port, receiving the time from the master clock, and master ports, disseminating the time to other slaves. Another type of clock is the *transparent clock*, which timestamps incoming and outgoing messages and updates the correction field in the messages to account for the delay across the device. The mechanisms used by the last two types of clocks ensure the scalability of PTP networks.

The PTP protocol message sequence is depicted in Figure 2.2. A PTP *master* sends a *Sync* message. The time when the Sync message was sent (T_1) is recorded at the master and sent to the slaves. If the master does not have the ability to embed T_1 in the Sync message, it sends an additional message after the Sync message, *Follow_Up*, that contains T_1. A PTP *slave*, or *client*, records the time when it received a Sync message (T_2). The difference between the send and receipt times represents the *master-to-slave delay,* d_{m2s}:

$$d_{m2s} = T_1 - T_2 \tag{2.3}$$

A PTP slave sends a *Delay Request* message. The slave records the time when the Delay Request message was sent (T_3), while the master records the receipt time (T_4). The difference between the send and receipt times of the Delay Request messages represents the *slave-to-master delay,* d_{s2m}. The master will reply with a *Delay Response* message which contains the receipt time T_4, thus:

$$d_{s2m} = T_3 - T_4 \tag{2.4}$$

By assuming that the propagation delays master-to-slave and slave-to-master are symmetric,

which usually translates to paths being symmetric, the *one-way delay* is computed as half of the sum of the two delays.

$$\text{OWD} = \frac{d_{m2s} + d_{s2m}}{2} \tag{2.5}$$

The time difference between the master and slave clocks represents the *clock offset* from master and is computed as a difference between the master-to-slave delay and the one-way delay.

$$\text{offset} = d_{m2s} - \text{OWD} = \frac{d_{m2s} - d_{s2m}}{2} \tag{2.6}$$

In the case that the master-to-slave and slave-to-master delays are asymmetric (due to network congestion for example), the clock offset will suffer perturbations and the precision of the clock synchronisation will be affected.

The messages sent by PTP fall in two categories: *event* and *general*. Messages like Sync and Delay Request message are event messages, whose send and receipt timestamps are used to compute the adjustment of the slave clocks, and thus the timestamps need to be accurate. Messages like Announce, Follow-up and Delay Response are general messages, and do not require accurate timestamps. Event messages are sent on port 319, while general messages are sent on port 320. PTP messages are sent using multicast messaging [Dee89], but devices can negotiate unicast transmission if desired. PTP messages are usually sent over UDP. PTP supports two delay measurement mechanisms: peer-to-peer and end-to-end. In the peer-to-peer mechanism, each network device is PTP-aware, and the time synchronisation operates between the end-host and the network device. In the end-to-end mechanism, which I use in this work, only the end nodes need to be PTP-aware.

The send and receipt timestamps for the PTP packets can be generated either by the host operating system's kernel (software timestamping), or by a dedicated hardware unit (hardware timestamping). The first type of timestamping has the advantage of being widely available, but the timestamps generated are less precise due to variable interrupt servicing latencies [CB06]. The second type of timestamping is precise, but requires special hardware. For example, Solarflare network interface card (NIC) [Solb] generates hardware timestamps for PTP packets using a dedicated time stamping unit which is driven by an oscillator. On the arrival or departure of a PTP packet, the unit generates a hardware timestamp which is passed by the NIC to the network device driver. Additionally, a PTP stack enabled by the NIC is running on the server to discipline the NIC's precision oscillator. A user space application can access the hardware timestamps for the received packets using the SO_TIMESTAMPING socket option available in the Linux kernel.

PTP uses various mechanisms to ensure that there is no interference in the clock synchronisation. Firstly, PTP can use hardware timestamping to eliminate the end-host delay caused by the network stack and variance due to interrupt service latencies [OLS08]. Secondly, PTP-enabled

switches that run as transparent clocks can modify a field in the PTP messages to account for the delay incurred across the switches. In this work, I do not use transparent clocks, as I want to leverage PTP's measurements to infer the actual network latency, which is affected by network conditions like congestion.

Software implementations of the PTP protocol are PTPd [PTP18] (open source) or TimeKeeper [Tim] (commercial). In Section 2.2.4, I describe PTPd, which I use in Chapters 3 and 4 as a building block for a data centre monitoring system.

2.2.4 PTPd

PTPd is a software-based system that uses software timestamps. It runs as a background user-space process. PTPd is lightweight, its CPU resource utilisation being less than 1% [CB06]. PTPd's precision is determined by the precision of sent and received messages timestamps. PTPd uses the Linux kernel's software clock. It adjusts the clock using the `adjtimex()` interface for clock tick-rate adjustment.

The PTPd clock discipline [CB06] was designed to counter the jitter determined by various factors (interrupt servicing, network queueing). A proportional-integral (PI) controller produces a tick-rate adjustment for the slave clock. The proportional term corrects the offset between the slave clock and the master clock, while the integral term corrects the rate difference between the slave clock and the master clock. The input to the PI is filtered using a Finite Impulse Response (FIR) low-pass filter for the offset to master (a two sample average), and a first-order Infinite Impulse Response (IIR) low-pass filter for the one-way delay (a modified exponential smoothing with a two sample average added), with a *stiffness* factor that controls the cutoff and phase of the filter. The FIR attenuates the high frequency noise from the input.

2.2.5 Other clock synchronisation mechanisms

Global Positioning System (GPS) [PEA+96] receiver antennas can be used for nanosecond-level precision clock synchronisation. GPS receivers provide Pulse-Per Second (PPS) and time encoding to NICs that can process the signal. However, only a few servers are equipped with PPS-capable devices. Due to data centre scale, it would be extremely expensive to have a full deployment in all data centres where GPS signals are provided directly to thousands of machines.

Attempts to address the data centre scale issue are the Datacentre Time Protocol (DTP) [LWS+16] and Huygens [GLY+18]. DTP [LWS+16] is a protocol that uses the physical layer (PHY) to synchronise the hosts clocks, with a single hop clock precision of 25.6ns and achieving 153.6ns clock precision for a data centre with six hops. It exploits the observation that a transmitter and a receiver are already synchronised in the PHY. It uses the gap between frames defined in the IEEE 802.3 standard to send messages for clock synchronisation. DTP is not immediately

deployable, since it requires PHY modifications in the hardware in the network data centre. Its advantages are the fact that the network load does not affect the clock synchronisation, and it also does not generate any additional network traffic. Huygens [GLY+18] is a software clock synchronization system that achieves synchronisation to within a few tens of nanoseconds. Being a software system, it is immediately deployable in data centres. Huygens uses NIC timestamps, but it does not require specialised switches to remove the network queueing delays. It uses statistical methods (Support Vector Machines (SVMs)) to remove the queueing delays and timestamp noise. Every server probes 10-20 other servers, and each server uses 5Mb/s bandwidth for probes. Huygens uses packet pairs, called coded probes. If the spacing between the packets at the receiver is close to the one at the sender, then the pair is retained. Another important factor that helps in achieving such a good precision is that Huygens leverages the network effect: it does not synchronise each pair individually, but instead it synchronises multiple pairs by looking at differences between clocks of servers that form a loop in the probing graph. Each client runs the coded probes and SVMs on the filtered probes to determine the clock offset and drift. A master gathers this information from each client, applies the network effect, computes a consensus of the time, and then distributes it to all the clients.

2.3 Data centres

2.3.1 Data centre network architecture

A data centre network architecture is comprised of the topology of the network that interconnects the servers, of the switches deployed in the network, of the end-host network configuration and of the communication protocols used. Companies do not reveal full details of their data centre architectures, since the performance given by their infrastructure can be an advantage over competitors, especially in the cloud computing business.

Network hardware and topology The most common topology used for data centres is *fat tree* [ALV08], which is based on Clos networks [Clo53]. Clos networks, originally designed for telephone circuit switches, are multi-stage circuit-switching networks, with three stages: the ingress stage, the middle stage, and the egress stage. Clos networks are strict-sense non-blocking networks, meaning that any input can be connected to an unused output without having to rearrange existing connections. Fat trees, on the other hand, are rearrangeable non-blocking networks, meaning that with a certain arrangement of the connections, any input can be connected with any unused output. An important feature of a network topology is the *bisection bandwidth*. The bisection bandwidth of a network is the bandwidth available between the two partitions when the network is partitioned in half. Full bisection bandwidth means that any input can communicate with any unused output at *full line-rate*. Non-blocking networks provide *full bisection bandwidth*. The edge of the network is usually *oversubscribed*. Fat tree topologies can offer full bisection bandwidth, but it may be difficult to achieve this while also avoiding

packet reordering in TCP flows [ALV08].

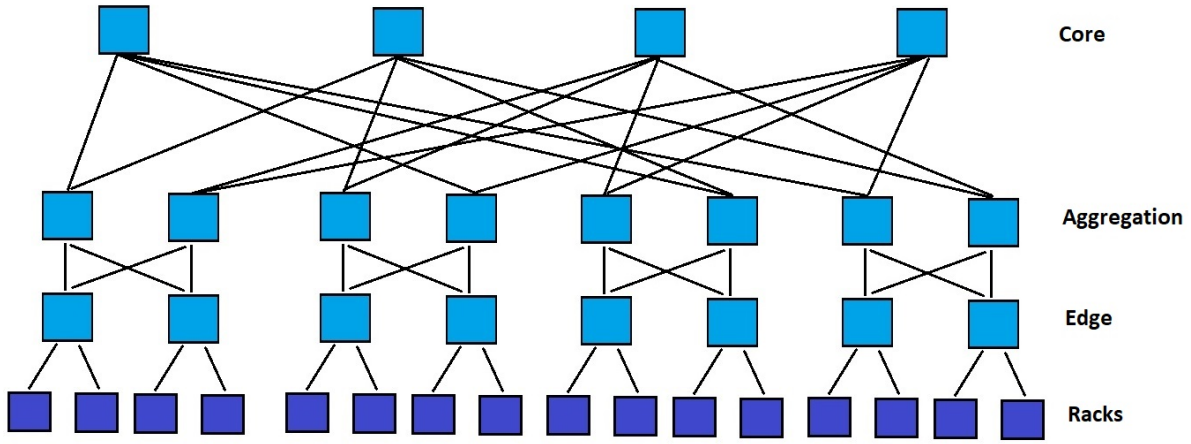


Figure 2.3: A data centre fat-tree topology.

Google [SOA+15], Microsoft [GJK+09] and Facebook [And14] data centre network topologies follow the fat tree design. A k-ary fat tree, shown in Figure 2.3, has 3 layers of k-port switches: core (the top layer), aggregation (the middle layer) and edge (the lowest layer, which is connected to the host layer). The hosts are grouped into *racks*, and are connected to edge switches, called *Top-of-Rack (ToR)* switches. The fat tree has k *pods*, and a pod contains two layers of switches, each layer having $\frac{k}{2}$ switches, and $\frac{k^2}{4}$ hosts. A k-port switch from the edge layer is connected to $\frac{k}{2}$ hosts and to $\frac{k}{2}$ switches from the aggregation layer. The fat tree topology has $(\frac{k}{2})^2$ core switches. A k-port core switch is connected to every pod, having its i-th port connected to the i-th pod such that an aggregation switch is connected to $\frac{k}{2}$ consecutive core switches. The fat tree topology provides multiple equal cost paths between every two hosts. Switches use Equal Cost Multipath routing (ECMP) [Hop00] to decide on which of the equal cost paths a flow should be sent. ECMP hashes the 5-tuple (IP source and destination address, source and destination port and protocol ID) of a packet, and based on the hash determines which path the flow will take.

The Google [SOA+15] network has evolved over the years. Google develops their own switches using 16×40Gb/s merchant silicon. In the Jupiter network, a ToR switch has 48×40Gb/s connections to hosts in the rack and 16×40Gb/s to the aggregation switches. Four such switches form a Middle Block (MB), which serves as building block in the aggregation block. The logical topology of an MB is a 2-stage blocking network, with 256×10Gb/s connections to ToRs and 64×40Gb/s connections to the spine. Each ToR connects to eight MBs with dual redundant 10Gb/s links. An aggregation block has 512×40Gb/s or 256×40Gb/s links towards the spine blocks. A spine block has six switches with 128Gb/s ports to the aggregation blocks. There are 64 aggregation blocks.

The Facebook data centre architecture [And14] has four planes of spine (core) switches. Each plane can accommodate up to 48 spine switches. Each fabric (aggregation) switch of each

pod connects to each spine switch within its plane. A pod has 48 server racks, with each pod being served by four fabric switches, one from every plane. A rack can have up to 192 hosts. Each ToR has 4×40Gbs/s uplinks. End-host have 10Gb/s connections. There is a 4:1 fabric oversubscription from rack to rack, with 12 spine switches per plane. Facebook also develops their own switches.

VL2 [GJK+09] is a 3-tier architecture where the core tier and the aggregation tier form a folded Clos topology. Other proposed data center network designs include Dcell [GWT+08], BCube [GLL+09], CamCube [ACR+10], Jellyfish [SHP+12], Xpander [VSD+16]. BCube [GLL+09] is a data centre network architecture based on a hyper-cube topology. Jellyfish [SHP+12] is a random graph topology designed to support easy incremental expansion of the data centre.

Hardware resource disaggregation is an emerging trend that will see the traditional rack replaced by pools of different resources (CPU, DRAM, disk) [GNK+16; SHC+18; Int18a] communicating over a high-speed network. This architecture presents several benefits: improved resource utilisation, failure isolation, and flexibility in adding or removing resources.

Network protocols The data centre protocol stack is based on the traditional TCP/IP stack, but it has evolved to deal with the challenges inherent to this type of environment: scale, cost, competing demands of applications (high throughput vs. low latency), unpredictable network traffic patterns. Most of the data centre operators today are using IP version 4 (IPv4) within their networks. Facebook uses only IP version 6 (IPv6) in their internal networks[1].

Traditional broadcast mechanisms such as Address Resolution Protocol (ARP) [Plu82] do not scale in data centres [GJK+09; NPF+09]. To solve this issue, data centres are managed via IP layer routing protocols, such as the Border Gateway Protocol (BGP) [RL95]. For example, Facebook uses BGP4, which is complemented by a centralised BGP controller that is able to override routing paths. Google developed its own protocol, named FirePath [SOA+15], which is a custom Interior Gateway Protocol (IGP). FirePath implements a centralised topology state distribution, and a distributed forwarding table computation. Microsoft's data centre protocol is thought to be similar to that of VL2 [GJK+09]: traffic originating from the edge switches is forwarded first to a randomly selected intermediate switch and then to the actual destination.

TCP's congestion control algorithm is not optimal for environments which have the characteristics of a data centre network (high-bandwidth, low latency). Consequently, TCP's performance suffers because of different issues, such as incast, bursty packet drops, and large queue buildup [AGM+10]. Thus, different TCP variants customised for data centres were developed over the years [AGM+10; AKE+12; VHV12]. The first such transport protocol variant was DCTCP [AGM+10]. DCTCP signals queue buildup earlier on through the use of the Explicit Congestion Notification (ECN) feature supported by certain switches. Source end-hosts estimate the fraction of packets marked through ECN, and deduce the amount of congestion. Google's data centres run a variant of DCTCP [SOA+15].

[1] Private communication

Moreover, clean slate designs have been proposed in this space [POB+14; HRA+17]. Fastpass [POB+14] is a centralised packet scheduler that aims to reduce in-network queueing. It proposes a timeslot allocation algorithm to determine when each packet is sent, along with a path assignment algorithm for each packet. Fastpass moves the queueing at end-hosts. NDP [HRA+17] is a radically different approach that requires a new end-host stack and new switches. It uses switches with small buffers. The senders send a full window from the start, with no initial handshake. When congestion occurs, the switches trim packets, removing their payload. The headers are sent to notify the receiver of which senders wish to send to it. Then, the receivers pull data from the senders who want to send to them, since the receivers have received the headers and know from whom to expect data.

2.3.2 Data centre applications

A key role in the data centre ecosystem is played by the applications that produce network traffic. Due to the scale of the input data and of the user demands, data centre applications are distributed. Data centre applications can be split in the following categories:

- Control and management applications: clock synchronisation (*e.g.*, Precision Time Protocol daemon [PTP18]), consensus and locking (*e.g.*, Chubby [Bur06]), and cluster management (*e.g.*, Borg [VPK+15]).
- Data storage and retrieval: distributed file systems (*e.g.*, GFS [GGL03]), distributed database systems (*e.g.*, Spanner [CDE+12]), key-value stores (*e.g.*, Memcached [Mem18]).
- Applications serving users' needs: data processing frameworks (MapReduce [DG04] style processing, graph processing (*e.g.*, Apache Giraph [Gir], Pregel [MAB+10]), stream processing (*e.g.*, Apache Storm [Sto]), machine learning analytics (*e.g.*, Tensorflow [ABC+16]), Web traffic [Pro18], search engine and social network backends. In addition to these, tenant applications running in VMs, which can be any of the previous applications or custom applications, have to be mentioned.

There are two aspects of the applications that are important for determining their networking requirements: the communication pattern of the application (which can be mapped to the underlying network) and the properties of representative workloads.

Communication patterns The most common communication patterns [CS12; KPT+12] are MapReduce [DG04], partition-aggregate (search engine and social networks backend) [AGM+10; KPT+12; CS12], dependent-sequential (constructing a user's home page in a social networking application) [KPT+12], star-like (machine learning parameter server [LAP+14]), and Bulk Synchronous Parallel (BSP) (*e.g.*, Pregel [MAB+10]).

In the *MapReduce* [DG04; ZCD+12] pattern, a mapper reads its input from the distributed file system, performs computations on the input read and then writes its intermediate result to disk.

A reducer reads the intermediate result from different mappers (the shuffle phase), performs computations on the data, and writes the output to the distributed file system. In the shuffle phase there are xy flows if there are x mappers and y reducers, and at least y flows for writing the final results.

In the *partition-aggregation* pattern [AGM+10; KPT+12], in order to provide an answer to a request received from a user, several responses from workers need to be aggregated. The aggregation tree can have multiple levels, with the leafs being the workers and the root being the final aggregator. The *dependent-sequential* pattern [KPT+12] entails that the next request is dependent on the previous request's results. These patterns are common in applications such as Web search and social network content backends.

A *Bulk Synchronous Parallel (BSP)* computation, named superstep, consists of concurrent computation, communication between worker processes, and barrier synchronisation. This pattern is common in graph processing frameworks [MAB+10; Gir].

Machine learning (ML) applications represent a common workload for data centres [HBB+18; ABC+16]. In general, an ML application fits a model to input data, and requires multiple iterations until the model's parameters convergence. Due to the huge amount of input data that has to be processed, ML frameworks have a distributed architecture [LAP+14; ZCD+12; KHL+16; XHD+15; ABC+16]. A machine learning framework usually has server nodes that store the globally shared parameters (parameter servers), and worker nodes that do local computations on their part of data or of the model, depending on the chosen approach. In a data parallel approach, the input data is partitioned across the machines and the ML model is shared. In a model parallel approach, the ML model is partitioned across the machines and the input data is shared. In the data parallel approach, each worker node can read and update all the model parameters, while in the model parallel approach, each worker node can access and update only its model parameter partition. In this context, the network plays an important role due to the inherent synchronisation between worker nodes and server nodes to update the model. If the computation is synchronous, after each iteration, the parameter serves aggregate the parameters from the workers. For example, if one of the workers is unreachable over the network or is slow to reply, the overall training time increases due to the wait for that worker's parameter updates. To lessen the importance of communication latency to the completion time of the training, some frameworks [ABC+16; KHL+16] use asynchronous communication, or bounded staleness synchronisation [XHD+15], where a certain degree of staleness in the parameters (meaning using parameters from previous iterations) is tolerated, but this can potentially lead to slower converge of the model [KHL+16].

Workloads There are a few studies which analyse the workloads of some data centre applications. One such study is the analysis and modelling of Memcached workloads based on data provided by Facebook [AXF+12]. The authors describe an analytical model that can be used to generate synthetic workloads whose properties are similar to the real world workloads, and which is implemented in the Mutilate load generator [LK14]. MapReduce workloads from

Facebook and Cloudera are discussed in [CAK12], providing insights about job size, storage access patterns, and cluster load. ML applications usually can be tested with well-known datasets, *e.g.*, MNIST dataset [LC10] for handwritten digit recognition, ImageNet [DDS+09] dataset for image classification tasks.

2.3.3 Data centre network traffic characteristics

Data centres network traffic characteristics are seen as sensitive information by companies, as they could reveal details about their network infrastructure to their competitors. As such, there is little information available on this topic. Still, there are three main studies [KSG+09; BAM10; RZB+15] which shed some light on this matter. Additionally, several papers [GJK+09; AGM+10; MPZ10; CZM+11; HKP+11] (see Table 2.2 and Table 2.3) present limited measurement studies from different data centres.

Benson *et al.* [BAM10] present a study of the network traffic characteristics of 10 data centres (3 university data centres, 2 private enterprise data centres, 5 commercial cloud data centres). The data sets used in characterising the network traffic are the following: network topology, packet traces from switches and SNMP polls. The data was collected over several weeks. There are several important findings. Firstly, perhaps predictably, the applications that run in the data centres depend on the organisation. There is a wide range of applications in each data centre: Web services, MapReduce, file storage, authentication, business applications, custom software applications, email and messaging. Secondly, there are important findings regarding flow sizes and interarrival times, as well as traffic locality. The number of active flows per second is under 10,000 per rack. 80% of the flows are smaller than 10KB in size. 80% of the flows have interarrival times of less than 1ms in private enterprise data centres, while 80% of the flows in university data centres and in most data centres have interarrival times between 4ms-40ms. 80% of flows are less than 11s long. Traffic originating from a rack has an ON/OFF pattern with properties that fit heavy-tailed distributions, and traffic that leaves the edge switches is bursty. In cloud data centres, 80% of the traffic coming from servers stays within the rack, because administrators colocate dependent applications, whereas in the case of university and private enterprise data centres 40-90% of the traffic leaves the rack. Link utilisation is higher in the core layer, while the edge layer is lightly utilised. A maximum of 25% of the core links are highly utilised (hot-spots). Losses are not correlated with high link utilisation, but are due to temporary bursts. Lastly, time of day/week influences link utilisation, especially in the core, and to a moderate degree in the other levels of the data centre.

The second main study about data centre traffic is [KSG+09]. Over 1 PB of measurement data was collected from 1,500 servers. The workloads were MapReduce style jobs using the Cosmos distributed file system. The servers were instrumented, and then socket level logs were collected along with user application logs. Similarly to the characteristics found in [BAM10] for cloud data centres, jobs that require high bandwidth are placed near each other (on the same

server, in the same rack, in the same VLAN) by the internal placement algorithm. Regarding traffic locality, the paper observes that there is a probability of 89% that servers within the same rack do not exchange traffic and 99.5% in the case of servers in different racks. Regarding congestion, 86% of the links have congestion periods of at least 10s, while 15% of the links have congestion periods of at least 100s. Over 90% of the congestion periods are less than 2s, but more than 1s. Regarding flow sizes, like the previous study [BAM10], most flows were short, 80% of the flows lasting less than 10s. Less than 0.1% last longer than 200s. However, more than half of the bytes are found in flows that last less than 25s. Another observation is that there is significant variability in the traffic matrix, both in magnitude and in the pairs of servers which exchange data. Even if the total traffic exchanged remains the same, the pairs of servers involved in this exchange change considerably. Moreover, the traffic experiences periodic short-term bursts, with the interarrival time 15ms at servers and ToR switches. The median arrival rate of all flows is 100 flows/ms. Lastly, there was no evidence of incast in the cluster.

The most recent study is on Facebook's data centre network traffic characteristics [RZB+15]. The data was collected using *Fbflow* (an internal monitoring system that samples packet headers with a sampling rate of 1:30,000), through port mirroring on the ToR, and mirroring of all traffic from one server. The traffic is one of the following: Web, MapReduce, MySQL, or traffic served from cache servers (leader and follower). Network traffic characteristics regarding flow sizes and traffic locality for Hadoop jobs are similar to the ones found by the previously mentioned studies. However, the traffic patterns for the other types of applications differ substantially from the ones described in [KSG+09; BAM10]. The majority of traffic is intra-cluster (57.5%, from caching follower servers), with only 12.9% intra-rack. Also, there is a significant portion of intra-datacentre and inter-datacentre traffic, in particular from the caching leader servers. The Hadoop traffic is more rack-local than other types of applications. Frontend traffic has minimal rack-local traffic, but significant intra-cluster traffic. Overall, the locality patterns are stable over time periods ranging from seconds to days. Regarding flow sizes, most Hadoop flows are short, while for the other types of services, they are long-lived, but internally bursty and do not carry a significant number of bytes. Cache flows are larger than Hadoop flows, and Web server flows are between the two. Facebook's use of load balancing is effective. It distributes the traffic across hosts, except in the case of Hadoop servers, which see jobs of different sizes, and traffic demands are quite stable over sub-second intervals. Consequently, heavy hitters' sizes are not much larger than the median flow sizes, and they change rapidly, making it hard to predict them. Packet sizes are small, median length for non-Hadoop traffic being less than 200 bytes, while for Hadoop the distribution is bimodal (1500 bytes or TCP ACKs size). The traffic does not exhibit ON/OFF arrival behaviour, unlike the traffic in the previous studies [KSG+09; BAM10]. Web servers and cache servers have 100s to 1,000s of concurrent connections, while Hadoop nodes have 25 concurrent connections on average, similar to the values reported in [KSG+09]. Median flow interarrival times are 2ms for Hadoop, and 3ms and 8ms for cache leaders and followers respectively. Cache followers and leaders communicate with 175-350 different racks

Study	Data centre	Duration	Workload	Measurements	Flow sizes
[BAM10]	10 data centres (3 university data centres, 2 private enterprise data centres, 5 commercial cloud data centres)	several weeks	Web services, MapReduce, file storage, authentication, business applications, custom software applications, email and messaging	• network topology; • packet traces from switches; • SNMP	• 80% of the flows are smaller than 10KB in size. • 80% of flows are less than 11 seconds long.
[KSG+09]	1500 servers	1 PB of measurement data	• MapReduce style jobs; • Cosmos distributed file system	• socket level logs; • user application logs;	• most flows are short, 80% of the flows lasting less than 10s • only less than 0.1% last longer than 200s • more than half of the bytes are found in flows that last less than 25s
[GJK+09]	1500 node cluster	-	data mining on petabytes of data	• SNMP • NetFlow	• 99% of flows are smaller than 100MB • 90% of bytes are in flows whose lengths are between 100MB and 1 GB
[AGM+10]	6000 servers in over 150 racks	1 month; 150 TB	web search and other services	• socket level logs; • packet level logs; • application level logs	• most background flows are small, but most of the bytes in background traffic come from large flows
[HKP+11]	• pre-production cluster with O(1K) servers running Dryad (Cosmos) • production cluster with O(10K) servers where the web search index is stored and where search results are assembled (IndexSrv)	76 hours and 114 terabytes of data	• a data mining workload for a large web search engine + jobs which are a mix of repetitive production scripts (e.g., hourly summaries) and jobs submitted by users • web search (latency sensitive)	estimate demand matrices and determine hotspot occurrence and predictability	• medium-sized flows
[MPZ10]	IBM Global Services • DC 1: 17000 VMs • DC 2: 68 VMs	10 days	-	latency measurements between each two servers; TCP incoming/outgoing connections	-
[RZB+15]	Facebook data centres	brief periods of time	Web services, MapReduce, MySQL, caching	• sampled traffic; • packet traces from switches; • packet traces from host	• Hadoop flows are short; • for other type of applications, flows are long-lived, internally bursty and do not carry a significant number of bytes.

Table 2.2: Data centre network traffic characteristics - part 1.

concurrently, while Web servers communicate with 10-125 racks. However, most of the traffic is destined to only a few 10s of racks. Link utilisation on links between hosts and the ToR is quite low, with the average 1-minute link utilisation less than 1%. Load varies significantly between clusters, a Hadoop cluster being five times more loaded than a Frontend cluster. The median utilisation between the ToR and aggregation switches is between 10-20% across clusters. At this level, the difference between clusters is not as significant as in the previous case, with the most loaded clusters being three times more loaded than the lightly loaded ones. Between the aggregation and core switches the utilisation is even higher.

To sum up, data centre traffic characteristics depend on the type of applications deployed, *e.g.*,

Hadoop, Web, caches, on diurnal patterns, and on the data centre operators' strategies for application placement and load balancing.

Table 2.3: Data centre network traffic characteristics - part 2.

Table 2.3 Beginning of table					
Study	**Concurrent flows**	**Interarrival times**	**Link utilisation**	**Communication between servers**	**Other**
[BAM10]	• the number of active flows per second is under 10,000 per rack	• 80% of the flows have interarrival times of less than 1 ms in private enterprise DCs • 80% of the flows in university DCs and in most DCs have interarrival times between 4ms-40ms. • traffic originating from a rack has an ON/OFF pattern with properties that fit heavy-tailed distributions • traffic that leaves the edge switches is bursty.	• link utilisation is higher in the core, while the edge is lightly utilised • a maximum of 25% of the core links are highly utilized (hotspots) • losses are not correlated with link high utilisation, but are due to temporary bursts.	• in cloud data centres, 80% of the traffic coming from servers stays within the rack • in the case of university and private enterprise data centres 40%-90% of the traffic leaves the rack.	• time of day/week influences link utilisation especially in the core and moderate in the other levels of the DC.
[KSG+09]	-	• periodic short-term bursts • interarrival time 15ms at servers and ToR switches • the median arrival rate of all flows is 100 flows/ms	• 86% of the links have congestion periods of at least 10 seconds • 15% of the links have congestion periods of at least 100 seconds • over 90% of the congestion periods are less than 2 seconds and more than 1 second.	• jobs requiring high bandwidth are placed near each other (on the same server, in the same rack, in the same VLAN) • probability of 89% that servers within the same rack do not exchange traffic and 99.5% in the case of servers in different racks.	• significant variability in the traffic matrix, both in magnitude and in the pairs of servers which exchange data • no evidence of incast
[GJK+09]	-	• a machine has 50% of the time about 10 concurrent flows	-	-	• lack of traffic predictability, no stable traffic matrix
[AGM+10]	• median number of concurrent flows per server is 36 • 99.99th percentile number of concurrent flows per server is more than 1600	• the variance in interarrival time is very high, with a very heavy tail; spikes occur; large number of outgoing flows happen periodically	-	-	• division between query traffic (latency critical) and background traffic
Continued on next page					

Table 2.3 Continued from previous page					
Study	**Concurrent flows**	**Interarrival times**	**Link utilisation**	**Communication between servers**	**Other**
[HKP+11]	-	-	• only a few ToR pairs send or receive a large volume of traffic • these ToRs exchange much of their data with few of the other ToRs • over 60% of the demand matrices have fewer than 10% of their links hot at any time • hot links are associated with a high fan-in (or fan-out) • fewer than 10% of hot links repeat	-	-
[MPZ10]	• 80% of VMs have average traffic rate (over two-week period) less than 800 KBytes/min • 4% of VMs have average traffic rate (over two-week period) 8000KBytes/min	-	-	-	• overall stable per-VM traffic at large timescales ($>$ 15 min) for more than 82% of the VMs • weak correlation between traffic rate and latency
[RZB+15]	• Web servers and cache servers have 100s to 1000s of concurrent connections; • Hadoop nodes have 25 concurrent connections on average;	• median flow interarrival times are 2ms for Hadoop; • median flow interarrival times are 3ms and 8ms for cache leaders and followers;	• between hosts and the ToR is quite low; • the median utilisation between the ToR and aggregation switches is between 10-20% across clusters; • between the aggregation and core switches the utilisation is even higher; • load varies significantly between clusters;	• cache followers and leaders communicate with 175-350 different racks concurrently; • Web servers communicate with 10-125 racks concurrently; • most of the traffic is destined to only a few 10s of racks;	• load balancing is effective; • traffic demands are stable over intervals as long as 10 seconds; • heavy hitters' sizes are not much larger than the median flow sizes, and they change rapidly; • packet sizes are small, median length for non-Hadoop traffic $<$ 200 bytes.
End of table					

2.3.4 Data centre network monitoring systems

In this section, I review the most important tools to measure network latency and packet loss in data centres. Measuring network conditions within a data centre is notoriously difficult, since the tools used need to satisfy several properties: be lightweight, always-on, not load the network, so as to not degrade users' application performance, offer information that can be quickly acted upon, and be easy to use and configure by network operators or users. Such a custom

data centre network monitoring tool can take advantage of the data centre's known topology, and hardware and software configuration. Furthermore, the measurement techniques must be complemented by a highly scalable storage and analysis system that can alert operators about issues within the network, such as high latency, packet loss, but also to provide historical data to understand trends. The systems designed for data centres are based on active measurement, and can be complemented by passive techniques, such as exploiting the timestamps carried in the TCP headers when these are enabled [Str13]. Passive measurement of TCP RTT [Str13] is comparable in accuracy with ICMP measurement. If losses occur, the application will experience higher latencies due to TCP's in-order delivery semantics. The segments received that appear after the lost segment must wait for the lost segment to be retransmitted and received before they can be delivered to the application. Thus, the application level RTT is greater than the one measured at the TCP layer. Another way to monitor network latency in a data centre, though costly, is to have each host equipped with a common clock, such as a GPS receiver, and run one-way delay measurements between hosts.

Table 2.4 compares the properties of systems used to measure network latency and packet loss in data centres, including PTPmesh, which is presented in Chapter 4. The comparison looks at aspects related to type of measurements taken, their frequency and coverage, availability, implementation, deployment, and data storage and analysis of collected measurements. A pair is defined by two hosts: one that sends a probe, and another that receives the probe and sends an answer. *Ping* and *traceroute* are the traditional tools to perform such measurements, however, these lack the precision, the flexibility and the scale of custom purpose built tools for data centre monitoring. Cisco IP SLA [Cisb] monitors network performance by sending probe packets. It runs on Cisco switches and it can collect data about one-way latency, jitter, packet loss and other metrics. The measurements can be accessed through SNMP or command-line interface, being stored in the switches.

Large-scale monitoring systems, such as NetNORAD [ALZ16], Everflow [ZKC+15], Pingmesh [GYX+15], or VNET Pingmesh [RBB+18], have originated from companies and cloud providers. NetNORAD [ALZ16] is a system used in Facebook's data centres to measure RTT and packet loss ratio by making servers ping each other, for different Quality-of-Service (QoS) classes of traffic. The system runs measurements at data centre, region and global level. Everflow [ZKC+15] is a system that monitors all control packets and special TCP packets for all flows (TCP SYN, FIN, RST), and supports guided probing by injecting crafted packets. Their behaviour is monitored through the network, and can be used to measure link RTTs. Pingmesh [GYX+15] is an always-on tool that runs RTT measurements between every two servers in data centres. The system measures inter-server latencies at three levels, Top-of-Rack switch, intra data centre and inter data centre. Pingmesh also reports the packet drop rate, which is inferred based on the TCP connection setup time. An extension to Pingmesh is VNET Pingmesh [RBB+18], which monitors latency for tenant virtual networks (VNETs), whereas Pingmesh performed bare-metal host monitoring. The TCP probes are sent from the virtual switch at the end-host. Unlike Pingmesh which measured latency from userspace, VNET

Pingmesh measures the latency from the kernel, but the measured values can be negatively affected by increased CPU utilisation, disk I/O operations and caching effects. SLAM [YLS+15] is a latency monitoring framework for SDN-enabled data centres, which sends probe packets in order to trigger control messages from the first and last switches of a network path. SLAM uses the arrival times at the controller of the control messages to compute a latency distribution for that network path and is able to detect increases in latency of tens of milliseconds on a path.

In a large-scale measurement system, probing is normally done between chosen pairs of servers at defined time intervals. Since a data centre has tens of thousands of hosts, a server does not ping every other host, but instead a subset of servers is selected to ensure the best coverage, while minimising the number of redundant probes and reducing the network traffic incurred. Another challenge associated with probing is the multi-path nature of the data centres coupled with the use of ECMP, making it hard to know which network path the probes are taking, unless tracing the trajectories of packets through embedded identifiers is used [PM15; TAL15]. In Pingmesh [GYX+15], all of the servers under a ToR switch form a complete graph for pinging each other, and similarly all of the ToR switches form a complete graph through designated servers from all racks, and all of the data centres form a complete graph using the same procedure. Unlike Pingmesh, VNET Pingmesh [RBB+18] covers only the network paths between tenant VMs. NetNORAD [ALZ16] deploys a small number of pingers in each cluster and responders on all of the machines. All of the pingers use the same global target list, which contains at least two machines from every rack. deTector [PYW+17] uses an algorithm to minimise the number of probes sent for detecting and localising packet losses and latency spikes.

Programmable switches [BGK+13] enable more sophisticated operations for network monitoring, allowing the measurement of latency and packet loss directly in-network. Examples are Inband Network Telemetry (INT) [HW16], LossRadar [LMK+16b], Marple [NSN+17]. The disadvantage of these frameworks is that programmable switches must be deployed in the network, with legacy networks not being able to run these frameworks. INT [HW16] measures the end-to-end latency between virtual switches. Each network element on the path appends their per-hop latency to a packet that flows between the two virtual switches located at the ends of the path. The end-to-end latency is computed by adding the per-hop latencies, and it assumes that switching and queueing delays dominate, while the propagation delays are negligible. LossRadar [LMK+16b] is a system that can detect packet losses in data centres within 10s of milliseconds, reporting their switch locations and the 5-tuple flow identifiers. It keeps specific data structures at switches, which are periodically exported to a remote collector and analyser. It does not perform latency measurements. Marple [NSN+17] uses programmable key-value stores on switches to compute different metrics, such as a moving exponentially weighted moving average (EWMA) over packet latencies per flow, packet loss rate per connection, or to capture packets experiencing high end-to-end queueing latency.

A common goal of most of these large scale measurement systems is fault localisation. For example, NetNORAD is used in conjuction with *fbtracert* [ALZ16], which traces multiple paths between two endpoints in the network in parallel to determine the location of the fault.

	Measurement	Probe Type	Probe Frequency	Availability	Coverage	Deployment	Data Storage and Analysis
Ping	RTT; packet loss ratio	ICMP	-	single measurement	targeted pair	Hypervisor or VM	locally; analyse independently
Traceroute	RTT	ICMP ECHO/ TCP SYN	-	single measurement	targeted pair	Hypervisor or VM	locally; analyse independently
Cisco IP SLA [Cisb]	RTT; one-way delay (requires synchronised clocks); packet loss	ICMP/ UDP/ TCP/ HTTP/ DNS	between 1 and 604800 seconds	always-on	targeted path	CISCO switches	locally; analyse independently
Pingmesh [GYX+15]	RTT; packet loss ratio	TCP/HTTP	minimum 10s seconds	always-on	inter-servers in a rack, inter-ToRs, inter-data centre	Hypervisor	Cosmos and SCOPE [CJL+08]
NetNORAD [ALZ16]	RTT; packet loss ratio	UDP	configurable	always-on	all pairs	Hypervisor or VM	Scribe and Scuba [ALZ16]
Everflow [ZKC+15]	link RTT	packet marked with debug bit	-	single measurement	targeted path	switches and controller	custom analyser and SCOPE [CJL+08]
SLAM [YLS+15]	network path latency distribution	crafted probe	-	single measurement	targeted path	OpenFlow switches	controller
INT [HW16]	end-to-end latency	crafted probe	-	single measurement	targeted path	programmable switches	last switch on path; analyse independently
LossRadar [LMK+16b]	packet losses at switches	no probes	10 ms	always-on	cover all paths	programmable switches	custom collector and analyzer
deTector [PYW+17]	packet drop	UDP	10 packets/sec	always-on	selected paths	end-host, central controller	custom analyser
007 [ACC+18]	packet drop	TCP and traceroute	per flow	always-on	all paths	end-host	custom analysis agent at end-host
PathDump [TAL16]	packet drop	no probes	per packet	always-on	all paths	end-host and switches	custom server stack and controller
Marple [NSN+17]	packet drop	no probes	per flow	always-on	all paths	end-host, programmable switches	programmable key-value store
VNET Pingmesh [RBB+18]	RTT; packet loss ratio	TCP	minimum 10s seconds	always-on	VNET full mesh	virtual switch at end-host	Cosmos and SCOPE [CJL+08]
PTPmesh [PM17] (chapter 4)	one-way delay (estimate); packet loss ratio	UDP	up to 128 probes per second	always-on	multiple pairs	Hypervisor or VM	locally; analyse independently

Table 2.4: Comparison between systems used to measure network latency and packet loss in data centres.

Pingmesh is used to detect switch silent packet drops. NetPoirot [ACL+16] presents a classification algorithm that identifies the root cause of failures using TCP statistics collected at one of the endpoints. The work in [RZB+17] looks from the end-host to identify the faulty links and switches, by correlating anomalies in end-host statistics with the network path of the packets. 007 [ACC+18] tracks the path of TCP connections that display retransmissions through traceroute, and identifies the links with the most retransmissions as the faulty ones. PathDump [TAL16] traces packets through data centre networks, and can report poor TCP performance when the number of consecutive packet retransmissions is above a certain threshold for a flow. [ZLZ+17] presents a sampling framework which can poll a subset of switch counters at microsecond-level granularity to determine microbursts in data centres.

To sum up, a data centre network monitoring tool should offer useful measurements, be lightweight, easy to configure and deploy, highly available, and offer sufficient coverage of the network.

2.4 Latency in data centers

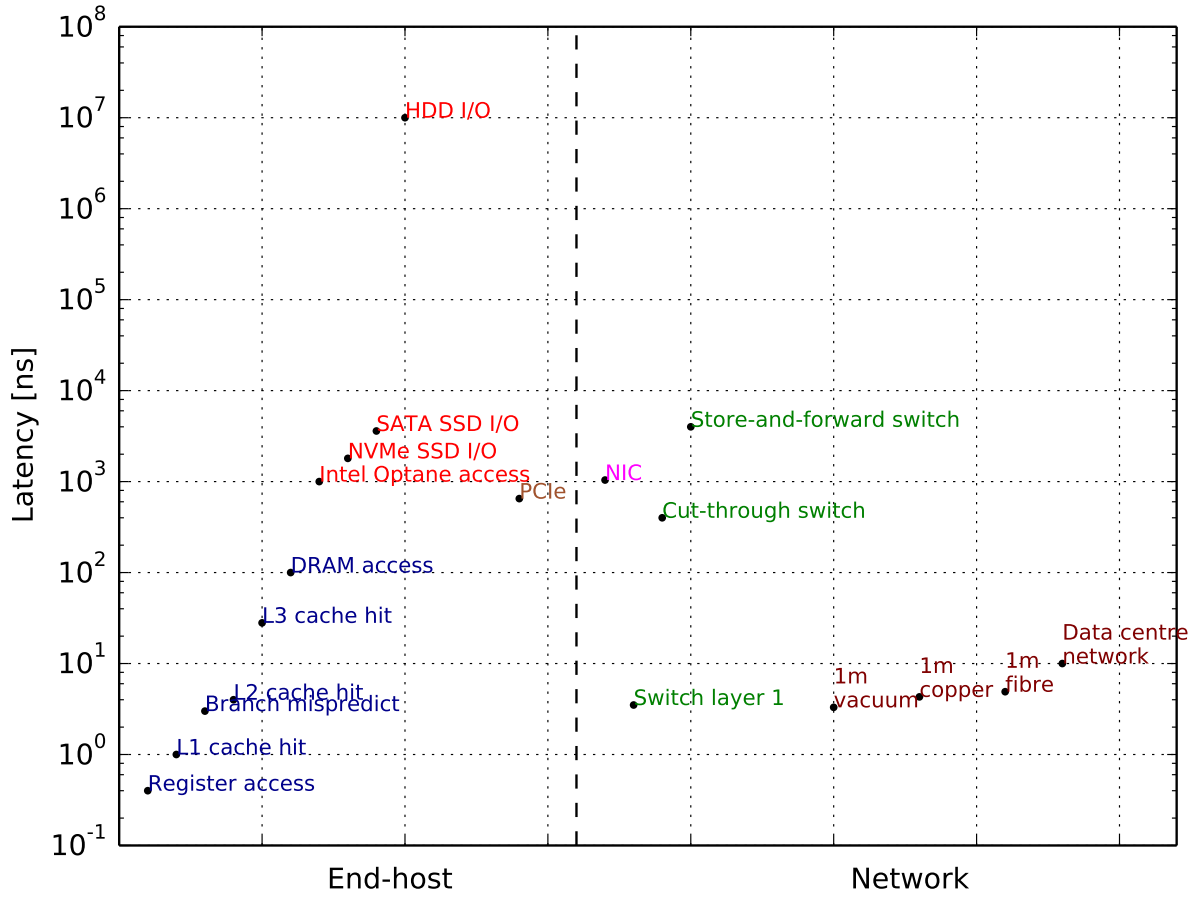


Figure 2.4: System events and their latencies.

The fact that network latency impacts performance is well known for wide-area networks (WAN) [Lid08; SCK+09; SCG+14; BAC+17], as it is implicitly a part of rate computations for

Event	Latency Range
Nanosecond events	
Register access [Lev09]	0.4ns
L1 cache hit [Lev09]	1ns
Branch mispredict [Lev09]	3ns
L2 cache hit [Lev09]	4ns
L3 cache hit [Lev09]	12ns-40ns
DRAM access [Lev09]	100ns
Switch Layer 1 [Exa18a]	2.4ns-4.6ns
Switch Layer 2 (cut-through) [Pao10; Neta]	330ns-500ns
PCIe Interconnect [NAZ+18]	400ns-900ns
1m vacuum	3.3ns
1m copper	4.3ns
1m fibre	4.9ns
Microsecond events	
NIC [Exa18b]	880ns-1.2μs
Switch Layer 2 (store-and-forward) [Netb]	<4μs
Data centre network propagation delay [MLD+15]	1μs-10μs
Intel Optane memory access [Int18e]	<10μs
NVMe SSD I/O [Int18d]	18μs-77μs
SATA SSD I/O [Int18c]	36μs-37μs
Millisecond events	
HDD I/O [AA15]	6ms-13.2ms
London-San Francisco RTT	152ms

Table 2.5: System events and their latencies.

TCP. The latency for WANs is in the order of tens of milliseconds to hundreds of milliseconds. However, a significant part of the communication today takes place within data centres, where latency values are far below the WAN scale. Likewise, host and network components within the data centre are several orders of magnitude faster than millisecond scale, being mostly in the order of hundreds of nanoseconds to tens of microseconds [BMP+17]. In Figure 2.4 and Table 2.5, I present typical latency values for common system events and network components. Storage access latencies have reduced dramatically over the years, going from traditional mechanical disks latencies of 10ms to NVMe SSD latencies [Int18d] in the order of tens of microseconds. Cut-through switch latencies have sub-microsecond transit latencies, and store-and-forward switches have transit latencies in the order of microseconds. What is interesting to note is that the latency of a switch is now at the same magnitude as the latency of traversing 100m one way within the data centre over fibre. *This means that the architecture and topology of the network within the data centre can significantly vary between cloud operators, exposing users to different data centre latency magnitudes and variances.* Data centre network fat-tree topology (§2.3) caters to general workloads. Besides the inherent differences due to the data centre network architecture, different network latencies between hosts can arise also because of link failures, network congestion, or load imbalance caused by ECMP's handling of flows of

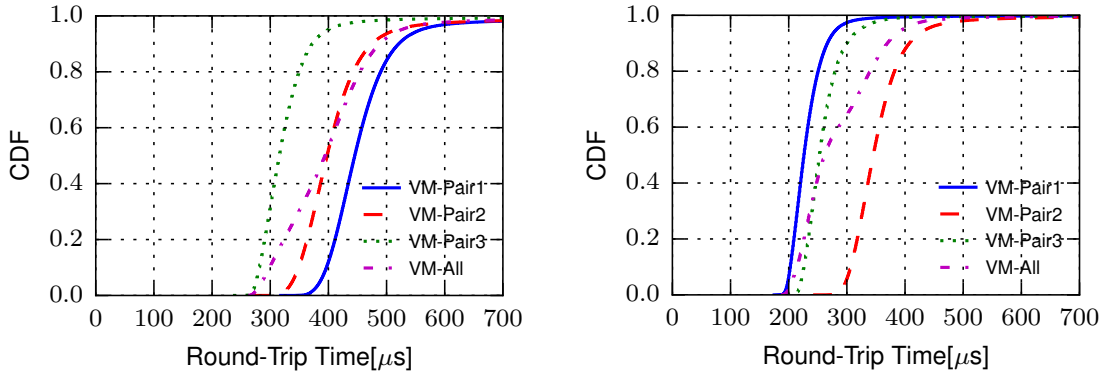
different sizes.

To get a preliminary understanding of the scale and distribution of latency as experienced by a user in a data centre, I measure the RTT between multiple VMs rented from different cloud operators. For each of three cloud operators, I choose one data centre from US and one from Europe, and I rent four VMs in each. The VM's type is the default type from each cloud operator, running Ubuntu 16.04. Since the VMs' performance may be affected by other colocated VMs and the network traffic within the data centres may be different due to diurnal and weekly patterns, I run measurements over several days, totalling 100 million RTT measurements between each VM pair. Information regarding the hop count between the rented VMs is not available, and *traceroute* does not reveal any useful information.

One of the VMs, operating as a client, measures the RTT to the three other VMs (operating as servers). The client VM sends a UDP packet to the first server VM and waits for a reply from it, measuring the time between the sending of the packet and the receipt of the reply. In one round, the client makes $100,000$ such measurements. Once a round finishes, the client VM waits 10 seconds before moving to the next server VM, and so on. The measurements are performed sequentially in a round robin fashion. Taking into account the time of each measurements round, the latency of each VM pair is measured approximately once per minute, for a thousand consecutive minutes.

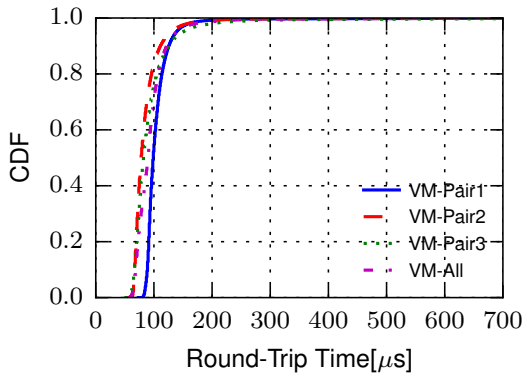
The UDP latency measurement methodology and source code are based on the tool utilised in [ZGP+17]; it is intended for accurate low latency measurement, and sends a single measurement probe at a time, rather than a train of packets. As a result, the latency measurements only observe the state of the network and do not congest it. The latency measurements use the CPU's Time Stamp Counter (TSC). TSC is a 64-bit register present on recent Intel x86 processor. It counts the number of cycles since reset, and provides a resolution of tens of nanoseconds (due to CPU pipeline effects) [ZGP+17]. Access to TSC is done using the *rdtsc* x86 assembly instruction. The RTT is measured on the client VM by computing the difference between two rdtsc reads: one just before the request is sent, and one as it is received. Using the rdtsc instruction results in an error within VMs running in Microsoft Azure, so I use the less precise *clock_gettime* function with the CLOCK_MONOTONIC parameter instead.

The RTT CDFs for Amazon EC2, Google Cloud Platform and Microsoft Azure are presented in Figure 2.5. I also present in each CDF an aggregate plot using all the RTTs measured by a single VM to the three other VMs within the same data centre. It can be observed that there are differences between cloud operators, but on the other hand, the measured latencies within data centres of the same cloud operator share the same characteristics. I ran the measurements between 10 and 13 December 2016, and then repeated them between 8 and 10 May 2017 (Figure 2.6). While changes can be observed between the two measurements campaigns, the ranges of medians are similar, with one exception only, where the median latency decreased (Amazon EC2 US data centre). These changes can be the result of any of the following factors: different VM placement, hardware or software upgrades, or different network utilisation.

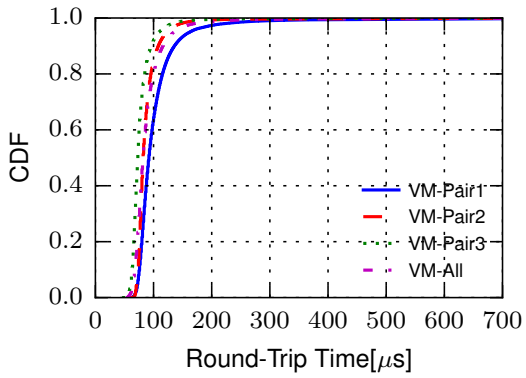


(a) Amazon EC2 US

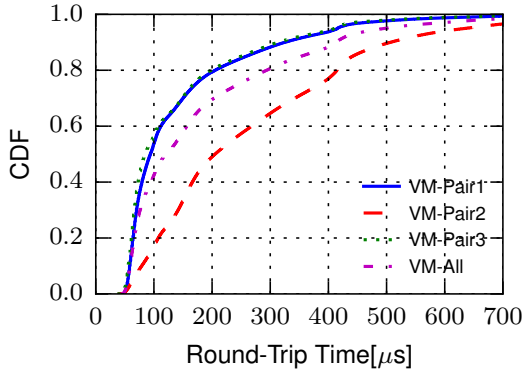
(d) Amazon EC2 Europe



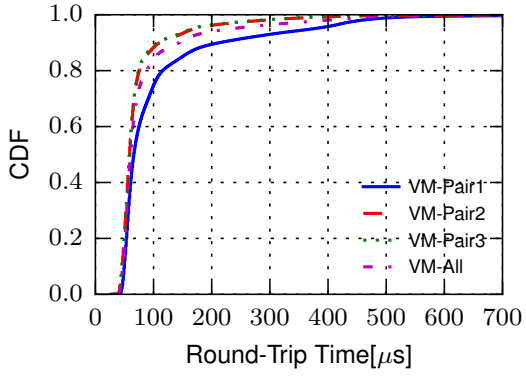
(b) Google Compute Engine US



(e) Google Compute Engine Europe

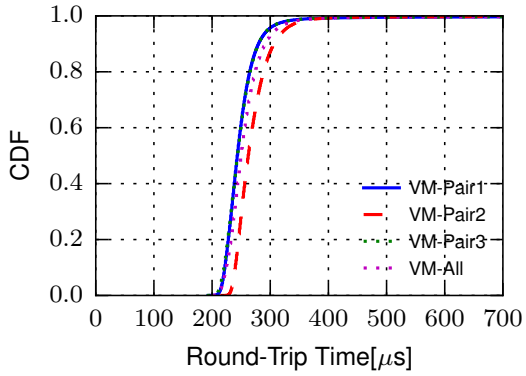


(c) Microsoft Azure US

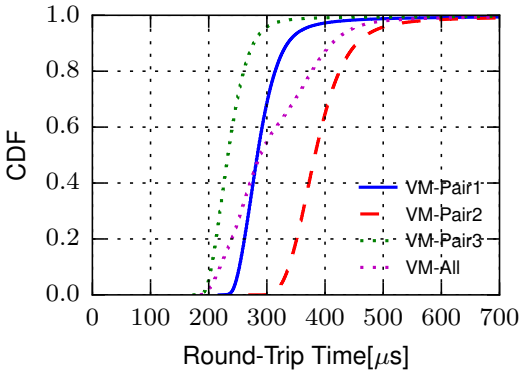


(f) Microsoft Azure Europe

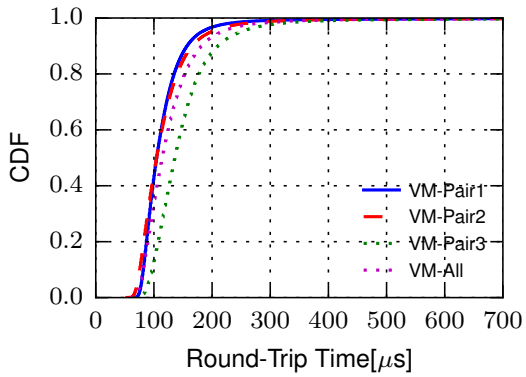
Figure 2.5: Measured RTTs within data centres for Amazon EC2, Google Compute Engine and Microsoft Azure in December 2016.



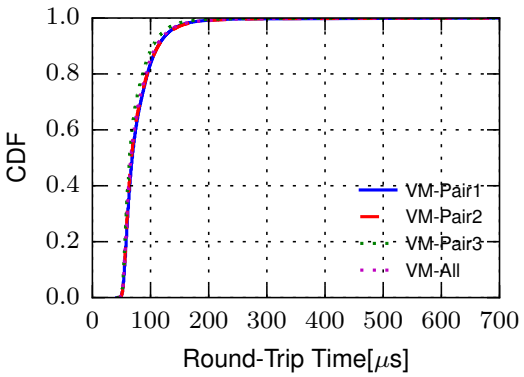
(a) Amazon EC2 US



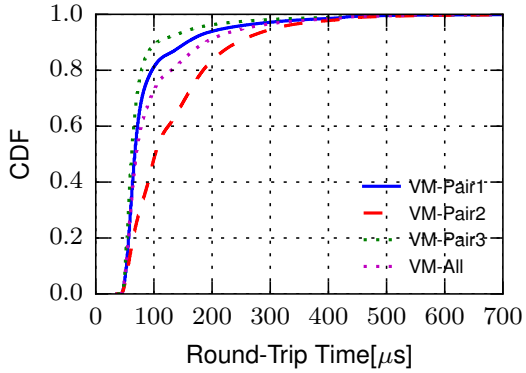
(d) Amazon EC2 Europe



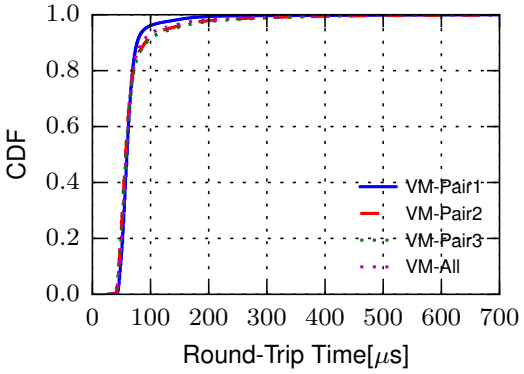
(b) Google Compute Engine US



(e) Google Compute Engine Europe



(c) Microsoft Azure US



(f) Microsoft Azure Europe

Figure 2.6: Measured RTTs within data centres for Amazon EC2, Google Compute Engine, and Microsoft Azure in May 2017.

2.5 Network latency impact on application performance

Network latency can affect a user's experience in a significant manner. For some applications, performance can decrease when subjected to increases in network latency. Figure 2.7 illustrates the impact of increases in latency on application performance for several common data centre applications.

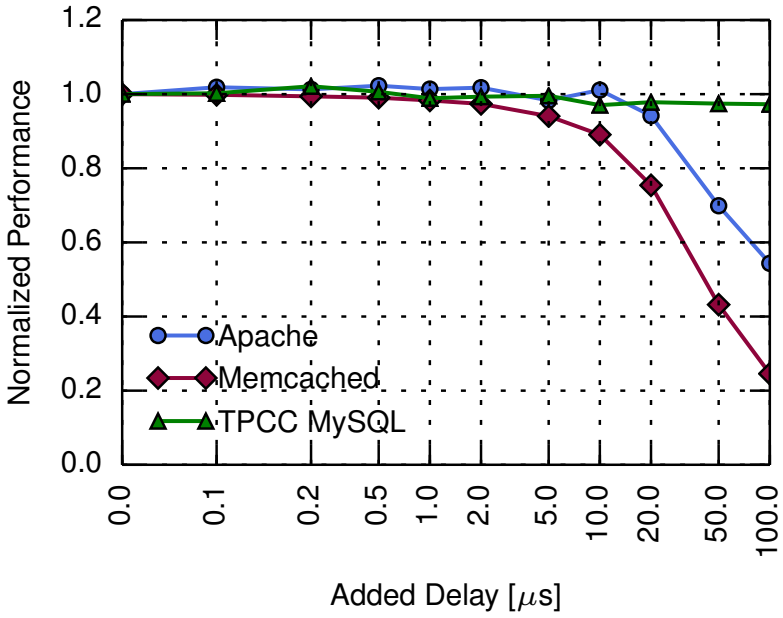


Figure 2.7: Network latency effect on application performance.

Using an experimental configuration described in Section 2.6.1, Figure 2.7 illustrates experimental results for three application benchmarks. Each benchmark reports results for an application specific performance metric. The three benchmarks I use are: Apache benchmark *ab* [Proa] reporting mean requests per second (with the Apache web server [Pro18]), Memcached benchmark *memaslap* [Prob] reporting queries per second (QPS) (with the Memcached server [Mem18], and TPC-C MySQL benchmark [Lab17] reporting New-Order transactions per minute, where New-Order is one of the database's tables (with the MySQL database [Ora18]). These results are normalised in order to compare the applications with respect to how latency-sensitive they are.

Between the two hosts of the experimental configuration described in Section 2.6.1, I insert the NRG latency-injection appliance (§2.1.4) that allows the injection of arbitrary latency into the system, as shown in Figure 2.8. The advantages of using a hardware-based approach over a software-based one were described in Section 2.1.4. The injected latency values range from 0μs to 100μs, which is the lower range of measured latency values in data centres from different cloud providers (§4.5 and Chapter 4).

Each test begins by measuring a baseline, which is the performance of each benchmark under the default setup conditions, taking into account the base latency introduced by the latency-

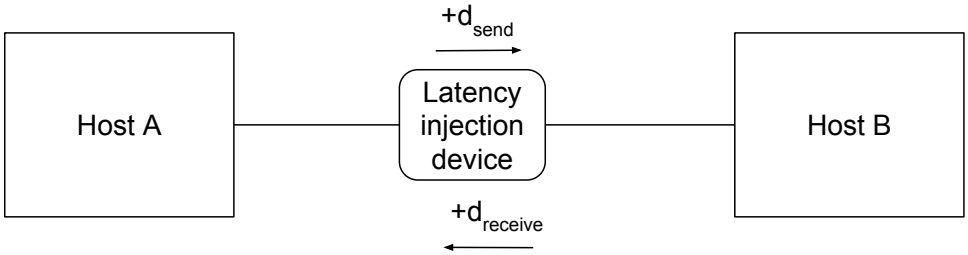


Figure 2.8: Network latency is injected between the two hosts in both directions (send and receive) by the hardware device.

injection appliance. Latency is then artificially inserted by the appliance, and the application-specific performance is measured. The impact on experiments of the artificially inserted latency can be derived by removing the baseline measurement. Figure 2.7 shows the effect of added latency for the three benchmarks. Each benchmark was run 100 times for the baseline and for each added latency value. The figure illustrates the average values, and standard errors are omitted for clarity, as their values are small. In one run, the Apache benchmark sends 100,000 requests and the Memcached benchmark sends 10 million requests. The TPC-C benchmark runs continuously for 1,000 seconds, with an additional time of 6 minutes of warm-up, resulting in 100 measurements over each 10 seconds period.

The application most sensitive to latency is Memcached: the addition of 20μs latency leads to a performance drop of 25%, while adding 100μs will reduce its throughput to 25% of the baseline. The TPC-C benchmark is the least sensitive to latency, although still exhibits some performance loss: 3% reduction in performance with an additional 100μs. Finally, the Apache benchmark observes a drop in performance that starts when 20μs are added, while adding 100μs leads to a 46% performance loss. TPC-C is not sensitive to these small latencies, since a transaction completion time is in the order of tens to hundreds of milliseconds, unlike Memcached and Apache, for which request-response request latencies are in the order of tens of microseconds in the case of Memcached, and hundreds of microseconds in the case of Apache.

While these results are obtained under optimal setup conditions, within an operational data centre worse results could be expected, as latency is further increased under congestion conditions and as applications compete for common resources. The results of Figure 2.7 clearly show that even a small increase in latency can significantly affect an application's performance.

2.6 End-host and in-network baseline latency contributions

While network latency values in WANs are in the order of milliseconds, masking end-host latency, latencies in data centres are in the order of tens to hundreds of microseconds (Section 2.4,

Chapter 4), making end-host latency values an important component of the end-to-end latency experienced by applications within the data centre. Furthermore, due to the distributed nature of data centre applications, tail latencies have dramatic effects on application performance [Bar14].

To understand where improvements can be made to reduce the end-to-end latency experienced by applications, authors in [ZGP+17], a paper of which I am also an author, perform a decompositional analysis of latency from the application level down to the wire. The analysis spans the latency between the time a request is issued by an application to the time a reply is received by the application. The work studied a system under ideal conditions, hence these results represent the baseline latency of networked systems. The different latency contributors are explored using a set of carefully designed experiments. The experiments focus on commodity hardware and Ethernet-based networking.

I restate the results presented in [ZGP+17], and I put them in the context of the work done in this dissertation. I first present the test setups (§2.6.1) on which the experiments are run. Next, I present the results of the experiments for in-host latency (§2.6.2), and for in-network latency (§2.6.3). The complete results are presented in Table 2.6. Each experiment is annotated with the corresponding entry number in Table 2.6.

2.6.1 Tests setup

The test setup uses two identical hosts running Ubuntu server 14.04LTS, kernel version 4.4.0-42-generic. The host hardware is a single 3.5GHz Intel Xeon E5-2637 v4 on a SuperMicro X10-DRG-Q motherboard. To minimise interference, all CPU power-saving, hyper-threading, and frequency scaling features are disabled throughout the tests. The host adapter evaluation is done on commodity NICs, Solarflare SFN8522, and Exablaze X10, using both standard driver or a kernel bypass mode. For determining the minimum latency, the interrupt hold-off time is set to zero. Both hosts have identical NICs in each experiment. Only Ethernet-based communication is considered. As illustrated in Figure 2.10, an Endace 9.2SX2 DAG card (7.5ns timestamping resolution) [End] and a Net Optics passive-optical tap [Opt] are used to intercept client-server traffic, allowing for independent measurement of client and server latency.

2.6.2 End-host latency contributors

Figure 2.9 presents the different elements contributing to the latency experienced within the host. The results of the experiments are presented in Table 2.6. The results are generalisable also to other platforms and other Linux kernel versions. This observation is made based on evaluation on Xeon E5-2637 v3, i7-6700K and i7-4770 based platforms, and Linux kernels ranging from 3.18.42 to 4.4.0-42.

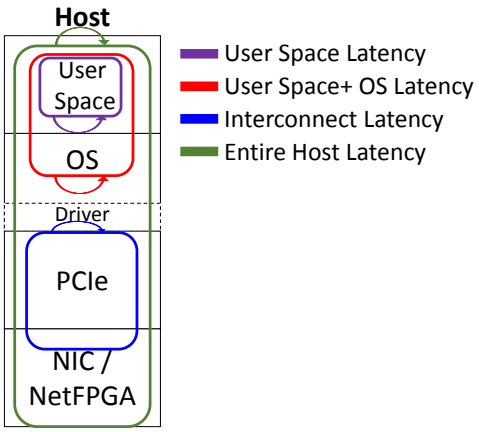


Figure 2.9: End-host tests setup [ZGP+17].

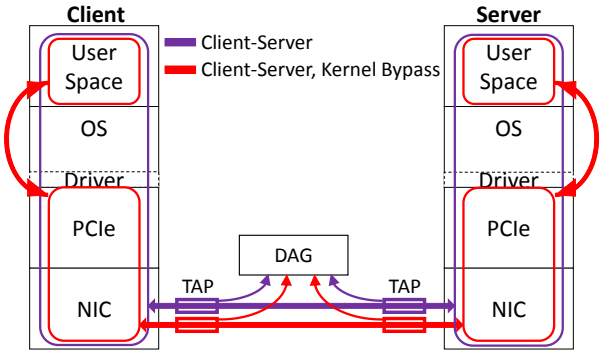


Figure 2.10: Client-server tests setup [ZGP+17].

Timestamp counter latency (1) To accurately measure latency, a baseline for the methods employed is made. The latency measurements are based on the CPU's Time Stamp Counter (TSC). TSC is a 64-bit register, present on the processor, it counts the number of cycles since reset and thus provides a resolution of approximately 288ps-per-cycle, although realistically the resolution is tens of nanoseconds due to CPU pipeline effects. Access to TSC is done using the *rdtsc* x86 assembly instruction. In order to understand hidden latency effects, and following the Intel recommendations for TSC access [Pao10], two register read operations were conducted consecutively. This simple TSC read operation is repeated a large number of times (order of 10^{10} events), and the time gap measured between every pair of consecutive reads. Results are saved into previously allocated and initialised buffers, and access to the buffers is outside the measurement code.

This experiment is conducted in three different modes: firstly, *Kernel Cold Start* (1a) which serves as an approximation of a bare metal test. Kernel Cold Start measures very early within the kernel boot process, before the scheduler, multiprocessing and multicore support have been started. The second test, *Kernel Test* (1b), runs from within the kernel, and represents an enhanced version of the recommended test described in [Pao10]. The third test, *User Space Test* (1c), provides high-accuracy timestamping measurement from within a user-space application. The application is pinned to a single CPU core and all other tasks and interrupts are moved to

other cores. This is representative of real-time application operation. In contrast with the Kernel Test, interrupts, such as scheduling preemption, are not disabled so as to represent the runtime conditions of real applications. As a result, some of the long time gaps measured for the third configuration are the result of OS scheduling.

Virtualised environment (1d) The contribution of a virtualised environment is examined by repeating the TSC tests from within a VM. The hypervisor used is VirtualBox [Ora] version 4.3.36, with an Ubuntu VM which has the same version as the base operating system. The VM was configured to run the guest OS on a single dedicated CPU core with no co-located native OS activities.

Up to the 99th percentile the latency is in the order of 10ns for the TSC measurements. Beyond this, TSC latencies can be in the order of microseconds or hundreds of microseconds, in both kernel and user space, and in the order of milliseconds for VMs. The authors state that the most prominent cause of long maximum (tail) latency events observed during the TSC experiments is not running an application in real time or pinned to a core.

User space + OS latency (2) This experiment investigates the combined latency of the (user-space) application and the operating system. The test sets up two processes and opens a datagram socket between them, measuring the RTT for a message sent from a source process to the destination process, and back. TSC is used to measure the latency and the time is measured by reading TSC before and after the message reply is received. While this does not fully exercise the network stack, it does provide useful insight into the kernel overhead.

Host interconnect (3) To evaluate the latency of the host interconnect (*e.g.*, PCI-Express), the authors use the NetFPGA SUME platform [ZAC+14], which implements x8 PCIe Gen3 interface. The DMA design is instrumented to measure the interconnect latency. As the network hardware and the processor use different clock sources, the one-way latency can not be directly measured. Instead, the round trip latency of a read operation (a non-posted operation [SA99] that incorporates an explicit reply) is measured. Every read transaction from the NetFPGA to the CPU is timestamped at 6.25ns accuracy within the DMA engine when each request is issued and when its reply returns. The cache is warmed up before the test, to avoid additional latency due to cache misses, and the memory address is fixed. The measured latency does not include the driver latency, as neither the driver nor the CPU core participate in the PCIe read transaction.

The resuls show that the PCIe is low latency, with the minimum value of 552 ns for the smallest packet size and 976 ns for the largest packet size. The maximum latency values are 50 ns larger than the minimum latency values, showing that the latency distribution has low variance. Additionally, it can be observed that the variance is not dependent on packet size.

Host latency (4) To measure the latency of an entire host, a bespoke request-reply test is used, which measures the latency through the NIC, PCIe interconnect, kernel and network stack, the application level, and back to the NIC. In contrast to the *User Space + OS Latency* experiment, here packets traverse the network stack only once in each direction. As illustrated in Figure 2.9, packets are injected by a second host, and using the DAG card, the host latency is isolated, measuring the latency from the packet's entrance to the NIC and until it returns from the NIC.

Kernel bypass (5) The latency contribution of the OS kernel and the impact of kernel bypass upon latency are compared. Similar tests to those used for the *Host Latency (4)* experiment are used with kernel bypass enabled and disabled. Two NICs are used in the experiments: X10 and SFN8522. Each experiment is run with both NICs of the same type. The latency values obtained in this experiment are in the order of nanoseconds compared to the ones without kernel bypass, which are in the order of microseconds. Additionally, the maximum latency value is substantially lower in this experiment compared to the one without kernel bypass.

Client-server latency (6) The experiments are extended from a single host to a pair of network hosts as shown in Figure 2.10. The two servers are directly connected to each other. Using a test method based upon that described in the *Host Latency (4)* experiment, authors add support for request-reply at both hosts. This allows them to measure the latency between the userspace application of both machines. This experiment is further extended to measure the latency of queries (both get and set) under the Memcached benchmark [Prob], indicative of realistic userspace application latency.

The results for this experiment show that a more complex application has larger tail latency values, 20 ms for Memcached compared with 200 μs for the simple UDP application.

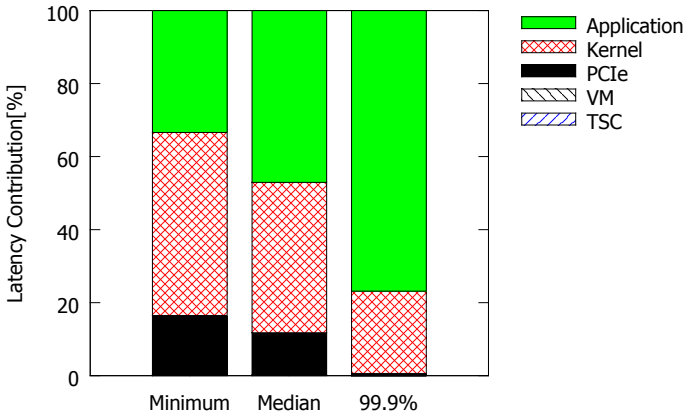


Figure 2.11: End-host latency contribution [ZGP+17].

Taking a holistic approach, Figure 2.11 shows the breakdown of latency within the host for the different percentiles. The latency of operations within user and kernel space is on the order of nanoseconds, whereas other operations take from hundreds of nanoseconds to microseconds.

Experiment	Minimum	Median	99.9th	Maximum	Observation Period
1a TSC - Kernel Cold Start	7ns	7ns	7ns	11ns	1 Hour
1b TSC - Kernel	9ns	9ns	9ns	6.9μs	1 Hour
1c TSC - From User Space	9ns	10ns	11ns	49μs	1 Hour
1d TSC - From VM User Space	12ns	12ns	13ns	64ms	1 Hour
2a User Space + OS (same core)	2μs	2μs	2μs	68μs	10M messages
2b User Space + OS (other core)	4μs	5μs	5μs	31μs	10M messages
3a Interconnect (64B)	552ns	572ns	592ns	608ns	1M Transactions
3b Interconnect (1536B)	976ns	988ns	1020ns	1028ns	1M Transactions
4 Host	3.9μs	4.5μs	21μs	45μs	1M Packets
5 Kernel Bypass	895ns	946ns	1096ns	5.4μs	1M Packets
6a Client-Server (UDP)	7μs	9μs	107μs	203μs	1M Packets
6b Client-Server (Memcached)	10μs	13μs	240μs	20.3ms	1M Queries
7a NIC - X10 (64B)	804ns	834ns	834ns	10μs	100K Packets
7b NIC - SFN8522 (64B)	960ns	985ns	1047ns	3.3μs	100K Packets
8a Switch - ExaLINK50 (64B)	0^{α}	2.7ns $^{\alpha}$	17.7ns $^{\alpha}$	17.7ns $^{\alpha}$	1000 Packets
8b Switch - ExaLINK50 (1514B)	0^{α}	2.7ns $^{\alpha}$	17.7ns $^{\alpha}$	17.7ns $^{\alpha}$	1000 Packets
8c Switch - 7124FX (64B)	512ns	534ns	550ns	557ns	1000 Packets
8d Switch - 7124FX (1514B)	512ns	535ns	557ns	557ns	1000 Packets

Table 2.6: Summary of Latency Results. Entries marked $^{\alpha}$ return results that are within DAG measurement error-range.

This means that for an application running on the host, for the common case, approximately half of the time is spent in the application, and approximately 40% is spent in the kernel and network stack. At the tail the application contributes nearly 80% of the latency. The takeaway is that there is no single component that contributes overwhelmingly to end-host latency: while the kernel (including the network stack) has an important contribution, the application level also has a significant contribution to latency as applications incur overheads due to user space/kernel space context switches.

2.6.3 In-network latency contributors

Next, the three components that contribute to network latency, namely networking devices within the network (switches, routers), cabling (*e.g.*, fibre, copper), and networking devices at the edge, are measured. The network device at the edge is represented by the NIC. The networking devices within the network considered are electronic packet switches (EPS), as they are the most common used networking devices within data centres. Networking devices such as routers will inherently have a latency that is the same or larger than a switch, but are not covered by these measurements.

Cabling The propagation delay over a fibre is 4.9ns per meter, and the delay over a copper cable varies between 4.3ns and 4.4ns per meter, depending on the cable's thickness and material

used. These numbers were derived by sending packet trains over varying lengths of cable and measuring using DAG the latency between transmit and receive. The authors note that the resolution of the DAG of 7.5ns puts short fibre measurements within this margin of error.

NIC latency (7) At least three components contribute to the measured NIC latency: the NIC's hardware, the Host Bus Adapter (a PCI-Express interconnect in this case) and the NIC's software device driver. There are two ways to measure the latency of a NIC: the first is injecting packets from outside the host to the NIC, looping the packets at the driver and capturing them at the NIC's output port. The second is injecting packets from the driver to the NIC, using a (physical or logical) loopback at the NIC's ports and capturing the returning packet at the driver. Neither of these ways allows to separate the hardware latency contribution from the rest of its latency components or to measure one way latency.

The authors chose the second method, injecting packets from the driver to the NIC. A loopback test provided by Exablaze with the X10 NIC is used. The test writes a packet to the driver's buffer, and then measures the latency between when the packet starts to be written to PCIe and when the packet returns. This test does not involve the kernel. A similar open-source test provided by Solarflare as part of Onload (*eflatency*) [Sola], which measures RTT between two nodes, is used to evaluate SFN8522 NIC. The propagation delay on the fibre is measured and subtracted from the NIC latency results.

Switch latency (8) The authors measure switch latency using a single DAG card to timestamp the entry and departure time of a packet from the switch under test. The switch under test is statically configured to send packets from one input port to another output port. There is no other traffic going through the switch. They vary the size of the packets sent from 64B to 1514B. The tests evaluate two switches, both of them cut-through switches: an Arista DCS-7124FX layer 2 switch, and an ExaLINK50 layer 1 switch (this switch allows dynamic reconfiguration of the network topology). The latency reported is one way, end of packet to end of packet.

Latest generation cut-through switching devices, such as Mellanox Spectrum and Broadcom Tomahawk, have lower latency than what the authors measure, in the order of 330ns, as described by industry analysis [Ent16].

The contribution of different latency components within the network depends greatly on the network topology. The authors explore four typical networking topologies, depicted in Figure 2.12, and use the median latency results listed in Table 2.6 for different network elements (NICs, fibres, switches). For the store-and-forward spine switch, they assume a latency comparable to that of the Arista-7500R switch [Netb], under 4μs latency for 64 bytes packet. It should be noted that this analysis to determine the in-network latency does not seek to evaluate aspects such as queueing and buffering, or congestion.

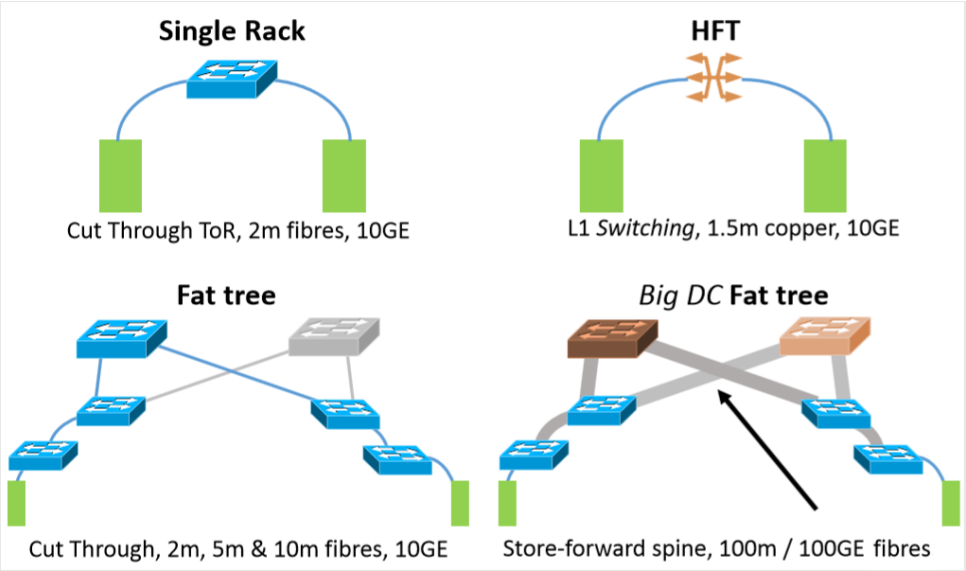


Figure 2.12: Different network topologies [ZGP+17].

The first topology is a single rack, where the hosts are connected using a cut through Top-of-Rack switch with 2 m fibres, and the link speed is 10Gb/s. The second topology is common in a high-frequency trading setting. The hosts are connected by a layer 1 switch, and the link speed is 10Gb/s. The third and fourth topologies are instances of the fat tree topology [ALV08]. In the third topology, the links are all 10Gb/s and use cut through switches across all layers of the topology. The fibres length are 2 m between the host and the Top-of-Rack switch, 5 m between the Top-of-Rack switch and the aggregation switch, and 10 m between the aggregation and core switches. In the fourth topology, store-and-forward spine switches are used in the core, with 100Gbit/s link speed and 100 m long fibres.

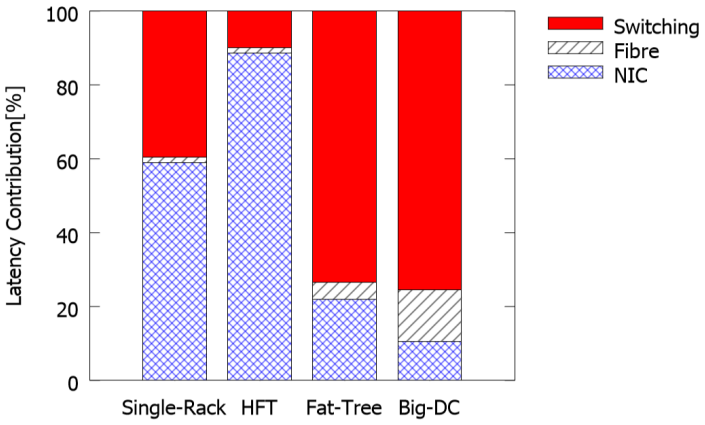


Figure 2.13: Network latency contributions [ZGP+17].

Figure 2.13 shows the relative latency contribution within each network topology. The latency contributon of the NIC and switching differs significantly for each use case, from the switching being 1/10th of the total HFT latency case, to switching taking up almost 70% of the time in the big data centre scenario. Furthermore, the significant impact of the NIC on the overall latency

should be noted, which means that there still is room for improvement for NIC latency. The latency of the fibres has a magnitude of microseconds in big data centres, being a significant contributor to the overall latency.

The experiments (§2.6.1) were designed to isolate the contribution of each component in the end-host and in network. The authors examined in their work only best-case scenarios. However, operational effects on the end-to-end latency (*e.g.*, network congestion, queueing) can make the overall latency worse. The results of the experiments (§2.6.2, §2.6.3) highlight how challenging it is to reduce end-to-end latency in data centres, as there are multiple contributors to the overall latency. Several conclusions can be drawn from these results. At the end-host, kernel bypass can substantially reduce the overall latency and latency variance, and this approach has been deployed by different cloud providers to improve end-to-end latency [FPM+18; DSA+18]. In large data centres [SOA+15], the impact of cable length is significant, the aggregated latency being in the order of microseconds. As a result, the network topology and the placement of jobs taking into account the end-to-end latency become important factors that can lead to better application performance. These two directions ought to be explored in order to reduce the network latency contribution to the overall latency. The first direction is currently explored through rack-scale computing [CBR+15]. Rack-scale computing implies resource dissagregation [GNK+16; Int18b], where resources (compute, storage, memory) are grouped in pools, no longer being confined to a server, and they communicate over a fast network. I explore the second direction in Chapter 6 by proposing latency-driven, performance-aware cluster scheduling, and I describe the NoMora cluster scheduling architecture, which aims to minimise the impact of network latency on application performance.

2.7 Cluster scheduling

I start by defining common terms used in the cluster scheduling literature. An application is called a *job*. A job may have multiple *tasks*. A task is an application instance of the job represented by one or multiple processes that run inside a container or virtual machine, usually on a single core. The tasks of a job must be placed on the available machines. A *cluster scheduler* decides on which machines to place the tasks of the jobs. Cluster scheduling in its simplest form is bin-packing of tasks on the available machines. However, this simple allocation mechanism might not yield the optimal performance for applications due to lack of adequate resources, or possible interference between tasks that share the underlying host hardware and network. These issues can be solved by respecting the job's resource demands or by defining placement constraints. Resource demands usually comprise the number of cores, the amount of memory, disk throughput, or network bandwidth that a task needs. Placement constraints can be defined to avoid colocation between tasks that might interfere, or to allocate a task to a machine with certain characteristics. Solving these issues has given rise to a large body of work on how

to best map job demands and constraints to job allocation systems.

In the following sections, I first review the main characteristics of publicly available cluster workloads (§2.7.1), and emphasizing their shortcomings. Next, I give an overview of the most important cluster schedulers developed by industry and researchers (§2.7.2), highlighting the cluster scheduling mechanisms that consider network resources in their placement decisions. Finally, I put in context my cluster scheduling policy that seeks to improve application performance taking into consideration an application's network latency demands.

2.7.1 Cluster workloads

In this subsection I discuss the characteristics of the main cluster workloads released by companies in the past years [RTG+12; CBM+17; APG+18].

Google workload The most well-known cluster trace is from Google [RTG+12]. It is a 2011 cluster trace from a 12,500 machines cluster. The Google workload is a 29-day trace of jobs that run on bare-metal hosts. The trace does not represent a cloud workload. Task runtimes are not uniform, with 80% of tasks running for less than 12 minutes [APG+18]. A similar observation can be done with regards to task resource requests, where 90% of the smallest jobs request 16 CPU cores or fewer [APG+18]. The trace has sub-second job interarrival times.

Microsoft Azure workload The Microsoft Azure VM workload [CBM+17] is the first of its kind publicly released. It spans three months, and it includes first-party workloads (internal VMs and first-party services offered to third-party customers), and third-party workloads (VMs created by external customers). More than 90% of VMs run for less than a day, and a small percentage of long-running VMs use up more than 95% of the total core hours. In terms of VM core count, almost 80% of VMs have a maximum of two cores, with almost 60% of VMs using only one core. In terms of memory, 70% of VMs use less than 4 GBytes. In terms of deployment sizes, around 40% have a single VM and 80% have at most 5 VMs. Regarding VM workloads, 68% of core hours are categorised as delay-insensitive (batch workloads, internal workloads), and around 28% are interactive, while the remaining 4% are not categorised. The VM arrival times are bursty and diurnal, and there are less VMs running during the weekend.

Both Google and Azure workloads have a high number of tasks placed (more than 140K tasks) at the beginning of the trace (timestamp 0) to setup the cluster state. Afterwards, the average number of task arrivals per hour is between 1800-3500 for the Azure trace (Figure 2.14), and 40K-70K for the Google trace [APG+18].

Two Sigma and Los Alamos National Laboratory workloads These traces share some of the characteristics of the Google workload. They have sub-second job interarrival times, requiring sub-second scheduling decisions. On the other hand, they display diurnal patterns in job submissions, similar to the Microsoft Azure workload, but not present in the Google workload. While most of the jobs have short durations and request a small number of cores in the

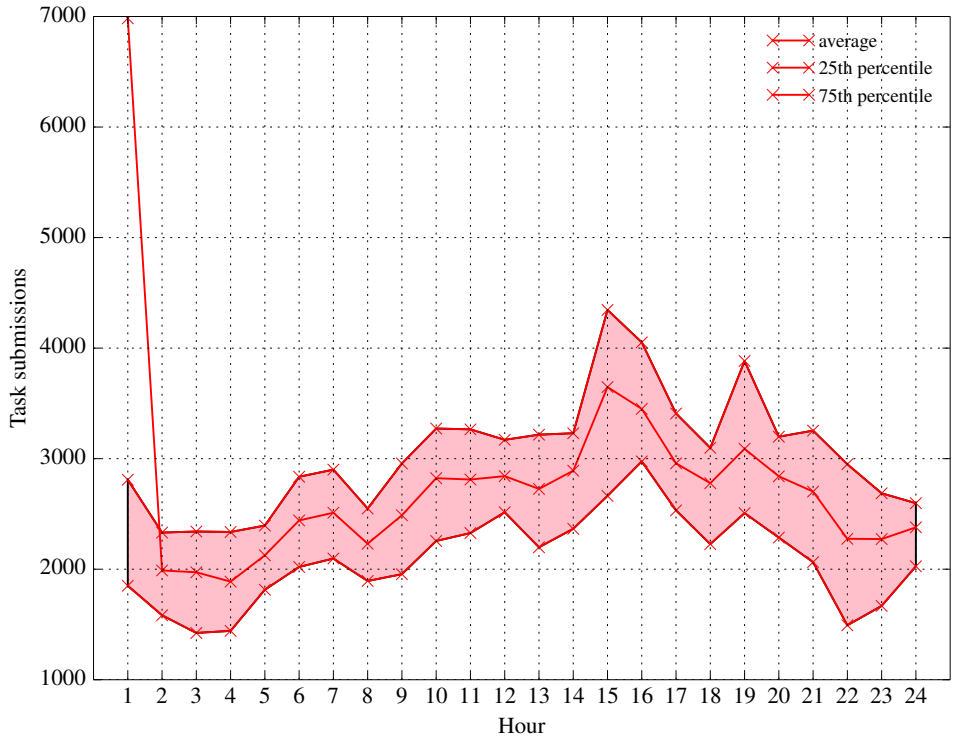


Figure 2.14: Azure workload number of task arrivals per hour - average, 25thand 75thpercentiles.

Google workload, this is not true for the Two Sigma and LANL traces. The median Google job is $4-5\times$ times shorter, and requests $3-406\times$ fewer CPU cores. Similarly, most of the VM deployments in the Microsoft Azure workload are small, with 80% of VMs having a maximum of two cores. Regarding job duration, 80% of Google jobs are less than 12 minutes, whereas in the other traces the same fraction of jobs are several hours long (2-6 hours). The Google workload has a long tail, with some jobs running for at least the duration of the whole trace.

A commonality of the cluster workloads released is that they do not include information related to networking demands, *e.g.*, network bandwidth, or how latency-sensitive the application is. The closest that is available is the Microsoft Azure workload, which mentions which VM workload is interactive. Still, this information is insufficient, since it does not offer a quantifiable metric, *e.g.*, what is the job's SLO. Therefore, this lack of information represents a challenge when developing new cluster scheduling policies that try to improve application performance while considering networking demands. To overcome this challenge, *I augment the cluster traces with application performance predictions dependent upon network latency* determined experimentally in Chapter 5. This represents a first step towards a comprehensive cluster workload which includes not only the usual information (CPU, RAM, etc.), but also information related to network resources required, thus offering a full picture of the applications deployed.

2.7.2 Cluster schedulers

Most of the cluster schedulers take into account any other constraints but those related to networking requirements or the state of the network. These constraints generally represent the needs of a job with respect to the number of cores, CPU utilisation, memory, affinity with other tasks, data locality, placement constrains, and low placement latency. I call these cluster schedulers *conventional*. Gog [Gog17] and Schwarzkopf [Sch16] give a comprehensive overview of these conventional cluster schedulers. In Section 2.7.2.1, I discuss the features of the most important conventional cluster schedulers.

Few cluster schedulers consider the networking demands of the applications and the network conditions in the data centre. Managing the network traffic to achieve short flow completion times is left to data centre transport designs [AGM+10; AYS+13; GNK+15; GSG+15; HRA+17] and flow schedulers [POB+14], this happening after the applications have been scheduled to run in the data centre. In Section 2.7.2.2, I discuss the main developments for cluster schedulers to guarantee applications network bandwidth and tail latency.

I conclude the section by discussing *application performance-aware cluster schedulers*. Instead of meeting the job's resource requirements, I consider meeting the job's performance requirements. However, given the current network conditions, the optimal performance may not be achievable. In this case, if the tenant whose job has to be scheduled is content with their job running with less than optimal performance, then the job is admitted into the system with the best achievable performance given the limits of the current network conditions. Otherwise, admission control is performed, the job being scheduled only if it can run with optimal performance.

2.7.2.1 Conventional cluster schedulers

Gog [Gog17] and Schwarzkopf [Sch16] identify the most important features of cluster schedulers: *architecture*, *multi-dimensional resource allocation*, *resource allocation and dynamic adjustment*, *task interference*, *constraint handling*, *data locality* and *low placement latency*.

Scheduler architecture There are several types of cluster scheduler architectures.

Centralised schedulers, such as Borg [VPK+15], Bistro [GSW15], Quincy [IPC+09], Firmament [GSG+16], TetriSched [TZP+16], have the advantage of having access to all of the information related to the cluster's state (*e.g.*, where the tasks are running, which tasks should be scheduled, and which machines have free cores). Consequently, these schedulers compute high-quality task placements. On the other hand, task placement latency can be significant because the computation delay grows in proportion to the cluster size [IPC+09]. Significant task placement latency can be detrimental for short-running tasks that might end up waiting to be scheduled for a time greater than their actual duration.

Distributed schedulers (Apollo [BEL+14], Sparrow [OWZ+13]) can solve this issue, but since they do not have the full view of the cluster state, they trade off high-quality task placements for lower task placement latency.

Hybrid schedulers' architecture is comprised of a centralised scheduler and one or more distributed schedulers. Examples of hybrid schedulers are Hawk [DDK+15], Eagle [DDD+16], Mercury [KRC+15]. The long-running tasks are usually placed by the centralised scheduler, while the short tasks are placed by the distributed schedulers.

Another architecture is that of *two-level schedulers* (Mesos [HKZ+11], YARN [VMD+13]), which have an application-level scheduler that maps the tasks to the resource allocations determined by a resource scheduler. For example, the MapReduce jobs are scheduled by YARN's job manager, while resources are allocated by YARN's resource manager.

Multi-dimensional resource allocation Nowadays complex data centre applications have different resource requirements (§2.3.2). Some cluster schedulers [IPC+09; FBK+12; OWZ+13; DDK+15; KRC+15; DDD+16] allocate the same amount of resources to all tasks regardless of their needs, and this may hurt application performance. Other schedulers [HKZ+11; VMD+13; SKA+13; DK14; GSW15; BEL+14; GAK+14; TZP+16; VPK+15] support specifying job resource requirements.

Resource allocation and dynamic adjustment In most cases, users can specify their jobs' resource requirements [VPK+15], but often overestimate their requirements [Gog17], or they do not know how much to request [MK15]. This can be solved in several ways: i) profiling the job before the actual execution [DK14], ii) decreasing or redistributing the initial resource allocation after an observation period [VPK+15; GCA+16], or iii) analysing historical traces of jobs to build a performance model of the expected performance depending on the job resource allocation [FBK+12; JCM+16; VYF+16; ALC+17; PBC+18]. Some performance models were developed for specific types of jobs (MapReduce and Dryad in Jockey [FBK+12], deep learning jobs in Optimus [PBC+18]). Also, techniques such as Bayesian optimisation [ALC+17] or non-negative least square [VYF+16; PBC+18] have been used to build performance models for different applications.

To draw a parallel with the work in my dissertation, in Chapter 5, I determine experimentally the relationship between network latency and application performance for typical cloud applications. This is similar to the profiling phase of applications and building a performance model done by different cluster schedulers, but these frameworks did not consider network latency demands in their profiling phase, nor in their performance model.

Task interference To achieve high cluster utilisation, multiple tasks are colocated on each machine. But tasks compete for the same resources (memory, cache, disk, network), and they may then interfere with each other, causing a decrease in application performance. As a consequence, some schedulers (Paragon [DK13], Quasar [DK14]) first determine whether the tasks waiting to be scheduled interfere with each other, and only then place the tasks based on the result.

Constraint handling Tasks can have different placement constraints. For example, an application's performance would benefit from running on a certain type of hardware, *e.g.*, machine learning tasks require specialist hardware such as General Purpose Graphics Processing Units (GPGPUs) or Tensor Processing Units [JYP+17] (TPUs). Constraints can be *hard* (mandatory), *soft* (not mandatory) or *complex* (combination of soft and hard constraints).

Hard constraints must be satisfied, the tasks not being scheduled until machines that satisfy these constraints are available. Hard constraints usually refer to hardware architecture and kernel version [SCH+11]. In the Google cluster workload, approximately 6% of all tasks have hard constraints.

Soft constraints, on the other hand, are not mandatory, and tasks can be scheduled to run even if their constraints are not met. Quincy [IPC+09] and Firmament [GSG+16] model the cluster scheduling problem as a min-cost max-flow optimisation over a flow network. The nodes in the flow network represent the tasks that are submitted and the machines of a cluster. The task and machine nodes are connected through task placement preference arcs which represent soft constraints. A min-cost max-flow algorithm run over the flow network computes task placements. In Medea [GKP+18] the constraints are soft by default, and weights can be assigned to the constraints to rate their importance. In Kubernets [HBB17], one can specify if a constraint is hard or soft.

Complex constraints are a combination of hard and soft constraints, and involve satisfying the requirements of multiple tasks or machines. They are supported by few schedulers, an example of such scheduler being TetriSched [TZP+16]. An important type of complex constraint is task affinity or anti-affinity. Task affinity refers to placing two or more tasks that have a dependency on the same resource. In contrast, task anti-affinity means placing the tasks on different resources. Kubernetes [HBB17] supports affinity/anti-affinity constraints.

Satisfying constraints generally increases task placement latency [SCH+11] and limits the scheduler's scalability.

Data locality Data locality used to be an important feature of cluster schedulers [IPC+09; JBM+15], since disk throughput is higher than network bandwidth, it is desirable for the data to be stored as close as possible to the application that uses it. However, the current data centre networks can provide one Pbps of bisection bandwidth [SOA+15] and the recent trend in resource disaggregation in data centres [GNK+16; SHC+18] make this feature less important, since the data is now accesible over a very fast network. Nevertheless, generating less network traffic by reading from local disks in a traditional server architecture lowers network utilisation, and thus reduces network congestion, providing more predictable application performance [GSG+15].

Low placement latency The time it takes to compute a placement for a task is an important feature for a cluster scheduler. A low placement latency favours high cluster utilisation, and reduces the waiting time of the tasks before being scheduled. Cluster schedulers that support complex constraints may be slow in placing tasks, but they compute high-quality task placements. Quincy's algorithm [IPC+09], for example, can take minutes for the Google workload

on a cluster with 12,500 machines [GSG+16]. This runtime is too large for workloads that have sub-second job interarrival times (§2.7.1), and it can lead to an increase in task wait time, which is especially detrimental for short-running tasks. On the other hand, Firmament's algorithm runtime [GSG+16] is sub-second for the same scenario, achieving both high-quality placements and low placement latency. Other cluster schedulers use less complex algorithms which take less time, but do not offer high-quality task-placements, since they do not take into account the tasks' constraints. Such an example is Sparrow [OWZ+13], which places short-running tasks in a random manner.

2.7.2.2 Network-aware cluster schedulers

In general, incorporating network demands within the cluster scheduler has been treated as a separate problem from cluster schedulers that take into account only the host resources required by a job. Table 2.7 describes the mechanisms to allocate network bandwidth between tenants and to provide tail latency guarantees to ensure predictable performance.

Network bandwidth guarantees In the past, network throughput variability in cloud providers was an important issue, with bandwidth varying by a factor of five in some cases [BCK+11], leading to uncertain application performance and, consequently, tenant cost. The throughput variability was the result of different factors, such as network load, tenant VM placement, and oversubscribed data centre networks.

Nowadays, data centre networks often utilise full bisection bandwidth [SOA+15; CBM+17]. Also, cloud providers' commercial offerings list the expected network bandwidth for each type of VM, be it an exact value (Microsoft Azure, Google Cloud Platform) or qualitative estimate (Amazon EC2). These changes fit with the observation that network bandwidth guarantees have improved in recent years. Recent studies have shown that cloud providers like Amazon EC2 [PMB+15b] and Microsoft Azure [PMB+15a] see less throughput variability. In the case of Microsoft Azure, larger VMs and which are placed within the same affinity group or virtual network have better network throughput and observe small variability, whereas medium sized VMs experience higher variability regardless of the policy applied, with some regions offering better performance than others. On the other hand, 60% of the tenants of Microsoft Azure use the smallest VM size (1 core) and 20% the medium VM size (2 cores) [CBM+17], which means that most of the tenants do not have strict network bandwidth guarantees even in today's data centres. In the case of Amazon EC2 [PMB+15b], the network throughput is stable over time regardless of VM size, larger VMs can achieve higher throughput compared to smaller ones, and there was no difference between regions observed.

Allocating network bandwidth between endpoints was first described in the context of Virtual Private Networks (VPN) [DGG+99]. In the cloud computing model, the VPN customers can be assimilated with the tenants from the cloud, and a VPN endpoint's equivalent is a VM. The customer-pipe model is the allocation of bandwidth on paths between source-destination pairs

of endpoints of the VPN. In this model, a full mesh between customers is required to satisfy the SLAs. In the hose model, an endpoint is connected with a set of endpoints, but the bandwidth allocation is not specified between pairs. Instead, the aggregate bandwidth required for the outgoing traffic to the other endpoints and the aggregate bandwidth required for the incoming traffic from the other endpoints in the hose is specified. These two models served as basis for bandwidth allocation in the cloud, with the hose model being frequently used [GLW+10; RST+11; BCK+11; JAM+13]. SecondNet [GLW+10] introduces an abstraction called virtual data centre (VDC) and multiple types of services (type 0 guaranteed bandwidth between VMs, type 1 per ingress/egress bandwidth reservation for VM, and best effort). Gatekeeper [RST+11] supports the hose model and sets minimum bandwidth guarantees for sending and receving traffic for a VM, which can be increased up to a maximum rate if unused capacity is available. It also proposes an extension to the hose model by composing multiple hoses for a VM to incorporate application communication patterns. EyeQ [JAM+13] uses a similar mechanism to Gatekeeper.

Several works extend the hose model. ElasticSwitch [PYB+13] provides minimum bandwidth guarantees by dividing the hose model guarantees into VM-to-VM guarantees, and taking the minimum between the guarantees of the two VMs. It rate limits the traffic from a VM to the specified guarantee or higher if there is available capacity. The unused capacity on a link is allocated proportionally to the bandwidth guarantees of the VM pairs using that link. Oktopus [BCK+11] uses the hose model (named virtual cluster, where all the VMs are connected through a single switch) and virtual oversubscribed cluster model (groups of virtual clusters connected through a switch with an oversubscription factor). Proteus [XDH+12] authors analyse the traffic patterns of several MapReduce jobs, finding that there is no need to allocate a fixed network bandwidth from the start to the end of the job, because the network demands of the applications change over time. They propose a time-varying network bandwidth allocation scheme: temporally interleaved virtual cluster, which is a variant of the hose model. CloudMirror [LTL+14] derives a network abstraction model, tenant application graph, based on the application's communication pattern. The applications considered in this work usually have multiple tiers or components, and each tier/component is formed of a number of VMs. Bandwidth within the component is allocated using the hose model. Bandwidth between components is allocated by guaranteeing each VM in a component C_1 a send bandwidth to send traffic to the VMs in a component C_2, and each VM in component C_2 is guaranteed a receive bandwidth to receive traffic from VMs in component C_1. Pulsar [ABK+14] provides end-to-end isolation for VMs and appliances (*e.g.*, load balancing, storage, monitoring). It forms a virtual data centre out of dedicated appliances connected to VMs through virtual switches, where each link between VMs and appliances has a throughput guarantee.

Implementation-wise, most of the works enforce rate limits at the end-host's hypervisor.

Profiling applications to determine their network throughput and keeping historical network throughput values are two aspects that can help to allocate bandwidth in a more efficient manner [XDH+12; LMB+14]. This is similar to modeling the relationship between application per-

	BW Guarantees	Work-conserving	Topology	Adaptability	Communication Pattern	Implementation	Latency Guarantees	VM placement
SecondNet [GLW+10]	Yes, Hose, VM-to-VM	No	No	No	No	Hypervisor, Source routing, MPLS, Central Controller	No	Yes
Gatekeeper [RST+11], EyeQ [JAM+13]	Yes, Hose	Yes	No congestion in the core	Yes	Yes	Hypervisor	No	No
Seawell [SKG+11]	No	Yes	No	No	No	Hypervisor	No	No
Oktopus [BCK+11]	Yes, Hose, Virtual oversubscribed cluster	No	Tree	No	No	Hypervisor, Central Controller	No	Yes
Proteus [XDH+12]	Yes, Temporally interleaved virtual cluster	No	Tree	Yes	Yes	Profiling, Central controller	No	Yes
NetShare [LRP+12]	No	Yes	No	No	No	Central Controller, Switches	No	No
PS-P [PKC+12]	Yes, Hose, Virtual oversubscribed cluster	Yes	Tree, Fat-tree	No	Number of VMs that communicate with a VM	Switches	No	No
Elastic-Switch [PYB+13]	Yes, Hose	Yes	No	Yes	Yes	Hypervisor	No	No
Hadrian [BJK+13]	Yes, Hose	Yes	Tree	No	Yes	Hypervisor, switches, central controller	No	Yes
Choreo [LDG+13]	Yes, Hose, VM-to-VM	No	Yes	No	Yes	Profiling, measurement, placement components	No	Yes
Cloud-Mirror [LTL+14]	Yes, Tenant Application Graph	No	Tree	No	Yes	Central controller, Elastic-Switch	No	Yes
Pulsar [ABK+14]	Yes, Virtual data centre	Yes	No	Yes	No	Central controller, Hypervisor	No	No
Cicada [LMB+14]	No	No	No	Yes	Yes	Hypervisor, Switches	No	Yes
Silo [JSB+15]	Yes	No	Tree	No	No	Central controller, End-host rate limiter	Yes, tail	Yes
QJump [GSG+15]	Yes	No	No	No	No	End-host rate limiter	Yes, tail	No
SNC-Meister [ZBH16]	Yes	No	No	No	Yes, traces	Centrall controller and end-host rate limiter	Yes, tail	Yes

Table 2.7: Systems providing network bandwidth and tail latency guarantees in data centres.

formance and network latency based on experimental data, as I have done in Chapter 5. Using real-time measurements of the network conditions, in particular throughput, has been employed to improve VM placement [LDG+13]. Choreo [LDG+13] measures the network throughput between VM pairs through packet trains, estimates the cross traffic, and locates bottleneck links. Based on the network measurements and application profiles (number of bytes sent), it makes VM placement decisions to minimise application completion time. The Choreo system is the closest system to the NoMora cluster scheduling architecture presented in Chapter 6, in which I

use network latency measurements between pairs of hosts and application performance predictions dependent upon current network latency to decide where to place tenants' VMs .

VM placement to provide network bandwidth guarantees starts by looking at subtrees in the topology to place the VMs, and goes upward in the tree to find a suitable allocation [BCK+11; XDH+12; BJK+13; LTL+14]. This naturally leads to the VMs being allocated within the same rack or within the same pod, which may hurt application availability if the links to the servers that run the application fail [LTL+14]. To mitigate this, CloudMirror [LTL+14] additionally incorporates an anti-affinity (anti-colocation) constraint in the VM placement algorithm. SecondNet [GLW+10] builds a bipartite graph whose nodes are the VMs on the left side and the physical machines on the right side, and then finds a matching based on the weights of the edges of the graph using min-cost max-flow. The weights are assigned based on the available bandwidth of the corresponding server. Hadrian [BJK+13] provides bandwidth guarantees for inter-tenant communication. The VM placement algorithm builds a flow network to express the VMs communication patterns and minimum bandwidth constraints. It then uses a greedy first-fit algorithm that respects the constraints to provide placement locality (a tenant's VMs are placed close to VMs it communicates with) and places a tenant's VMs in the smallest subtree possible. These approaches are similar to Quincy [IPC+09] and Firmament [GSG+16], which also model the scheduling problem as a min-cost max-flow problem. Firmament's network-aware policy avoids bandwidth oversubscription at the end-host by incorporating applications network bandwidth demands into the flow graph. [MPZ10] proposes an algorithm for traffic-aware VM placement that takes into account the traffic rates between VMs, and studies how different traffic patterns and data centre network architectures impact the algorithm's outcome.

Tail network latency guarantees Tail latencies have been recognised as a source of significant performance degradation [KPT+12; DB13; ZDM+12; XMN+13]. Several systems [JSB+15; GSG+15; ZBH16] have been developed in response. Silo [JSB+15] controls tenant's bandwidth to bound network queueing delay through packet pacing at the end-host. It then places VMs using a first-fit algorithm, while trying to place a tenant's VMs on the same server, in the same rack or further in the same pod, minimising the amount of network traffic that the core links have to cary. QJump [GSG+15] computes rate limits for classes of applications, ranging from latency-sensitive applications for which it offers strict latency guarantees to throughput-intensive ones for which latency can be variable. These systems provide worst-case latency guarantees. On the other hand, SNC-Meister [ZBH16] bases its design on the observation that tenants do not need worst-case guarantees, and that instead they require latency guarantees for lower percentiles, *e.g.*, 99.9$^{\text{th}}$percentile. SNC-Meister leverages this observation to admit more tenants in a data centre while keeping their (lower percentile) latency guarantees.

Application performance guarantees All of these approaches have looked at providing network bandwidth and (tail) latency guarantees, and, as a result, the application meets its performance guarantees. In my work, I change the point of view: if the tenant wants a certain performance for their application, what network conditions does the application need in terms of latency? If we know how the application reacts to latency and the current network conditions,

then we can place the tenant's application in the data centre ensuring the best performance achievable under the current network conditions. Furthermore, network bandwidth demands could be incorporated in the placement decision. Alternatively, one of the previously described systems or an orthogonal network bandwidth allocation framework [KJN+15] can be used to meet them.

Chapter 3

Measuring network conditions with the Precision Time Protocol (PTP)

While some data centre applications simply process and transfer data, many applications are latency-sensitive, such as Web search [AGM+10; KPT+12], social networking [AGM+10; KPT+12], ML frameworks [ABC+16] or key-value stores [AXF+12] (§2.3.2). These applications have stringent latency requirements, due to being interactive (search engine, social network) or due to their synchronous communication. Changes in network latency can lead to significant drops in application performance for latency-sensitive applications, as shown in Chapter 5 and Section 2.5 [BMP+17].

In this chapter, I investigate the use of the Precision Time Protocol (PTP) through an open source software implementation PTPd [PTP18], to measure network conditions in order to use it as a building block for a data centre network monitoring system in Chapter 4. PTPd offers measurements such as the master-to-slave delay, the slave-to-master delay, and an estimated one-way delay computed as average of the previous two delays (§2.2.4). RTT/2 is often not a good approximation for the one-way delay, as asymmetries often arise in networks [Pax06], caused by network congestion, data centre architectures, or link failures [ZTZ+14], leading to different latency values on the forward and reverse path.

I seek to answer the following questions:

1. What is the relationship between the one-way delay (OWD) metric reported by PTPd and the RTT reported by other tools?
2. How are the PTPd measurements affected by network congestion?
3. How are the PTPd measurements affected by virtualisation?
4. Can the PTPd measurements be used to determine other network information, *e.g.*, compute packet loss ratio?

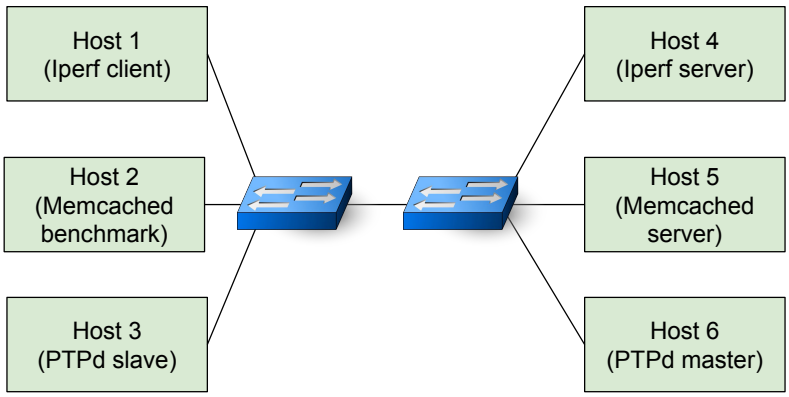


Figure 3.1: Testbed to analyse PTPd's behaviour under different network conditions.

5. How are the PTPd measurements affected by other network traffic originating from the same host?

3.1 Experimental setup and methodology

I use two testbeds for the experiments in this chapter. The first testbed in Figure 3.1 consists of six servers Intel Xeon E5-2430L v2 running at 2.40GHz, with Ubuntu 16.04, kernel version 4.4.0.64-generic, equipped with 10Gb/s Intel X520 NICs with two SFP+ ports. The servers are connected using two Arista 7050Q switches, and all network links are 10Gbps. This testbed does not have PTP-enabled NICs with hardware timestamping (§2.2.3). The experiments in this chapter use the default NIC settings. The two hosts running PTPd do not send or receive any other network traffic, thus the PTPd measurements can be affected only by the traffic originating from the four other hosts in the testbed (*Memcached* and *iperf* traffic).

For most of the experiments I do not use PTP-enabled NICs, because this type of NIC is not available to the tenants to access in cloud data centres. However, I additionally run experiments using PTP-enabled NICs to compare the results obtained in this second approach to the ones obtained using the testbed without PTP-enabled NICs. The second testbed is formed out of two hosts directly connected, running Ubuntu server 14.04 LTS, kernel version 4.4.0-62-generic. The host hardware is a single 3.5 GHz Intel Xeon E5-2637 v4 on a SuperMicro X10-DRG-Q motherboard, equipped with a Solarflare SFN8552 Network Interface Card (NIC) supporting PTP [Solb] with hardware timestamping (§2.2.3).

PTPd logs measurements such as the clock offset, the master-to-slave delay, the slave-to-master delay, and the one-way delay (§2.2.4). The interval for sending Sync and Delay Request messages can be configured in PTPd, up to 128 messages per second for each, expressed as $\log_2$ values between -7 and 7. The default setting is 0, which means sending 1 message per second of both *Sync* and *Delay Request* message types, and 2^{-7} means 7.8125ms between messages,

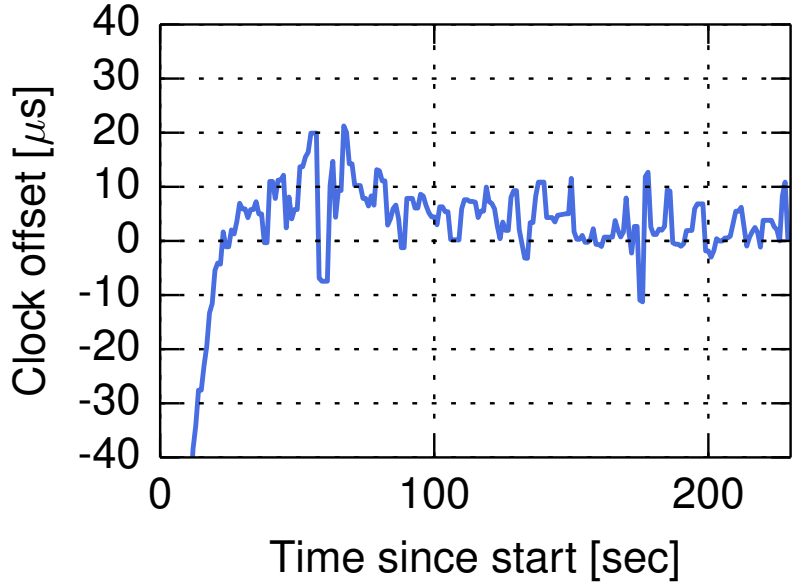


Figure 3.2: The slave's clock offset is within 20μs of the master's clock after less than five minutes after PTPd's start-up.

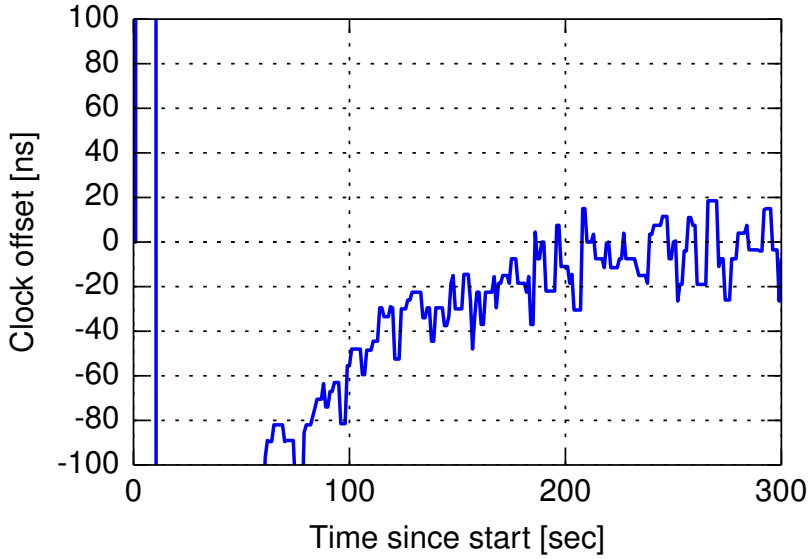


Figure 3.3: The slave's clock offset is within 40ns of the master's clock after less than five minutes after Solarflare's PTP daemon start-up.

with 128 messages per second. I call the number of messages sent per second *message frequency* in order to distinguish the name from message rate, since in the case of PTP, there is a set time interval between the messages exchanged, the messages being sent with a given frequency.

In all of the experiments, I wait for an initial period to reach a stable state before making changes to the system, *e.g.*, starting other applications that send traffic. After PTPd starts up, it performs an initial clock reset if the clock is off by one second. Then the slave clock gradually

synchronises with the master clock. Thus, before this convergence period ends, the system is not in a stable state, and this may distort the results of the experiments. Next, I measure on my first testbed the convergence period when using a message frequency of 1 message per second. I verify that five minutes are sufficient for the PTPd master and PTPd client to synchronise to within 20μs of each other (Figure 3.2). Allowing more than five minutes for convergence did not decrease the margin between the two clocks' values, with the clocks remaining within 20μs of each other. For the second testbed which uses PTP-enabled NICs, the clocks are synchronised within 40ns in less than five minutes (Figure 3.3). Hence, I wait for five minutes before running any intended experiment.

The results in the following sections represent sample runs of the experiments.

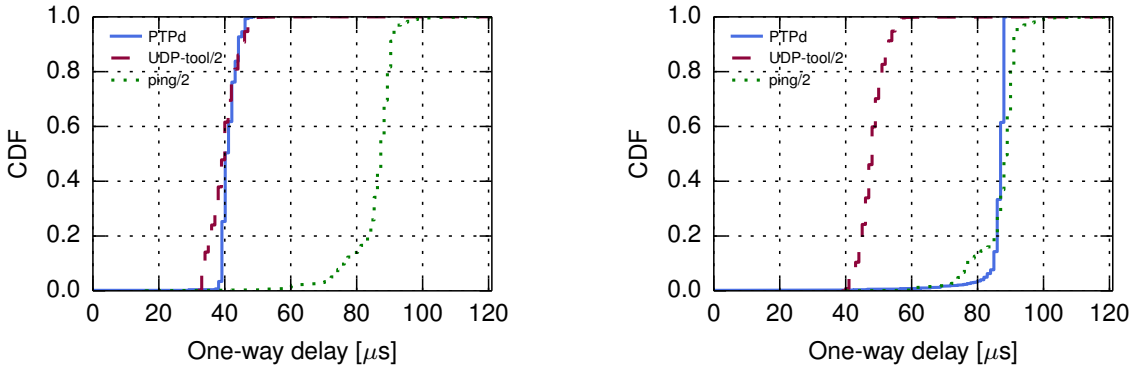
3.2 Measuring network latency

The experiments in this section aim to answer the first question posed in the introduction, namely to compare the one-way delay value computed by PTPd with the RTT values measured by two other measurement tools.

I measure the RTT in the first testbed between the PTPd master and PTPd slave hosts, using *ping* and the UDP-based tool [ZGP+17] that uses the Time Stamp Counter (TSC), and I compare the values obtained divided by two with the one-way delay reported by PTPd. For PTPd, I set the message frequency for Sync and Delay Request messages to 1 per second, and I run the clock synchronisation for 15 minutes. For the two other experiments, I run 1 million RTT measurements with the UDP-based tool, and 30,000 ping probes. There is no other network traffic in the testbed, and each test is conducted separately.

The network latency CDF is presented in Figure 3.4. Intuitively, one would have expected the one-way delay to be half of the values reported by the UDP-based tool, however this was not the case in the default configuration (Figure 3.4b). I investigated why this happened, and I found that changing the interrupt rate of the NIC at both the master and the slave by setting to zero the number of microseconds to wait before raising an RX interrupt after a packet has been received gives the expected results for the one-way delay reported by PTPd (Figure 3.4a). This experiment shows the drawbacks of software timestamping, and reinforces the importance of using hardware timestamping for obtaining precise measurements. Additionally, PTPd's message frequency can affect the OWD value, and this is discussed in detail in Chapter 4, Section 4.3.1.

Once the two clocks are synchronised, the one-way delay reported by PTPd is stable; there is no long tail for the reported one-way delay, due to filtering of abnormal values (§2.2.4). On the other hand, the RTT CDFs produced by the two other tools exhibit a long tail due to OS scheduling artefacts [ZGP+17]. The one-way delay reported by PTPd and the RTT/2 reported by the UDP-basel tool is approximately half of the median *ping* RTT/2 values. This



(a) Setting `rx-usecs` to 0

(b) Setting `rx-usecs` to 1

Figure 3.4: RTT/2 reported by *ping* and the UDP-based tool that uses the TSC [ZGP+17], and one-way delay reported by PTPd

difference may be due to how the ICMP traffic is treated in the network and at the end-host network stack [PCV+13]. Furthermore, *ping* may not be appropriate as a measurement tool for network latency in the cloud, because of the possibility that ICMP packets are treated differently, *e.g.*, being redirected for security checks [WN10].

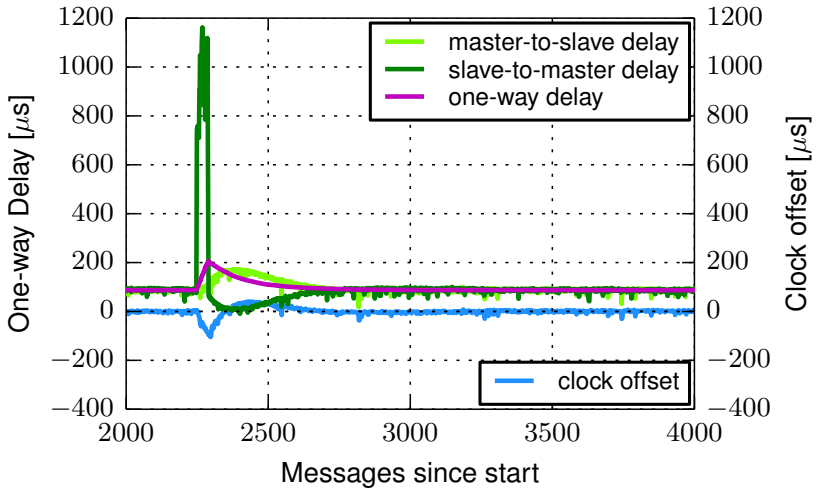
3.3 The effect of network congestion on PTPd measurements

The experiments in this section aim to answer the second question posed in the introduction, namely how network congestion affects the PTPd measurements.

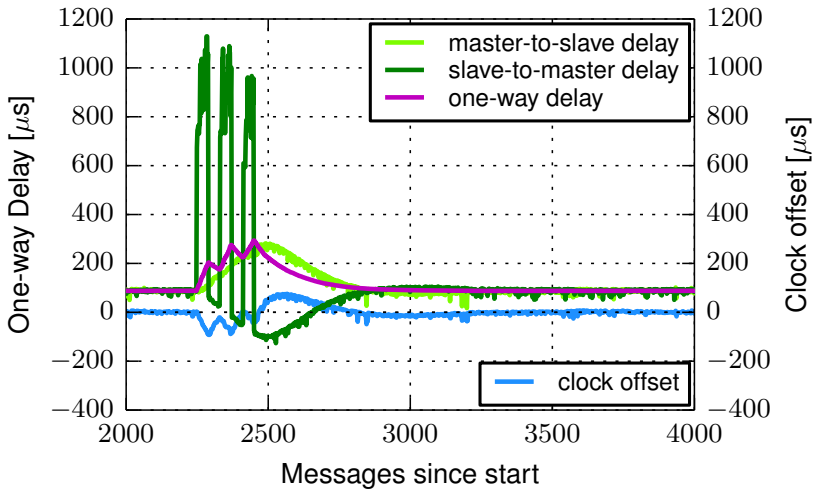
3.3.1 Concurrent network traffic

I study the effect of network congestion on the measurements reported by the PTPd slave using the testbed in Figure 3.1. In each test, I allow a clock synchronisation phase of 5 minutes for PTPd, before starting concurrently the two other applications, *Memcached* [Mem18] (with its corresponding benchmark *memaslap* [Prob]) and *iperf* in TCP mode. *Memcached* is a latency-sensitive application, for which increases in network latency lead to significant performance loss (§5.4). In this specific experiment, I set the interval for the Sync and Delay Request messages to 0.25 seconds (message frequency of 4 messages per second), but the results are similar for different message intervals. I run two experiments: i) a 5s iperf stream running (Figure 3.5a) and ii) three 5s iperf streams with 5s breaks between them (Figure 3.5b).

The first experiment (Figure 3.5a) shows that the congestion episode determined by iperf leads to an increase in the slave-to-master delay (on the iperf stream's direction). PTPd packets are queued in switches behind iperf's packets, thus it takes longer for the packets from the slave to reach the master, hence the increase in the slave-to-master delay. Consequently, the one-way delay increases. The PTPd slave interprets these changes as clock offset from the master clock,



(a) A 5s iperf stream starts running at second 300.

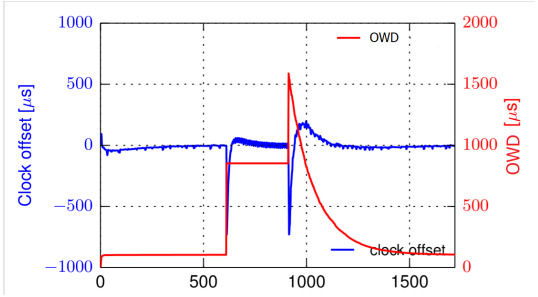


(b) Three 5s iperf streams start running at second 300s, 310s and 320s, respectively, with 5 seconds breaks between streams.

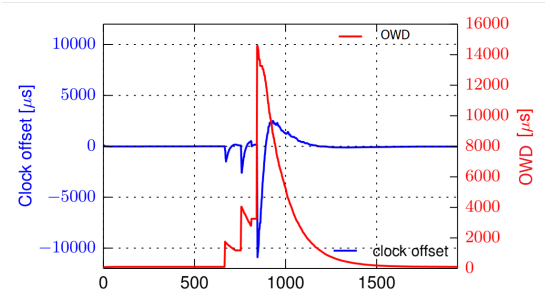
Figure 3.5: Network congestion effect on PTPd measurements.

then corrects its clock accordingly, and as a result changes also appear in the master-to-slave delay. After TCP exits the startup phase and reaches the steady state, and assuming that the iperf stream continues to run after this state is reached, the slave's clock will gradually reconverge, with the clock offset nearing zero. However, the one-way delay will still reflect an increased delay determined by the iperf traffic.

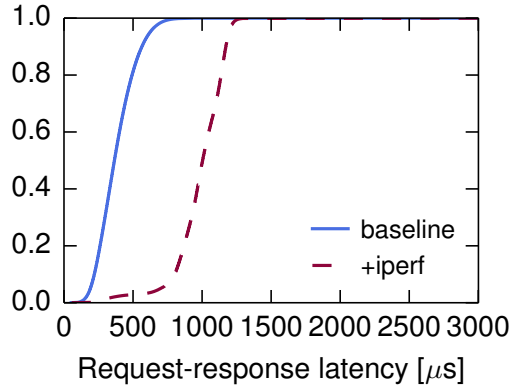
The second experiment (Figure 3.5b) shows that if there are several congestion episodes before the slave clock manages to resynchronise with the master clock, the one-way delay reported by the slave is not indicative of the actual delay, but it still indicates that there is an event (network congestion, link failure) on that network path. In this experiment, the first congestion episode caused by iperf has the same effect as in the first experiment. The next two intervals of iperf traffic produce further deviations to the slave-to-master delay, because the two clocks did not have time to resynchronise before the start of the next iperf stream. The figure illustrates how



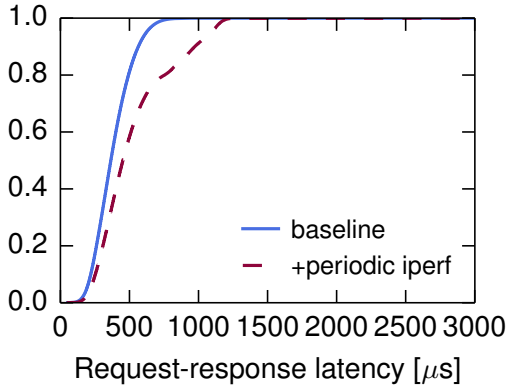
(a) Clock offset and OWD for Memcached, PTPd and iperf



(b) Clock offset and OWD for Memcached, PTPd and periodic iperf



(c) *memaslap* with iperf request-response latencies



(d) *memaslap* with periodic iperf request-response latencies

Figure 3.6: Network congestion effect on PTPd measurements and on *memaslap*'s performance.

the delays are gradually going back to the baseline values, but before this can fully happen, a new iperf stream starts. While this experiment shows that the one-way delay does not provide the true value of the latency between hosts at all times, the approach still has merits, and with appropriate data post-processing the accuracy of the OWD measurement could be increased.

I perform similar experiments as the ones presented in the previous paragraph to relate the effect that network congestion has on the PTPd measurements with the application performance of a latency-sensitive application, *Memcached*. These experiments show that the changes observed in the PTPd measurements can serve as an indicator that the performance of other applications running on the same network may suffer. Firstly, I run an experiment with the benchmark in parallel with the PTPd clock synchronisation, and no other competing traffic, to measure the baseline request-response latency of the Memcached benchmark, memaslap [Prob]. Next, I run *iperf* in parallel with the benchmark for 150 seconds. The PTPd measurements deviate from the normal values, due to queue buildup in switches, as illustrated in Figures 3.6a. After the TCP stream reaches steady state, the slave's clock resynchronises with the master clock. The OWD increases for the duration of the iperf stream, while the clock offset reconverges to normal values close to zero. After the iperf stream ends, the OWD returns gradually to its previous value. It should be noted that there is a period during which the slave clock is

again desynchronised, after the iperf stream ends. The return to the previous clock offset and OWD values happens gradually due to PTPd's clock servo algorithm (§2.2.4). Figure 3.6c shows increased request-response latencies due to the iperf traffic. In the third experiment, iperf runs for periods of 30 seconds alternating with breaks of 30 seconds (Figure 3.6b), PTPd measurements displaying the same behaviour as in Figure 3.5b. Similarly, Figure 3.6d shows an increase in request-response latencies due to iperf, but less than in the previous experiment, as iperf runs for a total shorter period of time.

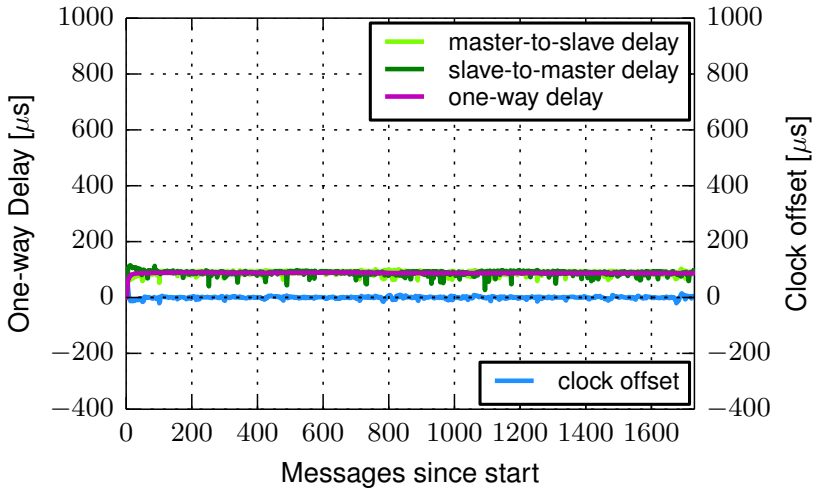
3.3.2 Changing the message frequency of the Sync and Delay Request messages

This experiment explores the time resolution at which network congestion affects the PTPd measurements by changing the interval at which messages are exchanged between the PTPd master and PTPd client. I perform the same experiment with *iperf* and *memaslap* concurrently running with PTPd, but *iperf* runs for a duration of 1s. I vary the interval at which the Sync and Delay Request messages are sent, from 1s down to 7.8125ms, meaning from 1 messager per second to 128 messages per second. This allows detection of congestion periods at milliseconds resolution. Increasing the message frequency beyond 128 messages per second would allow detection at an even higher resolution. In Figure 3.7a, it can be seen that iperf does not produce any change in the PTPd measurements, since the interval between messages is the same as iperf's runtime, hence it is not running long enough to delay the PTP packets. However, when the interval is decreased (Figures 3.7b and 3.7c), the iperf traffic leads to deviations in the PTPd measurements, and an increase in the one-way delay, similar to Figure 3.5a. The clock offset, master-to-slave and slave-to-master delays oscillate between larger values when the Sync and Delay Request interval is smaller (comparing the width of the lines in Figure 3.7b and Figure 3.7c). This may happen because of software timestamping, or because of PTPd clock servo algorithm's settings (§2.2.4). The one-way delay does not exhibit such significant oscillations, as it is computed as the average of the master-to-slave and slave-to-master delays.

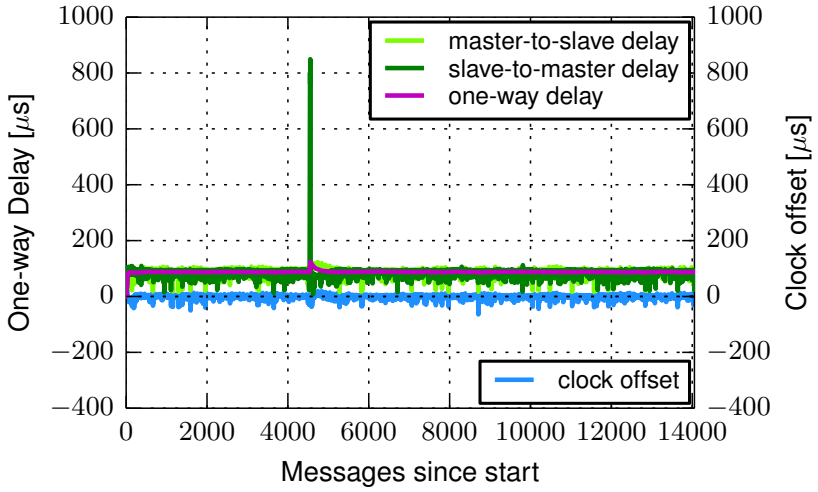
3.3.3 Convergence period for the PTPd measurements after network congestion

This experiment explores how long it takes for the PTPd measurements to return to the same values they had before network traffic that caused congestion was injected in the network. The experiment is performed for different intervals at which messages are exchanged between the PTPd master and PTPd client.

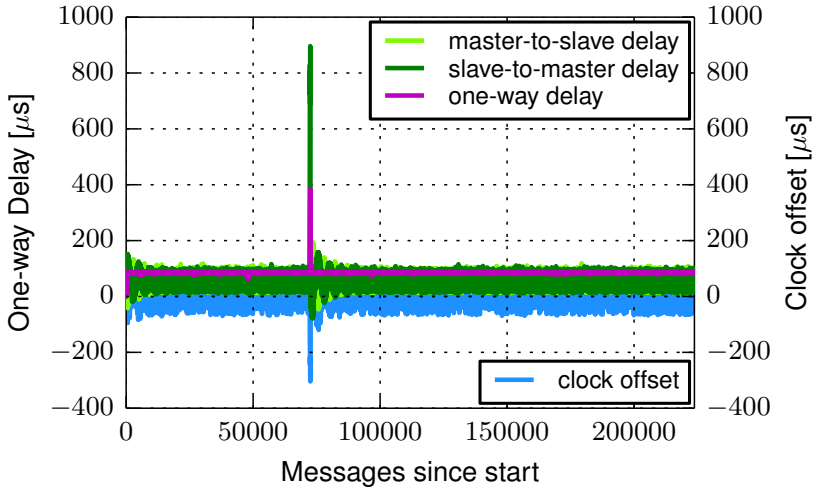
As shown in Section 3.3.1, the PTPd measurements do not instantly go back to their values after a network congestion period. Figure 3.8 illustrates how the one-way delay is affected by network congestion determined by a stream of iperf of 1s for different message frequencies.



(a) 1s interval for Sync and Delay request messages. A 1s iperf stream starts running at second 300, PTPd measurements do not detect it.



(b) 125ms interval for Sync and Delay request messages. A 1s iperf stream starts running at second 300, PTPd measurements detect it.



(c) 7.8125ms interval for Sync and Delay request messages. A 1s iperf stream starts running at second 300, PTPd measurements detect it.

Figure 3.7: Changing the interval for the Sync and Delay Request messages.

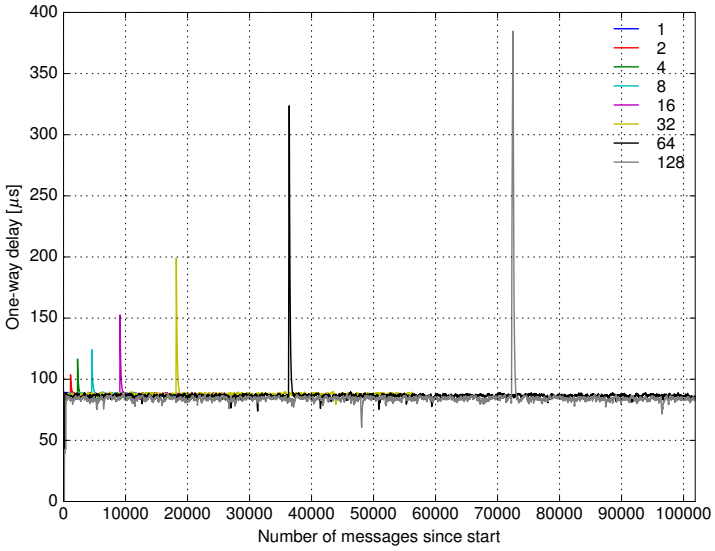


Figure 3.8: Number of messages needed for the one-way delay to return to normal values after network congestion caused by an iperf stream of 1s for different message frequencies.

# msg/s	# msgs for convergence	Approximate convergence period
1	-	-
2	255	63.75s
4	298	37.25s
8	428	26.75s
16	454	14.18s
32	536	8.37s
64	605	4.72s
128	684	2.67s

Table 3.1: Approximate number of messages needed to converge to the baseline OWD and how long it takes to reach the baseline OWD.

The experiment is run for 15 minutes. For 1 message per second, the one-way delay is not affected at all. The one-way delay increases with the message frequency, since with a lower message frequency the congestion period may actually be missed or not fully captured in the one-way delay. I count the number of samples greater than the baseline OWD value for the first testbed in Section 3.2 to determine how many messages are needed before the OWD returns to this baseline value after iperf finished running. The results are presented in Table 3.1, and it can be seen that the higher the message frequency is, the shorter the reconvergence time will be.

3.4 Measuring network latency in virtualised environments

The experiments in this section aim to answer the third question posed in the introduction, namely what is the impact of virtualisation on PTPd measurements.

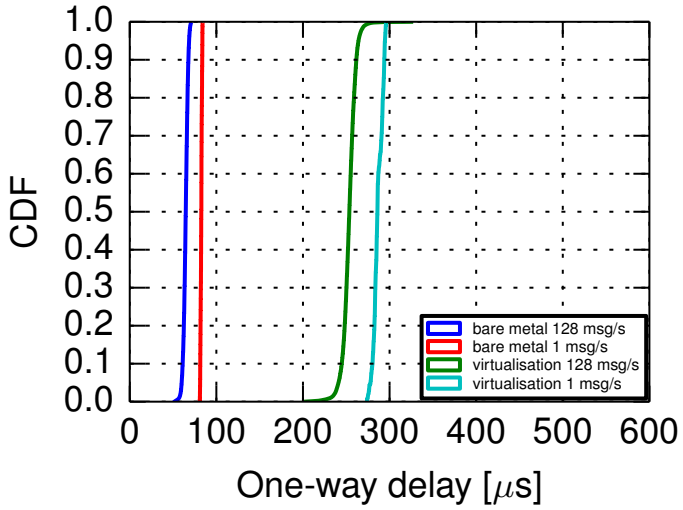


Figure 3.9: One-way delay reported by a PTPd client on a bare metal host and with virtualisation for different message frequencies.

	Min	**Average**	**Median**	**99th**	**99.9th**	**Max**	**Std. dev.**
Bare metal 1msg/s	80.85μs	82.59μs	82.68μs	84.06μs	84.15μs	84.15μs	0.8μs
Virt. 1msg/s	273.96μs	286.7μs	285.98μs	295.72μs	295.9μs	295.92μs	5.27μs
Bare metal 128msg/s	51.28μs	64.54μs	64.7μs	69.97μs	71.12μs	72.1μs	2.65μs
Virt. 128msg/s	201.69μs	253.33μs	253.59μs	271.21μs	284.86μs	327.25μs	7.62μs

Table 3.2: One-way delay reported by a PTPd client on a bare metal host and with virtualisation for different message frequencies.

All of the previous experiments were performed without virtualisation, on bare-metal hosts. In order to be able to interpret the measurement data collected in the cloud in the next chapter, where virtualisation is the norm, I run an experiment on the first local testbed to quantify the overhead of virtualisation on PTPd measurements, more specifically on the OWD, with the PTPd master and PTPd client running in VMs. The hypervisor used is Oracle VM VirtualBox version 5.0.40 Ubuntu r115130. The results of the two experiments are shown in Figure 3.9 and Table 3.2, and show that virtualisation adds almost 200μs of overhead and causes increased jitter to the OWD compared to a non-virtualised setting. Even so, the standard deviation of the OWD is not significant. These issues can be solved by using PTP-enabled NICs which provide hardware timestamping. Additionally, OS bypass through a custom software packet processing path [DSA+18] or through custom hardware [FPM+18] alleviates these issues. Also, some cloud providers (Amazon AWS) offer bare metal instances [AWS18b], thus the virtual switch overhead does not exist in this case.

To sum up, these results show that the OWD increases due to virtualisation, but the results are demonstrative of the *OWD's stability*, the OWD having low standard deviation even in the presence of virtualisation, making PTPd a convenient way to estimate network latency.

3.5 Estimating packet loss ratio

The experiments in this section aim to answer the fourth question posed in the introduction, namely can the PTPd measurements be used to measure other network conditions, such as packet loss.

Packet loss increases the latency perceived by the user, since dropped packets need to be retransmitted [GYX+15]. It is thus important to keep track of the packet loss ratios, as these can be correlated with the observed application performance. Additionally, tracking packet loss helps to uncover software or hardware faults in the network.

PTPd records the number of messages sent and received (Announce, Sync, Followup, Delay Request, Delay Response), and it is possible to export these numbers periodically. The counters can be reset after they are exported. On the slave side, a difference between the number of Delay Request and Delay Response messages would indicate packet loss. The packet loss ratio over a defined interval of time can be approximated as:

$$pkt\_loss\_ratio = 1 - \frac{\#Delay\_Response\_messages}{\#Delay\_Request\_messages} \tag{3.1}$$

Normal operation should see the same number of Delay Request and Response messages or a difference of at most one message. One disadvantage of computing the packet loss ratio in this way is that it does not account for the Announce, Sync and Followup messages that were potentially lost, as well as other types of packets that may be lost (ARP packets for example).

I verify if the proposed metric can be used as a coarse estimation for the packet loss ratio by artificially introducing packet loss in the network. I use *NetEm* [Hem] (§2.1.4), an enhancement of the Linux traffic control facilities, to emulate packet loss on the outgoing network interface of the host which runs the PTPd master. In this scenario, none of the Delay Request messages are lost, although in practice this may happen. Since outgoing PTPd packets are looped back via the IP_MULTICAST_GROUP [OLS08], loss conditions are applied on both the physical interface and the loopback interface. I use the *loss random* option of NetEm, which adds an independent loss probability to the packets outgoing on the chosen network interface. I use packet loss ratios of $1\%, 5\%$ and 10%, and I compute the packet loss ratio as described above to see if it matches the induced loss ratios. I run the clock synchronisation for 50 minutes with a packet loss of 1%, 10 minutes with a packet loss of 5%, and 5 minutes with a packet loss of 10%, and for each loss ratio I perform 5 runs. The results are presented in Table 3.3. It can be seen that the metric I defined can serve as a coarse estimate of the packet loss ratio over a defined interval of time.

NetEm packet loss ratio	Max. sample size (Delay_Request messages)	Packet loss median	Packet loss std. deviation
1%	2961	1.08%	0.23%
5%	571	5.43%	0.74%
10%	285	9.47%	0.34%

Table 3.3: Packet loss ratio computed based on the number of *Delay Request* and *Delay Response* messages reported at the PTPd slave.

3.6 PTP-enabled NICs

NICs supporting PTP are becoming increasingly available. The measurements performed by a PTP implementation that leverages this support do not suffer from end-host interference caused by other network traffic that originates from the same host. Furthermore, this type of NIC also removes the delay associated with the end-host network stack or virtualisation layer from measurements, the measurements thus reporting only the actual network latency.

I run experiments to see if PTPd measurements are adversely affected by other network traffic that originates from the same host to answer the fifth question posed in the introduction. This might happen because of end-host packet processing delays under increased load. On the second testbed described in Section 3.1, I compare the clock offset and one-way delay obtained from *sfptpd*, Solarflare NIC [Solb] PTP daemon which uses hardware timestamping (see Figure 3.10, note: ns y-scale), and from *PTPd* (see Figure 3.11, note: μs y-scale), which uses software timestamping, with and without running an *iperf* TCP stream between the hosts. One host is the PTP master, while the other acts as a PTP slave. It can be seen from the two figures that the clock offset reported by *sfptpd* is not affected by the iperf traffic. However, in the case of *PTPd*, the clock offset deviates when the iperf stream starts and ends. Furthermore, it should be noted that for hardware timestamping the clock offset's magnitude is nanoseconds, while for software timestamping it is microseconds.

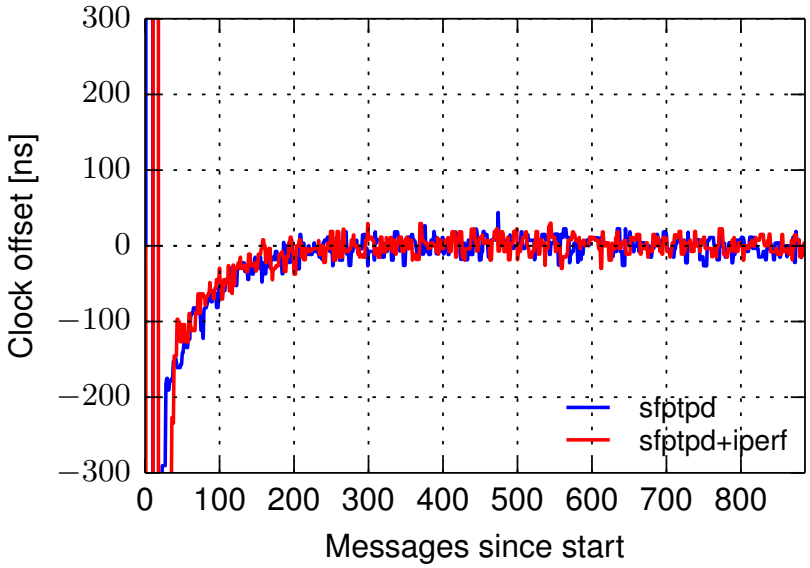


Figure 3.10: The clock offset reported by *sfptpd* is not affected by the iperf traffic, since it uses NIC hardware timestamping.

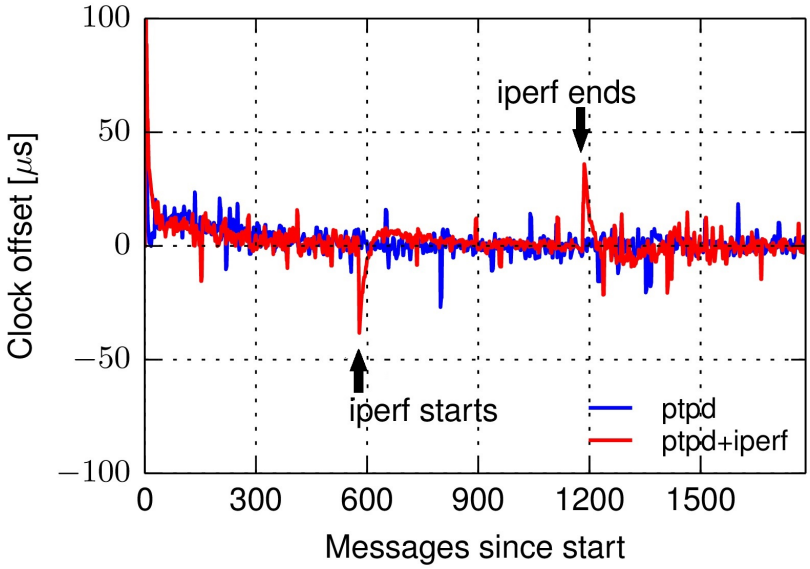


Figure 3.11: The clock offset reported by PTPd is adversely affected by the iperf traffic because of end-host interference.

3.7 Summary

In this chapter, I conducted an analysis to investigate and validate the use of PTP, through a software implementation PTPd, for measuring network conditions: network latency and packet loss. I first conducted experiments to determine how to use the one-way delay measurement offered by PTPd to estimate network latency (§3.2). I then showed that network congestion events are captured by the OWD measurement (§3.3). Next, I performed experiments on a virtualised testbed to determine the overhead of virtualisation on the OWD measurement, finding

that the OWD increases, but the standard deviation of the OWD does not change significantly (§3.4). Furthermore, I defined a metric to estimate packet loss ratio based on the number of messages exchanged between the PTPd slave and the PTPd master, and I verified that the metric can provide a coarse estimation for packet loss ratio (§3.5). Finally, I conducted experiments on a testbed with PTP-enabled NICs with hardware timestamping to show that the OWD is not affected by concurrent network traffic originating from the end-host when using hardware timestamping, but it is when using software timestamping (§3.6). The experiments in this chapter show that while PTP with software timestamping has drawbacks, it can offer estimates of network latency and packet loss that are useful for monitoring network conditions.

Chapter 4

Measuring the cloud network with PTPmesh

Network latency matters for certain distributed applications even in small amounts, affecting their application performance. Even though in recent years network performance has improved substantially in the cloud, network latency variability is still common in data centres (§2.4) [MK15]. In order to ensure the best application performance, one needs to be able to continuously measure the network latency across paths in data centres. Having up-to-date network latency values helps in tracking network SLAs for applications and in quickly finding failures [GYX+15; ALZ16]. These monitoring challenges can be addressed using a network monitoring system for data centres (§2.3.4). The system should be able to measure network latency across network paths and to detect packet losses, as these have a huge negative impact on application performance [GYX+15; ALZ16].

In the previous chapter, I validated the use of the Precision Time Protocol (PTP) through small-scale experiments for estimating network latency and packet loss. In this chapter, I propose *PTPmesh* as a network monitoring tool for data centres (§2.3.4). *PTPmesh*'s building block is PTPd [PTP18], whose measurement capabilities were analysed in Chapter 3. To validate the use of PTPmesh under real data centre network traffic, I carry out a measurement study in different cloud providers (Amazon AWS, Google Cloud Platform, and Microsoft Azure) in ten data centres in different regions across the world, highlighting their characteristics regarding latency magnitude, latency variance and packet loss.

PTPmesh is easy to deploy on cloud tenants' VMs, making it a feasible tool for cloud tenants to obtain network performance statistics without significant overhead and without needing access to any custom hardware at the end-host or in the network from the cloud providers (§2.3.4, Chapter 3).

4.1 Deployment scenarios

I consider two possible deployment scenarios for a system based on PTP in data centres [ALV08]. In the first scenario, the cloud provider deploys PTPd [PTP18] (or a different software implementation for PTP) in the hypervisor, possibly alongside a separate clock synchronisation mechanism. Several PTPd clients can run on the same machine in different PTP domains, and thus they do not interfere with each other. In the second scenario, the tenants themselves run PTPd inside their VMs and use the reported measurements to check the network conditions. PTPmesh's design follows the second scenario. In both scenarios, the PTP traffic should not be prioritised, and switches in the network should not be PTP-aware, otherwise the measurements would not be indicative of the actual network latency.

Since ECMP is used in data centres to load balance the traffic across the available network paths between two servers, the PTP traffic between the servers may not follow the same network path as other network traffic that exists between these two servers. To mitigate this issue, a similar approach to the one used in Pingmesh [GYX+15] and NetNORAD [ALZ16] can be used, specifically changing the port numbers on which PTPd is running. PTP uses port numbers 319 and 320 (§2.2.3). Since the port number is part of the ECMP hash computation, for every port number a different ECMP hash value is obtained. As a result, ECMP may select different network paths to route the PTP packets for different port numbers. Looping over a range of port numbers would ensure that the PTP traffic is sent over each distinct network path between two servers [GYX+15; ALZ16]. Moreover, if the cloud operator knows how ECMP is implemented on their switches and they do not use randomness in the ECMP hash function [GC13], then the cloud operator can define a list of port numbers for PTPd to ensure that each distinct network path between the two hosts is covered. Alternatively, if the trajectory of the packets can be traced using techniques such as the ones described in [TAL15; RZB+17], then it would be straightforward to verify whether all network paths between two servers are covered when using a range of port numbers for PTPd. I try the first approach in Section 4.8.

4.2 Measurement methodology

I use PTPd v2 2.3.1 [PTP18], with the latest source code from the public repository. I measure the one-way delay between multiple VMs from different cloud providers. In cloud computing terminology, a *region* is a geographical location where compute resources can be deployed, and it comprises one or more *zones* [1]. Usually, a region has three or more zones. For each of the three cloud platforms, Amazon AWS EC2 [2], Google Cloud Platform - Compute Engine [3], and Microsoft Azure [4], I choose several zones, as illustrated in Figure 4.1. I deploy two, four or

[1] `https://cloud.google.com/compute/docs/regions-zones/`
[2] `https://aws.amazon.com/ec2/`
[3] `https://cloud.google.com/compute/`
[4] `https://azure.microsoft.com/en-gb/`

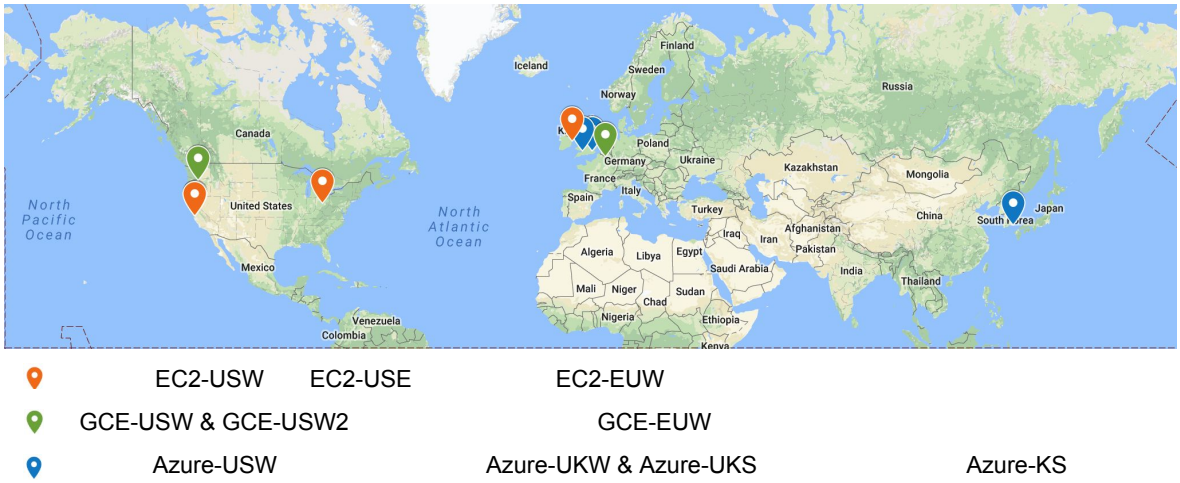


Figure 4.1: The zones in which PTPmesh was deployed to take measurements from different cloud providers.

ten VMs in each zone. The VMs run Ubuntu 16.04. I run the PTPd master on one VM, while the other VMs act as PTPd slaves, running simultaneously. The VMs' types, specifications and underlying hardware are described in Table 4.1. It should be noted that the latency measurements collected may be influenced by the underlying server hardware, as shown by previous research [NAZ+18].

Since currently multicast either requires additional configuration and expenses in the case of Amazon AWS [AWS] or is not supported at all in the case of Google Compute Engine [Pla] and Microsoft Azure [Azu], I used PTPd in unicast mode for the cloud deployment, with unicast negotiation and end-to-end delay measurement (§2.2.3). PTPd supports up to 2,048 unicast destinations, and it scales up to 1,000 slaves with high message frequencies of 32 messages per second[5].

I list the zones along with an assigned name to identify the traces collected. For Amazon AWS EC2, I run measurements in regions Ireland zone eu-west-1a (EC2-EUW), Northern California zone us-west-1b (EC2-USW) and Ohio zone us-east-2a (EC2-USE). For Google Compute Engine, the measurements are run in us-west1-b (GCE-USW) and europe-west1-b (GCE-EUW), and between us-west1-b (GCE-USW) and us-west1-d (GCE-USW2). For Microsoft Azure, I use UK West (Azure-UKW), UK South (Azure-UKS), US West (Azure-USW) and Korea South (Azure-KS). In the UK South, the VM type I use is the Standard D2s v3 and Standard_E16s_v3 or Standard_E32s_v3 (with Azure Accelerated Networking [FPM+18] enabled), while in the other zones I use the Standard D1 v2. I will refer to a zone as a data centre in the rest of the dissertation, but the underlying network topology and configuration of a zone is not disclosed by the cloud providers.

[5] `https://github.com/ptpd/ptpd/blob/master/INSTALL`

Cloud provider	Instance type	vCPU	CPU (Intel)	Mem (GB)	Storage (GB)	Storage Type	#NICs used	Network bandwidth
AWS	t2.micro	1	Xeon	1	10/30	Elastic Block Store [AWS18a]	1	moderate
GCE	n1-standard-1	1	Intel Sandy Bridge/ Ivy Bridge/ Haswell/ Broadwell/ Skylake	3.75	10	Standard Persistent Disk	1	$\leq$ 2Gbps
Azure	Standard D1 v2	1	Haswell E5-2673 v3	3.50	50	Local SSD	1	moderate
Azure	Standard D2s v3	2	Haswell E5-2673 v3	8	50	Local SSD	1	moderate
Azure	Standard_E16s_v3	16	Broadwell E5-2673 v4	128	256	Local SSD	1	high
Azure	Standard_E32s_v3	32	Broadwell E5-2673 v4	256	512	Local SSD	1	high

Table 4.1: VM types and specifications for the three cloud providers studied.

4.3 Measurement calibration

4.3.1 Message frequency impact

According to the PTP standard, the interval between messages can be set between 2^{-7} to 2^{7} seconds, which means a maximum of 128 messages per second. PTPd's experimental implementation allows message frequencies of up to 2^{30} messages per second. I perform several experiments where I vary the number of messages between 1 to 2^{7} per second to determine whether a different message interval yields different one-way delay values. I vary the message frequencies of both the Sync and Delay Request messages exchanged between the master and the slave, and I use the same message frequency for both.

I perform an experiment with a PTPd master and a single PTPd client running in the Azure-KS data centre with different message frequencies. Figure 4.2 shows the OWD CDF for different message frequencies. As the message frequency increases, the OWD decreases, going from median 262.92μs and 99$^{\text{th}}$percentile 286.6μs for 1 message per second, to 191.49μs and 99$^{\text{th}}$percentile 237.85μs for 128 messages per second. It is speculated that the cause of this behaviour is that, when the message frequency increases, the code that performs the timestamping remains in the cache, leading to smaller OWD values [6]. Another cause might be due to the way the interrupts are coalesced at the NIC, since messages are not timestamped by the NIC, but by the kernel.

While increasing the message frequency leads to better accuracy for the one-way delay measurements, the CPU utilisation and network bandwidth consumption increase. Since the initial goal was to have a low-overhead measurement system that runs as a service in a VM or in the hypervisor, choosing the message frequency implies a tradeoff between host and network resources consumption and measurement accuracy. I run measurements using the same setup in the Azure-KS data centre for each message frequency from 1 message per second to 2^{30} mes-

[6] Private communication with George Neville-Neil, developer of PTPd.

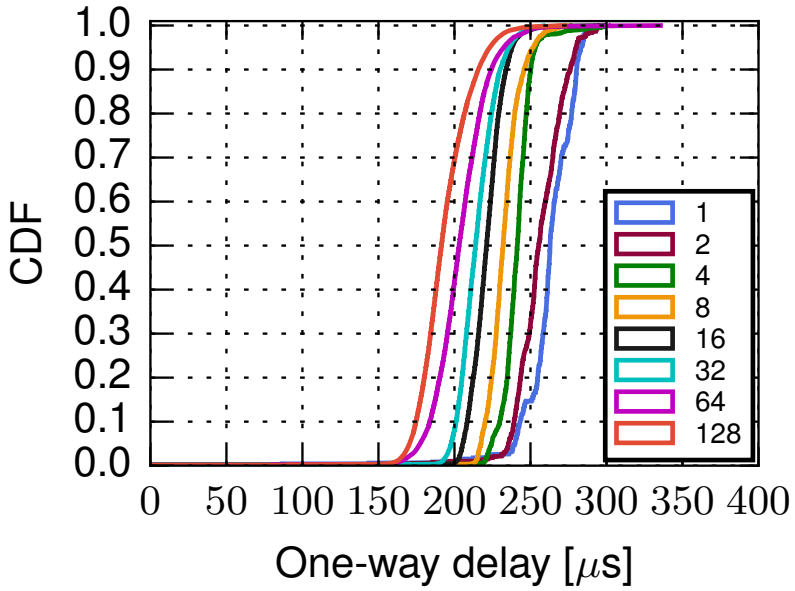


Figure 4.2: OWD measured using PTPd for periods of 15 minutes between two VMs in Azure-KS.

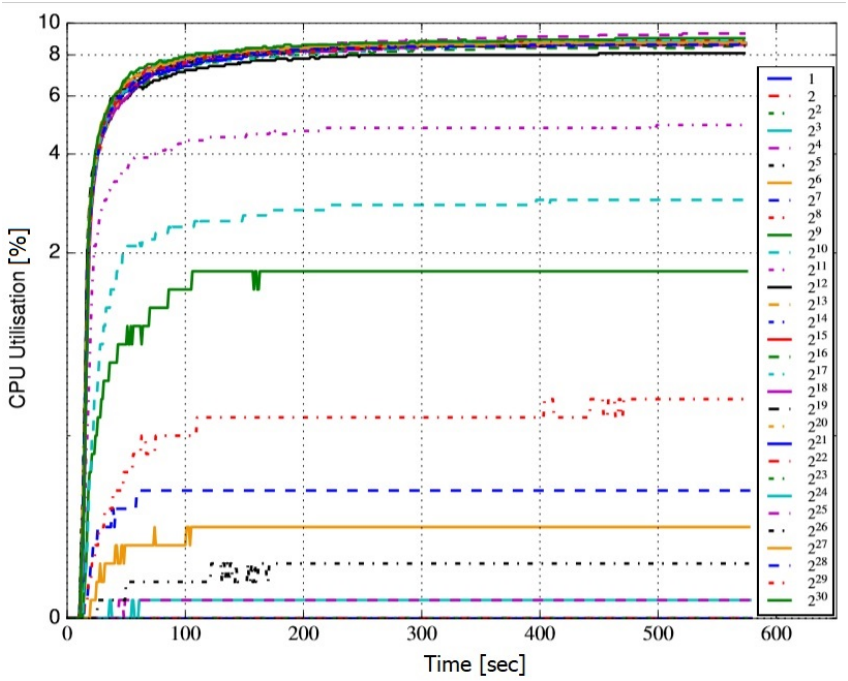


Figure 4.3: CPU utilisation of the PTPd master running on a VM in the Azure-KS data centre synchronising with one PTPd client.

sages per second for 15 minutes to monitor the average CPU utilisation (using *top*), memory, and send and receive network bandwidth (using *iftop*). Figure 4.3, Figure 4.4, Figure 4.5 and Figure 4.6 present the results for CPU utilisation and network bandwidth used by the PTPd master depending on the message frequency.

Figure 4.3 shows that it takes approximately 100 seconds to reach a value close to the maximum

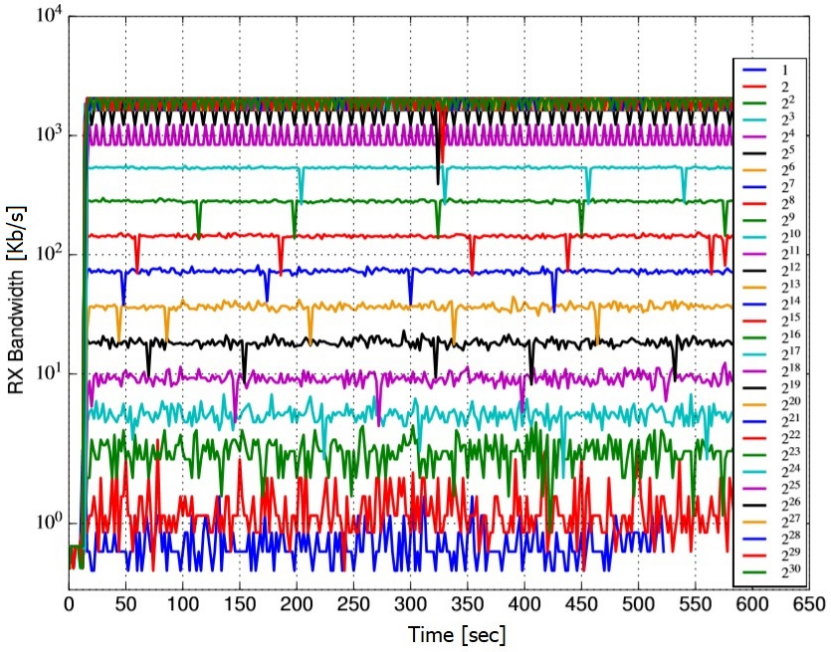


Figure 4.4: Receive network bandwidth of the PTPd master running on a VM in the Azure-KS data centre synchronising with one PTPd client.

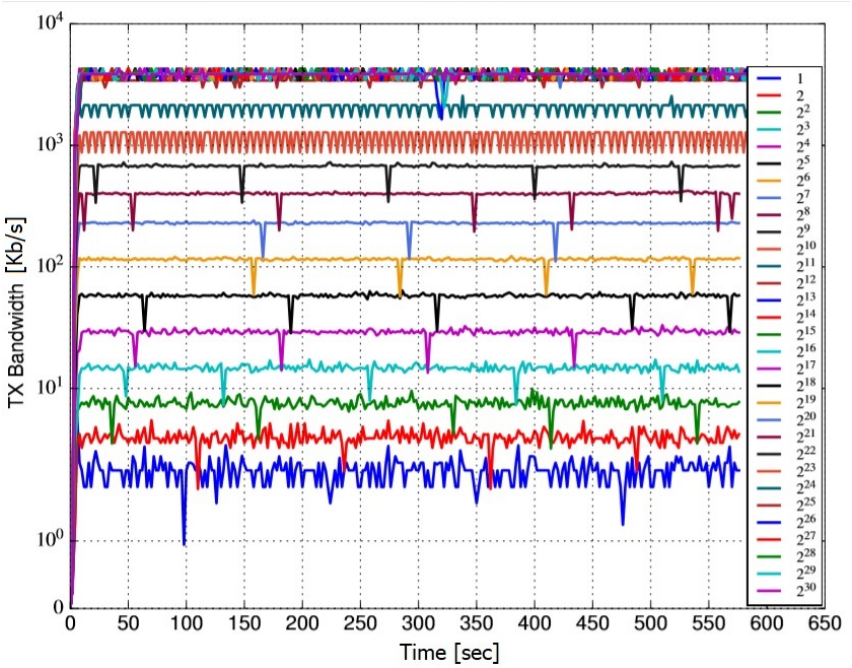


Figure 4.5: Send network bandwidth of the PTPd master running on a VM in the Azure-KS data centre synchronising with one PTPd client.

CPU utilisation of the PTPd master. CPU utilisation is less than or equal to 0.7% for message frequencies up to 2^7 messages per second. The CPU utilisation doubles with the doubling of the message frequency, the maximum CPU utilisation being less than 10%. When using two PTPd clients, the average CPU utilisation reported by the PTPd master increases to 0.4% for 32 messages per second, 0.7% for 64 messages per second, 1% for a message frequency of

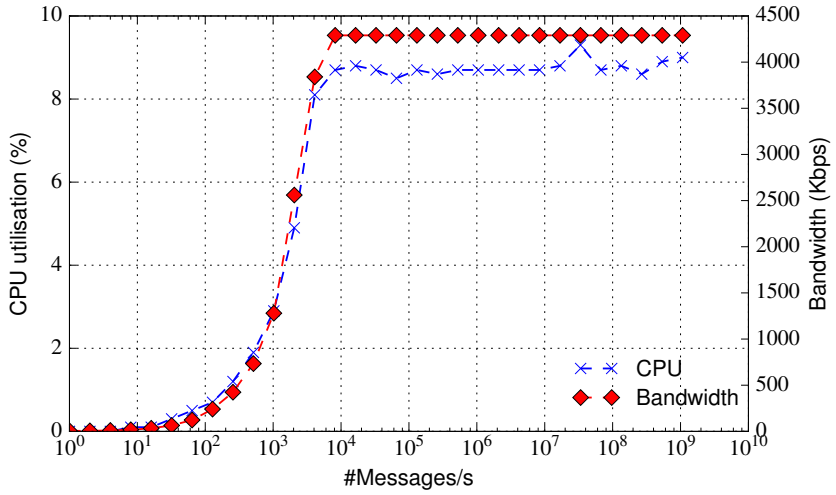


Figure 4.6: CPU and network bandwidth of the PTPd master running on a VM in the Azure-KS data centre synchronising with one PTPd client.

128 messages per second, and approximately 14% for high message frequencies of over 2^{12} messages per second. For a frequency of 1 message per second, the average CPU utilisation is almost 0% and the average network bandwidth consumption is 0.65Kb/s (receive) and 2.13Kb/s (send). For a frequency of 2^7 messages per second, the average CPU utilisation is 0.7% and the average network bandwidth consumption is 68.77Kb/s (receive) and 223.7Kb/s (send). From 2^{12} upwards, due to the end-host's limited packet processing capabilities, and the fact that the protocol operates in a request-response manner and taking into account the latency on the network, the message frequency does not actually achieve the set message frequency. The VM has a network bandwidth of 0.75 Gb/s, which is not reached for message frequencies greater than 2^{12}, the maximum bandwidth consumed being less than 10 Mb/s. Since these values are reported for a single slave, when synchronising with multiple slaves, it is expected that the network bandwidth will increase proportionally. For example, if using $1,000$ slaves, the average network bandwidth at the PTPd master would be 6.87 Mb/s (receive) and 22.37 Mb/s (send). The memory usage for the PTPd master is the same regardless of the message frequency, being 0.1% when using a VM with 1.6 GB RAM. However, the number of PTPd clients has an impact on the amount of memory used by the master [PTP18], with the maximum number of clients (unicast destinations) supported being 2048.

Table 4.2 presents the CPU utilisation and network bandwidth for different message frequencies. The results show that the accuracy of the OWD measurements improves as the message frequency increases with the average OWD decreasing, while the precision of the measurements stays roughly the same, the standard deviation for different message frequencies being almost the same. It should be noted that the OWD measurements may have been affected by concurrent network traffic within the data centre. On the other hand, the CPU and network resources double as the message frequency doubles.

To sum up, depending on the available resource budget, more accurate OWD measurements

# msg/s	CPU utilisation [%]	Average network bandwidth (receive) [Kb/s]	Average network bandwidth (send) [Kb/s]	Average [μs]	Std.dev. [μs]
1	0	0.65	2.13	262.11	21.54
2	0	1.12	3.88	255.88	19.15
4	0	2.21	7.38	240.63	14.1
8	0	4.35	14.38	232.42	13.19
16	0.1	8.68	28.44	221.02	13.21
32	0.2	17.35	56.53	214.7	13.56
64	0.4	34.43	112.12	203.67	18.4
128	0.7	68.77	223.7	193.37	17.56

Table 4.2: The setup has one PTPd master and one PTPd client. CPU utilisation and network bandwidth double as the message frequency doubles. OWD average goes down, while standard deviation is roughly the same.

can be obtained, but at the expense of more CPU and network resources. Memory requirements for running the PTPd master are consistently low. There is a tradeoff between measurement accuracy and resource consumption that should be considered when choosing the message frequency. A *lightweight network monitoring system* should not incur significant overhead on the end-host or the network. For example, Pingmesh [GYX+15] uses less than 45MB memory, the average CPU usage is less than 0.26%, and it sends only tens of Kb/s. Thus, based on the Pingmesh resource usage and depending on the number of PTPd clients that synchronise with one PTPd master, the message frequency for PTPd, and thus for PTPmesh, should be chosen between 1 and 32.

In Section 4.5, I conduct most of the latency measurements using a message frequency of 1 message per second on a setup with one PTPd master and three PTPd clients as part of PTPmesh. I choose this value in order to have similar CPU utilisation and network bandwidth consumption as Pingmesh. I additionally perform measurements with a message frequency of 128 messages per second for higher OWD measurements accuracy.

4.3.2 Number of concurrent PTPd clients

Another aspect that needs to be taken into account is the number of slaves a master can synchronise with before the end-host becomes overloaded because of processing too many messages, which may affect the measurement accuracy. To see if the number of clients affects significantly the OWD values, I perform a suite of experiments with PTPmesh on a local testbed with ten bare-metal hosts, using one PTPd master and a maximum of nine PTPd clients, and a similar experiment in EC2-USE (one VM PTPd master and up to nine VM PTPd clients), using 128 messages per second. Using the local testbed from Chapter 5, I found that the reported OWD is not affected by the number of clients, with median values between 18.5μs and 19μs, with standard deviations less than 1μs and a maximum value of 20μs across runs with different numbers

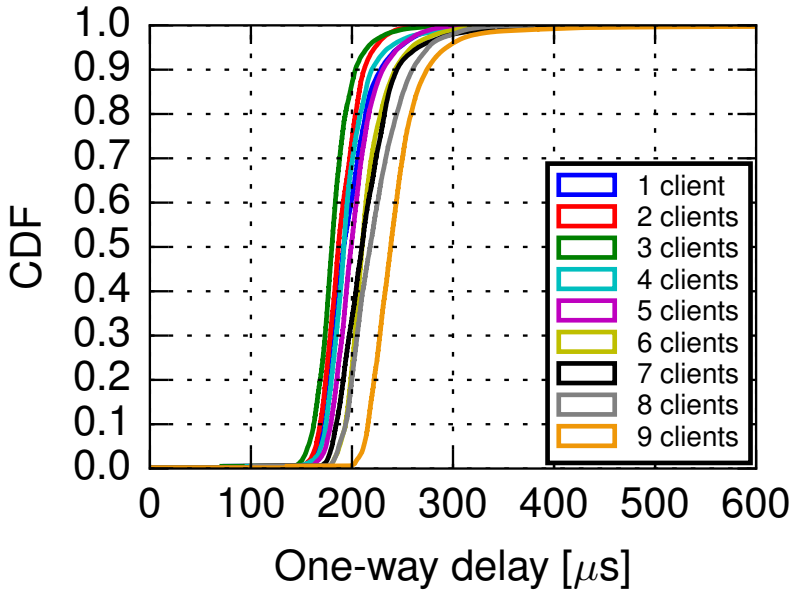


Figure 4.7: Varying the number of PTPd clients that synchronise with the PTPd master in EC2-USE.

of clients. In contrast, Figure 4.7 shows the OWD between one pair of VMs when varying the number of concurrent PTPd clients synchronising with the same PTPd master. The variations in the OWD when having up to four clients are not related to the number of clients. However, adding another client leads to an increase in the median latency of 7 μs. Having six or seven clients leads to increases in the OWD by approximately 10 μs. For eight clients, the median OWD is larger by approximately further 7μs. For nine clients, the median OWD is approximately larger by 20μs. In data centres from the two other cloud providers (GCE and Azure) I did not see any noticeable impact when using a maximum of four VM clients. I hypothesise that this is related to the hypervisor, since the ten hosts testbed is composed of bare metal hosts, but the results can also be influenced by competing network traffic in the cloud, whereas in the local testbed there was no other traffic. Given that all measurements were taken using a maximum of three simultaneous VM clients, my measurements were not affected by this behaviour. To mitigate this issue, the current infrastructure of PTPmesh can be extended to perform measurements between VMs independently in pairs. Alternatively, all VMs can be visited in a round robin manner with measurements running for several minutes per VM client to allow the VM client's clock to synchronise with the VM master.

Trace	#Msgs	Start time	Duration	Avg.(μs)	50th(μs)	99th(μs)	99.9th(μs)	Max(μs)	Std.dev.(μs)	#L.s.
EC2-EUW-1	1204053	2017-11-06 14:43:20	7d00h03m10s	304.87μs	289.57μs	415.18μs	516.5μs	2.69ms	49.71μs	19
EC2-EUW-2	879099	2017-11-06 14:43:22	5d15h12m50s	352.77μs	345.17μs	481.53μs	2.32ms	14.36ms	192.57μs	90
EC2-EUW-3	906750	2017-11-06 14:43:24	5d17h28m36s	352.17μs	350.97μs	459.5μs	616.3μs	3.16ms	50.68μs	48
EC2-USW-1	934978	2017-11-07 12:56:48	5d10h08m30s	259.63μs	256.93μs	335.08μs	486.3μs	1.73ms	33.18μs	7
EC2-USW-2	953109	2017-11-07 12:56:50	5d12h50m25s	279.22μs	275.86μs	361.42μs	474.35μs	1.25ms	29.08μs	12
EC2-USW-3	870190	2017-11-07 12:56:52	5d01h04m16s	287.82μs	283.44μs	363.88μs	429.06μs	686μs	24.15μs	5
GCE-EUW-1	1208861	2017-10-16 14:11:06	7d00h00m00s	138.32μs	97.1μs	1.2ms	2.74ms	7.72ms	223.32μs	237
GCE-EUW-2	1069898	2017-10-16 14:10:59	6d04h45m41s	138.29μs	87.34μs	1.44ms	3.48ms	6.58ms	268.65μs	216
GCE-EUW-3	1208306	2017-10-16 14:10:51	7d00h03m09s	132.78μs	82.97μs	1.38ms	3.6ms	8.83ms	275.73μs	243
GCE-USW-1	1210156	2017-10-16 16:38:32	7d00h02m57s	81.05μs	76.9μs	120.34μs	981.94μs	4.57ms	60.64μs	16
GCE-USW-2	1210507	2017-10-16 16:39:15	7d00h02m14s	72.07μs	71.35μs	92.24μs	119.22μs	531μs	8.65μs	1
GCE-USW-3	1209171	2017-10-16 16:41:33	6d23h59m56s	79.7μs	78.79μs	104.34μs	128.65μs	396μs	8.03μs	1
GCE-USW2-1	42907	2017-04-07 23:58:42	0d05h59m28s	191.65μs	180.66μs	526.84μs	908.27μs	1.21ms	63.04μs	5
Azure-UKW-1	1206919	2017-09-13 15:51:29	6d23h45m10s	441.37μs	447μs	529.27μs	570.62μs	1.38ms	47.4μs	2380
Azure-UKW-2	1160593	2017-09-13 15:51:33	6d17h11m17s	432.95μs	441.3μs	522.88μs	565.55μs	1.01ms	48.7μs	3979
Azure-UKW-3	1208739	2017-09-13 15:51:40	6d23h59m59s	412.59μs	419.62μs	483.48μs	521.42μs	827μs	39.96μs	134
Azure-USW-1	1203300	2017-09-13 15:26:09	6d23h08m41s	313.42μs	315.14μs	357.72μs	379.86μs	549μs	22.78μs	1
Azure-USW-2	1208955	2017-09-13 15:26:11	7d00h00m00s	282.46μs	281.21μs	330.64μs	362.23μs	717μs	15.31μs	3
Azure-USW-3	1208849	2017-09-13 15:26:16	6d23h59m59s	357.83μs	358.46μs	415.68μs	449.11μs	732μs	22.22μs	4
Azure-UKS-N1	108073	2018-02-22 20:11:00	0d16h37m07s	268.49μs	261.31μs	363.22μs	481.65μs	598μs	25.16μs	2
Azure-UKS-A1	96635	2018-02-22 22:18:24	0d13h54m09s	95.7μs	94.54μs	139.79μs	212.75μs	268μs	11.08μs	0
EC2-USE-1	21864606	2018-02-19 17:27:23	0d23h59m48s	181.96μs	172.05μs	291.55μs	411.82μs	2.04ms	30.71μs	139
EC2-USE-2	21864606	2018-02-19 17:27:23	0d23h59m49s	197.33μs	190.08μs	293.74μs	390.46μs	1.77ms	27.14μs	79
EC2-USE-3	21891242	2018-02-19 17:27:23	0d23h59m49s	188.53μs	196.83μs	301.86μs	406.26μs	1.48ms	30.65μs	86
GCE-USW-1	19378393	2017-12-22 22:31:59	1d10h04m00s	65.48μs	64.33μs	89.08μs	106.15μs	451μs	7.14μs	0
GCE-USW-2	21161197	2017-12-22 22:32:17	1d09h17m34s	70.4μs	69.47μs	92.8μs	106.11μs	295μs	8.35μs	0
GCE-USW-3	20854143	2017-12-22 22:32:29	1d07h52m56s	58.47μs	57.94μs	76.02μs	86.28μs	286μs	6.33μs	0
Azure-UKS-1	20164042	2018-02-17 22:12:21	0d23h59m48s	286.58μs	269.34μs	684.45μs	884.02μs	1.22ms	74μs	2235
Azure-UKS-2	21111652	2018-02-17 22:12:21	0d23h59m48s	271.41μs	249.55μs	724.43μs	907.24μs	1.37ms	84.65μs	4747
Azure-UKS-3	17427943	2018-02-17 22:12:21	0d23h59m48s	340.02μs	322.17μs	760.13μs	949.7μs	1.29ms	81.7μs	2793
Azure-UKS-A2	5043445	2018-02-23 12:47:02	0d05h54m20s	83.23μs	82.28μs	118.92μs	178.79μs	459μs	9.11μs	0

Table 4.3: Traces collected in data centres across the world from three cloud providers. The last column represents the number of latency spikes (l.s.) ($> 500\mu$s) observed throughout the trace.

4.4 Datasets

I collect several datasets whose characteristics are presented in Table 4.3. A trace represents the measurements taken between two VMs (master and client). The trace is the log of a PTPd client running in a VM. I list the start time and duration of the trace. Each trace is identified by the assigned name of the data centre and a number. For the first part of the table, the low message frequency was used (see Section 4.5.1), while in the second part of the table, the high message frequency was used (see Section 4.5.2). These traces are indicative for the temporal perspective of network conditions in data centres, as they have been captured for periods of up to a week. The spatial perspective is limited, since I use a maximum of three VM PTPd clients at the same time, hence I do not capture the full scale of conditions in the studied data centre. All datasets, except one, contain measurements taken between VMs that are located within the same data centre (zone). One dataset (GCE-USW-2) contains measurements taken between VMs that are located in different data centres (zones) within the same region (§4.2).

4.5 One-way delay (OWD) measurements

In the first instance, I set the number of Sync and Delay Request messages to 1 per second, since this is the default value configured in PTPd, which will be named in the rest of the chapter as the low message frequency. I run a full week of measurements in six data centres using the low message frequency. Additionally, I perform measurements in three data centres for one day using a higher message frequency of 2^7 messages per second, which will be named in the rest of the chapter the high message frequency. The challenge with using a higher message frequency is, on one hand, the increased CPU utilisation and network bandwidth at the end-host, while on the other, the amount of data collected for which additional storage is needed if measurements are performed for an extended period of time. The advantages of using a higher frequency are better OWD accuracy (§4.3.1) and detection of possible network congestion events with a higher resolution. Regardless of the message frequency used, the OWD values offered by PTPmesh can serve as reference for normal network conditions and can be used to detect anomalies.

4.5.1 Low message frequency measurements

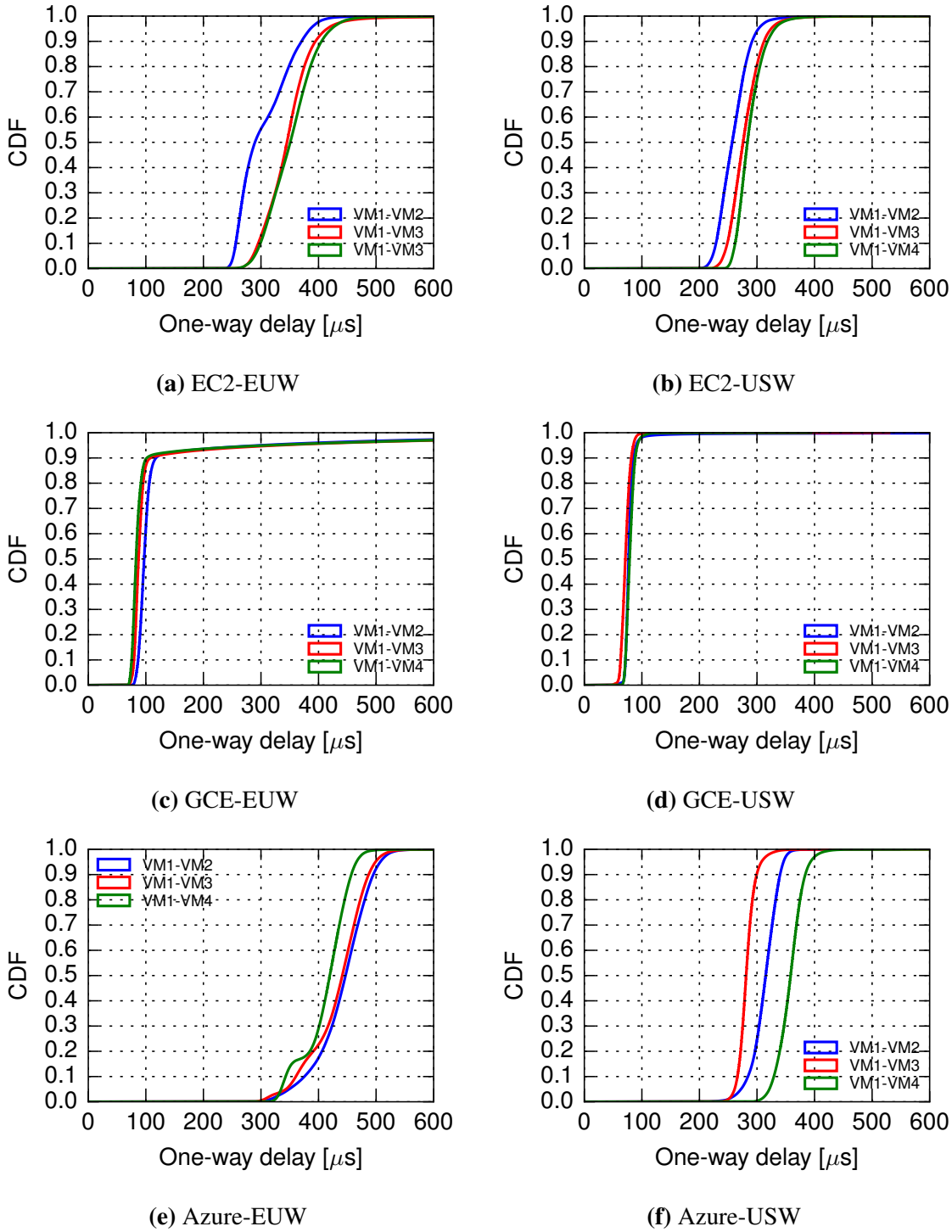
Latency magnitude. Table 4.3 lists the average, median, $99^{\text{th}}$and 99.9^{th}percentiles, maximum, standard deviation for the OWD values, and the number of latency spikes (a sudden increase in latency to values over 500μs) for the trace. OWD values are higher in the EU data centres than the US data centres for EC2 and Azure. The GCE-EUW data centre OWD values are similar to the ones in the GCE-USW data centre, the difference coming from the extended period of increased latency that can be seen in Figure 4.9b. Most of the traces have maximum observed

OWD values in the order of milliseconds. Figure 4.8 shows the CDFs of the one-way delay for the traces.

In Figure 4.9a, in the EC2-EUW-2 trace multiple latency spikes can be observed, with a maximum of 14.364ms. In the GCE-EUW trace, the OWD values are less or slightly higher than 100μs up to the 90thpercentile, with a maximum 99thpercentile of 1.44ms and maximum 99.9thpercentile of 3.6ms amongst the three VM pairs. In contrast, for GCE-USW data centre, the maximum 99this 120.34μs, and only in the case of the trace between VM1 and VM2 the 99.9thpercentile is higher, 981.945μs, compared to the traces for the two other VM pairs. The timeline of the measurements between VM1 and VM2 (Figure 4.10) shows multiple latency spikes, being different than the two other pairs (Figure 4.9h). The traces captured in the GCE-EUW data centre stand out in comparison to the other traces collected, since they contain major disruptions for latency values over a prolonged period, accompanied by a high packet loss ratio. Between 2017-10-17 09:25 and 2017-10-18 05:16 the OWD reported varied greatly, reaching a maximum value of 8.83ms, with a significant part of the latency spikes of over 500μs taking place during this interval. These millisecond-scale latencies indicate switch queueing and packet loss [RBB+18]. These events can be noticed for all three VM pairs, which can lead to the hypothesis that these events were data centre-wide, or that the VMs were placed within the same rack or on the same host. While the median latencies within the same data centre are between 71 and 97 μs, the median latency between two data centres in the same region is 180 μs (GCE-USW2-1), almost double compared to the one ones within a data centre.

The Azure-UKW data centre traces show a decrease of the OWD values of approximately 100μs towards the end of the trace (Figure 4.9c), which corresponds to the network traffic for Sunday. The last part of the trace was captured on Monday, showing an increase in the OWD values back to the values before Sunday, except for the VM1-VM3 pair. In the case of the Azure-USW data centre (Figure 4.9i) in the second day of measurements (after 172800 messages), a sudden decrease by approximately 50μs in OWD can be noticed for all three pairs for a period of time, followed by an increase for the OWD to values higher by approximately 50μs than the ones before the dip. It is interesting to see that the traces share similar characteristics for certain changes in the OWD values, meaning that the events were data centre-wide or that the VMs were placed within the same rack or the same host. I also perform experiments using more powerful machines in Azure-UKS (Azure-UKS-N1), the median latency is in similar ranges to the ones obtained using slower machines. I additionally perform experiments with VMs with the new feature [FPM+18] enabled, which removes most of the software-based networking stack into FPGA-based smartNICs, and found that the one-way delay reported is significantly lower than the other reported values, with median values of 94.54μs with low message frequency (Azure-UKS-A1) and 82.28μs with high message frequency (Azure-UKS-A2). Recently, EC2 has started offering a similar option using SR-IOV [Ama18], but I have not performed measurements using it.

Latency variance. An important aspect of network latency is latency variance [DB13], as it can cause a decrease in application performance. If the variance is low, then the application per-



(a) EC2-EUW

(b) EC2-USW

(c) GCE-EUW

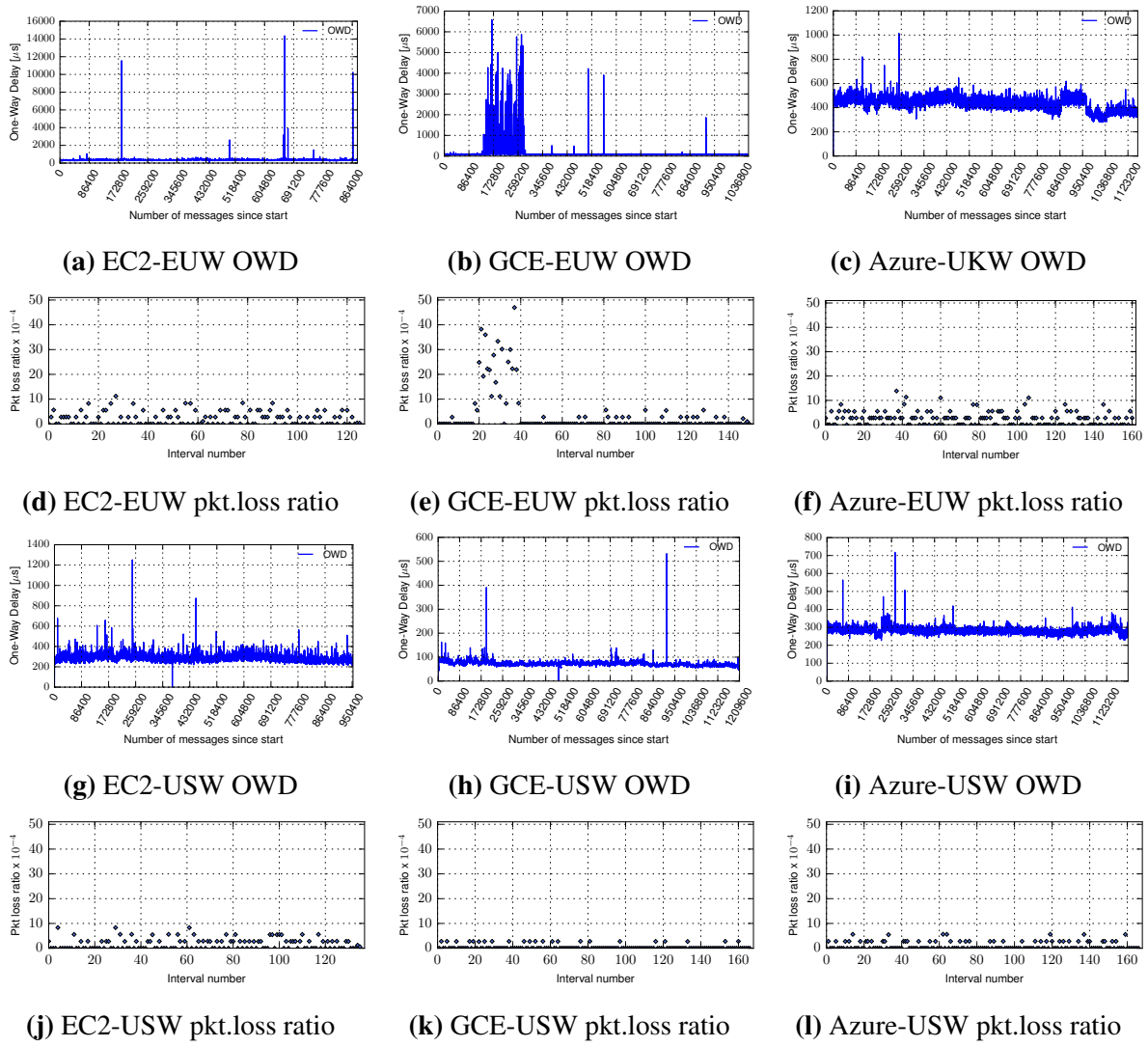
(d) GCE-USW

(e) Azure-EUW

(f) Azure-USW

Figure 4.8: CDF of OWD in different data centres using the low message frequency.

formance will be determined by the median latency observed, essentially reducing the problem to improving the static component of latency in data centres (Chapter 5). To this end, I compute the standard deviations of the OWD measurements over different intervals of time. The histograms in Figure 4.11 show the standard deviations of the OWD for intervals of 1 minute, 10 minutes and 1 hour, binned in bins of size 1, truncated to 100. The distributions for the standard deviations OWD are skewed to the right, towards small values, with a few values that are larger



(a) EC2-EUW OWD **(b)** GCE-EUW OWD **(c)** Azure-UKW OWD

(d) EC2-EUW pkt.loss ratio **(e)** GCE-EUW pkt.loss ratio **(f)** Azure-EUW pkt.loss ratio

(g) EC2-USW OWD **(h)** GCE-USW OWD **(i)** Azure-USW OWD

(j) EC2-USW pkt.loss ratio **(k)** GCE-USW pkt.loss ratio **(l)** Azure-USW pkt.loss ratio

Figure 4.9: OWD and packet loss ratios over 1-hour intervals between VM1-VM3 in EU and US data centres over one week.

than the rest.

The histograms for the two AWS EC2 data centres are similar. When looking at periods of 1 minute, the standard deviations fall mostly between 0μs and 10μs (medians 5.44μs and 4.59μs). When looking at periods of 10 minutes, the standard deviations fall between 10μs and 20μs (medians 15.34μs and 12.42μs), while when looking at periods of 1 hour, the standard deviations fall between 10μs and 30μs (medians 22.67μs and 18.31μs).

The histograms for the two GCE data centres are similar, but they are different from the two other cloud providers, in that the standard deviations of the OWD values are smaller. For the GCE-USW trace, for 1 minute intervals, most of the values are between 0μs and 1μs (median 0.759μs). When looking at periods of 10 minutes, the standard deviations are between 0μs and 5μs (median 2.84μs). For 1 hour intervals, most of the values are between 1μs and 10μs (median 4.63μs). The median values for the GCE-EUW trace are slightly higher, due to the increase in the OWD for a long period of time (1.5 days).

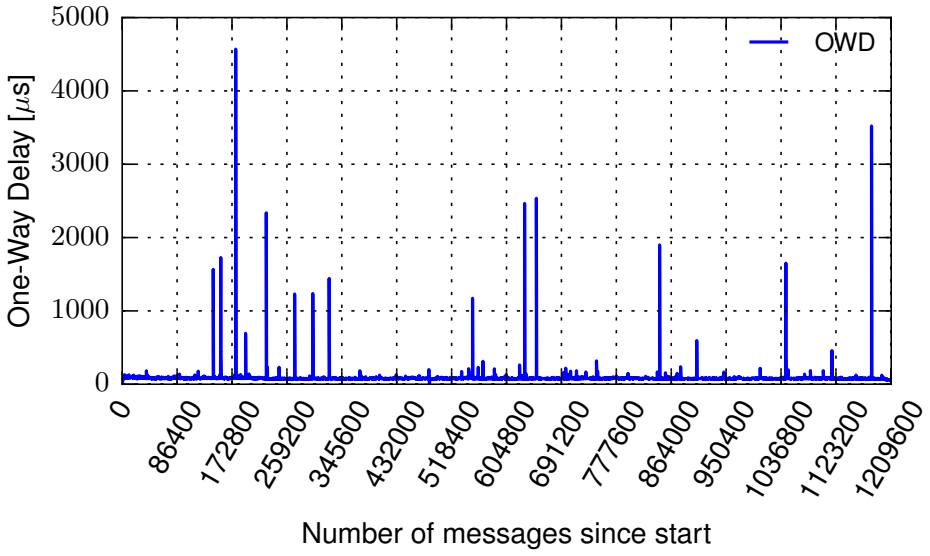


Figure 4.10: Measured OWD between VM1 and VM2 in GCE-USW data centre.

On the other hand, there are differences between the two Azure data centres. In the case of the Azure-UKW trace, for 1 minute intervals, most of the values are between 1μs and 15μs (median 7.42μs). When looking at periods of 10 minutes, the standard deviations are between 10μs and 20μs (median 16.6μs). For 1 hour intervals, most of the values are between 10μs and 30μs (median 20.29μs). In the case of the Azure-USW data centre, the values are slightly lower. For 1 minute intervals, most of the values are between 0μs and 10μs (median 4.49μs). When looking at periods of 10 minutes, the standard deviations are between 5μs and 15μs (median 9.63μs). For 1 hour intervals, most of the values are between 10μs and 20μs (median 12.17μs).

The results show that the OWD in GCE has the lowest variance. EC2 and Azure are similar, with more variance seen for EC2. When enabling [FPM+18], the Azure latency variance profile becomes similar to the GCE one. Having less variance for OWD is better, since tail latencies can lead to a decrease in application performance [DB13].

4.5.2 High message frequency measurements

Latency magnitude. The OWD values measured using high message frequency are lower than the ones measured using the low message frequency (§4.3.1). The EC2-USE OWD median values are between 172μs and 196μs (Figure 4.12a). The GCE-USW OWD values have medians between 58μs and 69μs (Figure 4.12b). In GCE, the low message frequency measurements may have been redirected through switch gateways (due to the low throughput of the measurements run, less than 20kbps), whereas the high message frequency ones may have been sent host-to-host [DSA+18]. The median OWD values for Azure-UKS are between 271μs and 340μs (Figure 4.13). The three traces are correlated, displaying periods of increased latency at the same time and having the same shape.

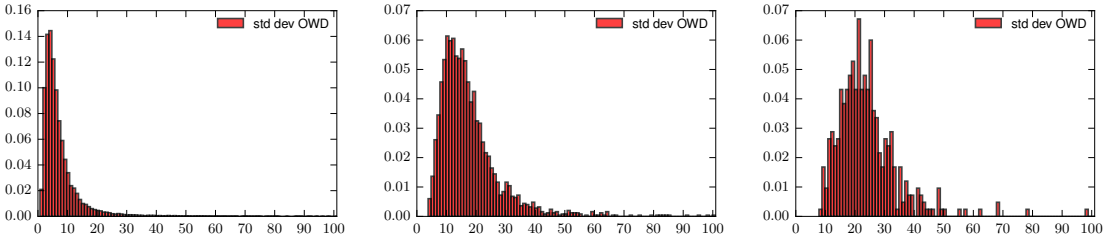
Latency variance. In the case of GCE-USW, the latency variance profile is similar to the one obtained using the low message frequency. The EC2-USE latency variance profile is similar to the EC2-USW one, and the Azure-UKS one is similar to the Azure-UKW one, even if EC2-USW and Azure-UKW latency variance profiles have been obtained using the low message frequency.

4.5.3 High value OWD events timescale

After analysing the general characteristics of the traces, I take a closer look at the timescale of high OWD events, as these are important in the context of latency-driven performance-aware cluster scheduling. If the OWD is stable, then the VM placement decision will have a lasting effect throughout the execution of the application. On the other hand, admitting more applications into the data centre can lead to increased network utilisation, and hence increased network latency. If the OWD is not stable, it might be better for certain applications to be preempted and migrated to a different placement.

Long timescale events are considerable changes in latency during several hours or days. These types of events are evident in the week-long Azure traces, where the latency decreases during the weekend. Similarly, the GCE EU traces display significant increases in latency for more than one day. Also, in the Azure-KS data centre, after restarting the VMs, I consistently got substantial latencies (median 1.391ms) compared to previous values (median 191.486μs), that I kept on measuring even after several VM restarts, and across almost one month of measurements. The first time I observed these large latency values was on the 29th of December 2017, and the last time I performed measurements in this data centre was 23rd of January 2018. In this case, it might be better to migrate the application to a different data centre.

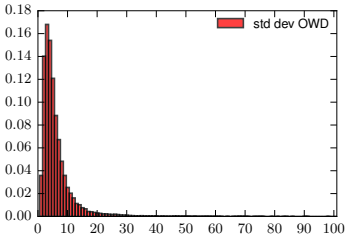
Short timescale events are transient latency spikes. The difference between the measured median latency and the maximum latency observed during the spike should be substantial (*e.g.*, more than 500μs). For example, while performing the EC2-USE measurements, the latency has suddenly increased substantially from median 200μs to median 1.75ms, as seen in Figure 4.14. It can be noticed that the latency values return for a brief period of time (2s, with high message frequency) to the previous values, but then the latency increases again. Similarly, the Azure-UKS traces (Figure 4.13) display several short latency spikes. In this case, if the OWD is not stable and suffers from frequent changes, it may be better for the application to be migrated to a different placement.



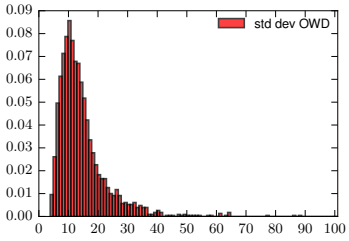
(a) EC2-EUW 1 min

(b) EC2-EUW 10 min

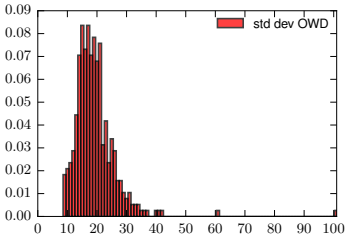
(c) EC2-EUW 1 hour



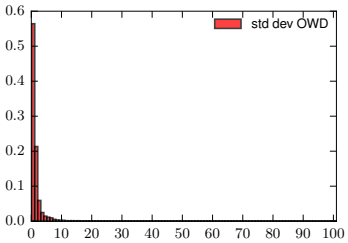
(d) EC2-USW 1 min



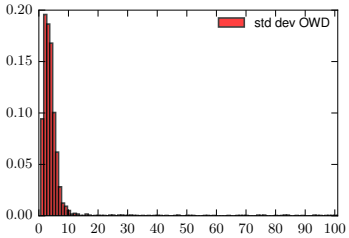
(e) EC2-USW 10 min



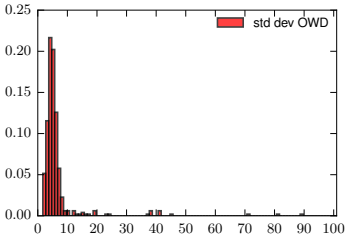
(f) EC2-USW 1 hour



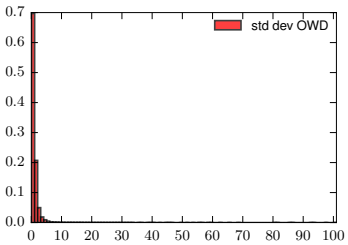
(g) GCE-EUW 1 min



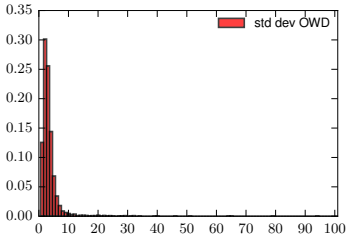
(h) GCE-EUW 10 min



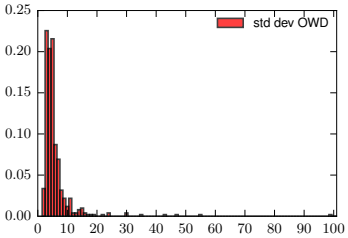
(i) GCE-EUW 1 hour



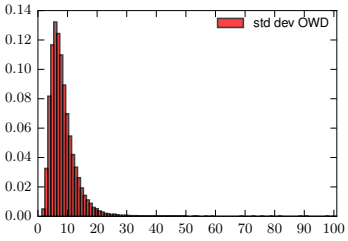
(j) GCE-USW 1 min



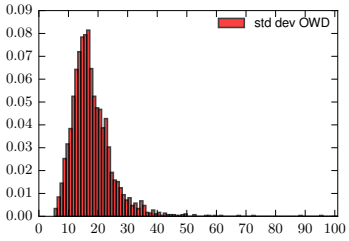
(k) GCE-USW 10 min



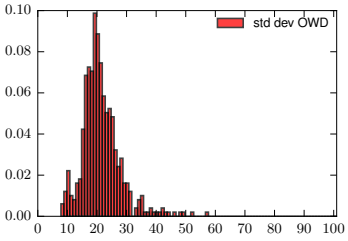
(l) GCE-USW 1 hour



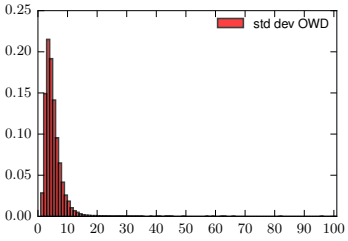
(m) Azure-EUW 1 min



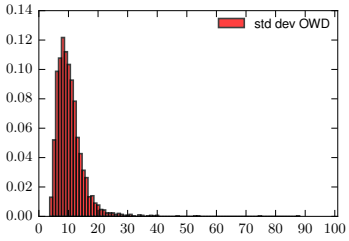
(n) Azure-EUW 10 min



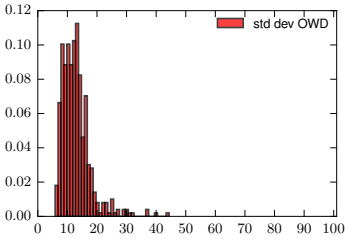
(o) Azure-EUW 1 hour



(p) Azure-USW 1 min

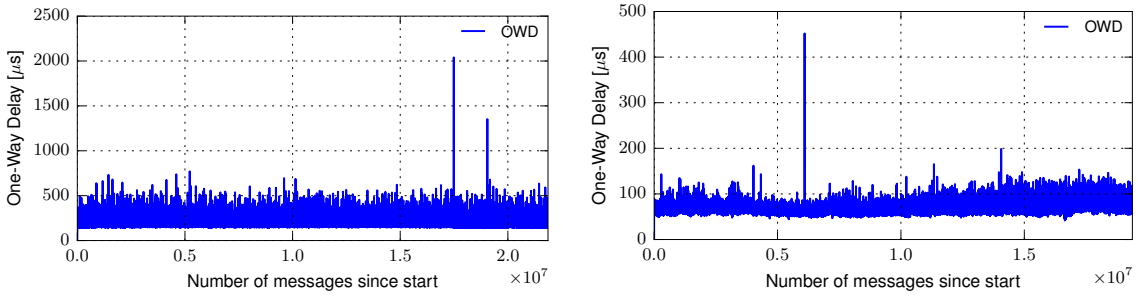


(q) Azure-USW 10 min



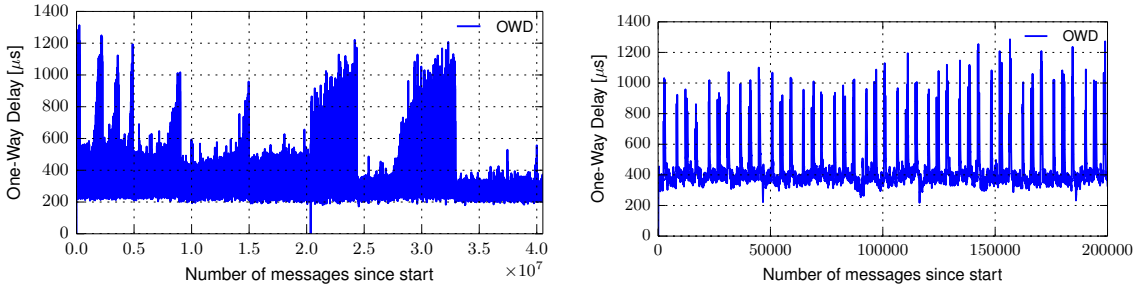
(r) Azure-USW 1 hour

Figure 4.11: Histogram of standard deviation values for OWD computed for different intervals of time (1 minute, 10 minutes, and 1 hour) for different data centres.



(a) VM1-VM2 timeline EC2-USE **(b)** VM1-VM2 timeline GCE-USW

Figure 4.12: Measured OWD between VM1 and VM2 using the high message frequency.



(a) Timeline **(b)** First 15 minutes of timeline

Figure 4.13: Measured OWD between VM1 and VM2 in Azure-UKS using the high message frequency.

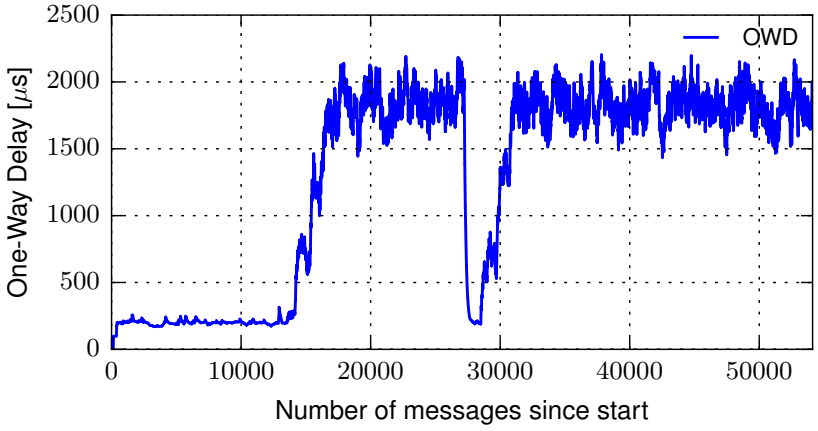


Figure 4.14: Measured OWD between VM1 and VM10 in EC2-USE data centre.

Data center	1 hour					1 day				
	min	*average*	*median*	*max*	*stddev*	*min*	*average*	*median*	*max*	*stddev*
AWS-EUW	0.0	2.028	0.161	11.198	2.511	0.886	1.737	1.679	2.548	0.461
AWS-USW	0.0	1.06	0.0	8.324	1.74	0.116	0.779	0.777	1.617	0.373
GCE-EUW	0.0	2.96	0.0	46.961	7.081	0.109	2.95	0.463	14.082	4.56
GCE-USW	0.0	0.476	0.0	8.373	1.158	0.0	0.154	0.0	0.81	0.256
Azure-UKW	0.0	2.45	2.758	16.533	2.806	1.273	2.405	2.197	3.707	0.633
Azure-USW	0.0	1.244	0.0	11.123	1.9	0.116	0.843	0.753	1.618	0.417

Table 4.4: Packet loss ratio $\times 10^{-4}$ over one week.

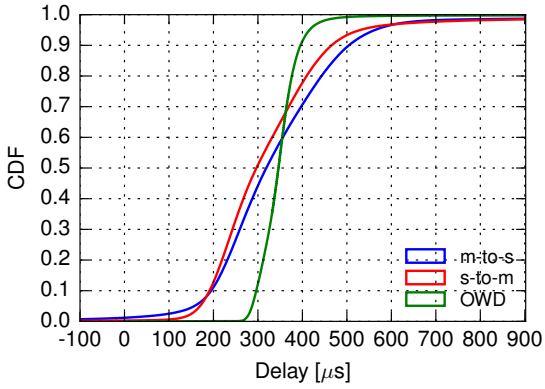
4.6 Packet loss ratio measurements

I investigate packet losses in six data centres over a week for each of the VM pair. I log the number of *Delay Request* and *Delay Response* messages exchanged between the clients and the master in PTPmesh for the measurements with the low message frequency. Using the metric I defined for computing packet loss ratio in Section 3.5, I compute the packet loss ratio over intervals of 1 hour and 1 day over one week. In Table 4.4, I show the minimum, average, median, maximum and standard deviation for all the 1-hour and 1-day intervals across all pairs. Interestingly, all EU data centres have higher packet loss ratios than the US data centres across all cloud providers. Figure 4.9 presents timelines over one week for packet loss ratios computed over 1 hour intervals for one VM pair in EU and US data centres, respectively. The ratios computed depend on the message frequency, but they can serve as baseline for normal conditions, and to determine anomalies when deviating from these baseline values.

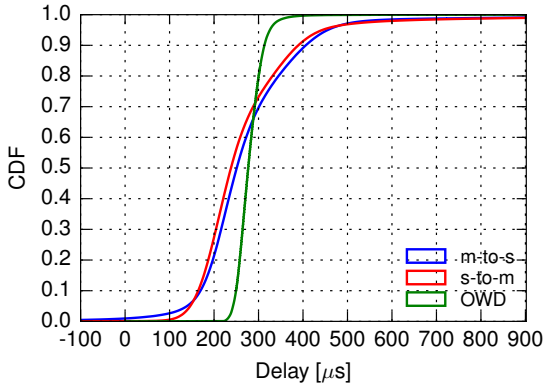
In general, the packet loss ratios have low values for all data centres, with most of the 1-hour intervals having no loss or having 1-4 messages lost per hour (out of 3600), which is at most approximately 11.1×10^{-4}. For AWS EC2, the number of messages lost per hour is at most four (Figure 4.9d and Figure 4.9j), with more losses observed in the EU data centre. High packet loss values of up to 46.96×10^{-4} appear in the first part of the GCE-EUW data centre traces (Figure 4.9e), and significant increases in network latency can be seen in Figure 4.9b, but later in the trace the values are normal, with at most three messages lost per hour. In the GCE-USW data centre (Figure 4.9k), the number of messages lost per hour is at most two, being the data centre with the smallest packet loss ratio. For Azure-EUW (Figure 4.9f), slightly higher packet loss ratios can be observed, while for Azure-USW (Figure 4.9l) the maximum number of messages lost per hour is four.

4.7 Path symmetry

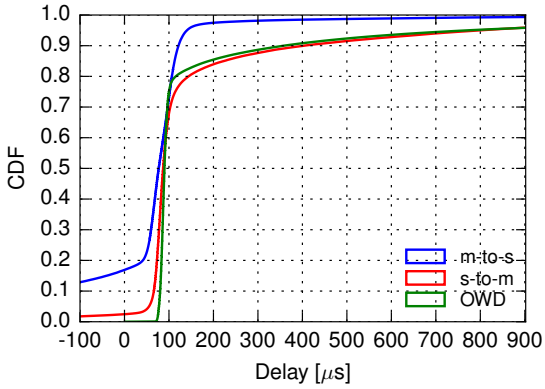
PTPd reports the *master-to-slave* and *slave-to-master* measured delays. These two measurements can be used to determine if the paths from the master to the slave and from the slave to the master are symmetric, but with some caveats, as these two metrics incorporate the offset between the two slave and master clocks and possible congestion effects. I am interested in



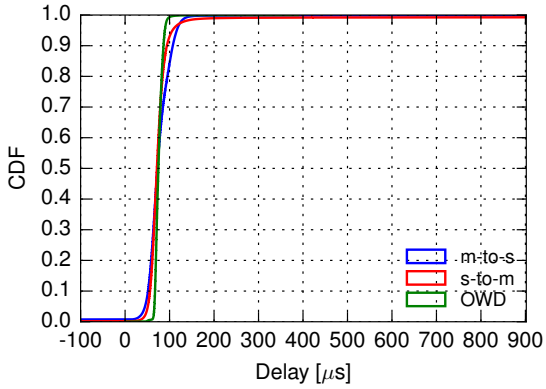
(a) EC2-EUW VM1-VM3



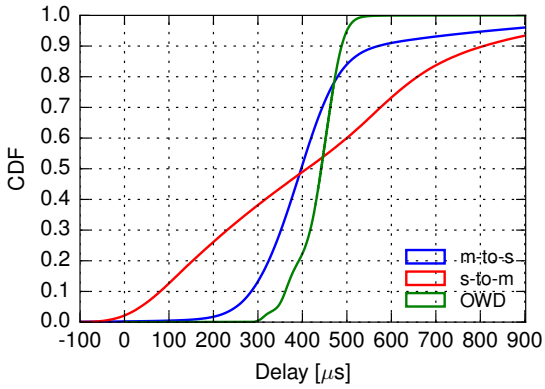
(b) EC2-USW VM1-VM3



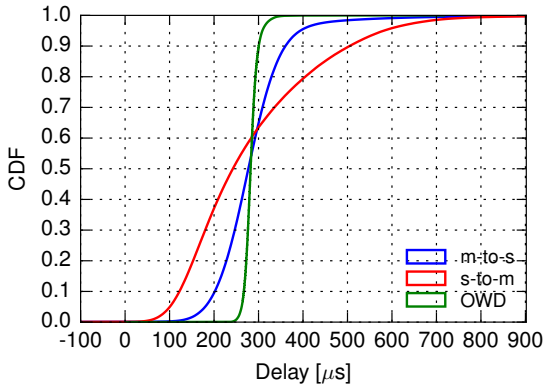
(c) GCE-EUW VM1-VM3



(d) GCE-USW VM1-VM3



(e) Azure-EUW VM1-VM3



(f) Azure-USW VM1-VM3

Figure 4.15: CDF for the master-to-slave (m-to-s) delay, slave-to-master (s-to-m) delay and OWD in different data centres.

determining whether the simplifying assumption that the OWD can be computed as half of the measured RTT holds true in data centres. Note here that the values of the two delays can be negative, due to the differences between the master and slave clocks.

Figure 4.15 shows the master-to-slave and slave-to-master delays for the VM1-VM3 pair in six data centres. The plots for the other pairs and data centres are similar. Based on the collected data, EC2 and GCE forward and reverse paths are symmetrical, while the paths in Azure are

not. Azure data centres (Figures 4.15e and 4.15f) display significant differences between the master-to-slave and slave-to master delay CDFs (different curve shapes), leading to the conclusion that the forward and reverse paths between the VMs are not symmetrical. In the case of the GCE-EUW trace (Figure 4.15c), the long congestion period is reflected in a vertical translation of the master-to-slave and slave-to-master delay CDFs, but this does not mean that the paths are asymmetrical.

4.8 Identifying different network paths within data centres

I patch PTPd in order to be able to specify the port number on which the PTP event and general messages are sent and received. Since the PTP event messages' send and receive timestamps are the ones used in the computation, I change only the port number used for this type of messages. I run PTPd using different port numbers to see if the one-way delay reported by PTPd changes significantly between runs, possibly signalling that a different network path was used due to ECMP hashing on a different header. The results in the Azure-KS data centre collected using 50 different ports with a 10 minutes run for each did not show any indication that could lead to this conclusion. This means either that the network paths of the PTP packets were similar in delay, or that the packets were taking the same path, which is unlikely, given that Pingmesh and NetNORAD use the port number changing strategy to cover as many paths as possible.

4.9 Discussion

PTPmesh offers end-to-end measurements, including the intermediate virtualisation layer. The one-way delay latency values offer insights with respect to end-host overhead, in-network congestion and data centre network architecture. When combining these measurement results in data centres and the virtualisation overhead measurements from Section 3.4 with the network latency contributions percentages presented in Section 2.6, I arrive at the same conclusion as prior research: the end-host, with the hypervisor, is a significant contributor to the overall measured latency from within the VM. The other significant contributor is network queueing at switches. When looking at the network latency contributors in Section 2.6, it can be noticed that switching in the data centre fat tree topology takes up almost 75%, which is approximately 15μs in this analysis, while the rest is taken up by NICs and fibre length, with an estimate of 20μs. This analysis represents baseline contributions. On top of this, the hypervisor's overhead can be added, which based on my measurements from Section 3.4 is 200μs when using a low message frequency, and 190μs when using a high message frequency, giving a median OWD baseline of 220μs and 210μs, respectively. This back-of-the-envelope calculation shows that the remaining latency may come from in-network congestion, traffic bursts from other colocated VMs, transparent VM live migration, when it is observed for short periods, or from sustained

increased network utilisation (whose cause may be bulk network transfers across the data centre, competing traffic from other colocated VMs, cluster drains), when it is observed over longer periods of time. Smaller values than this baseline may mean that the OS is bypassed [DSA+18; FPM+18], or that the VMs may be colocated on the same host, or that there is a shorter network path between VMs.

4.10 Limitations

End-to-end measurements The network latency measurements presented in this chapter represent end-to-end measurements from within the VMs, which include the virtualisation layer. Cloud tenants usually do not have access to advanced features of the underlying hardware, *e.g.*, hardware timestamping. If access were provided, the precision of the network latency measurements would be improved due to the removal of the end-host network stack latency contribution from the measured network latency [RBB+18] (§2.6). On the other hand, end-to-end measurements offer a more accurate value of the latency that the application experiences, encompassing also the end-host network stack latency contribution. Determining the cause of latency spikes (end-host issue or network fault) may be difficult without access to the internal cloud infrastructure, and even then, finding the root cause may still prove tedious [RBB+18].

Spatial analysis The measurement study conducted in this chapter does not cover the spatial analysis of the data centres. However, my results show that, in some cases, small groups of VMs are clustered together, observing the same network conditions. This means that, to measure the data centre network across the core and aggregation switches, one would have to rent a substantial number of VMs to ensure they are not placed within the same rack or on the same machine. On the other hand, this might not hold true for all cloud providers. For example, Microsoft Azure's tenant VMs are placed randomly within a cluster, or they can even be spread across data centres in a region [RBB+18].

Scalability PTPmesh's design implies that $O(n^2)$ measurements are taken for n VMs. This can quickly become a bottleneck both at the end-host (in terms of CPU resource consumption) and in the network (in terms of network bandwidth taken up by the messages exchanged between VMs). Thus, it is important to choose an appropriate message frequency, as discussed in Section 4.3.1. However, even if a tenant has a small number of VMs, as it is often the case [CBM+17], the combined resource usage of PTPmesh across different tenant networks can be a burden to the data centre infrastructure. To mitigate part of this issue, the PTPd master could be consolidated to run in the hypervisor or in the virtual switch, similar to VNET-Pingmesh [RBB+18]. For example, if the hypervisor used is Xen [BDF+03], the PTPd master should run in `Dom0`. This design decision has some tradeoffs. While the CPU and network resources used are smaller than in the current design, the network latency measured does not express the latency experienced by the user applications within a VM [RBB+18] (§2.6).

4.11 Summary

In this chapter, I first discussed different deployment scenarios in the cloud for PTPmesh (§4.1). Next, I gave an overview of the experimental setup and methodology used for collecting the PTP measurement data by PTPmesh (§4.2). Following, I calibrated PTPd in the cloud by performing several experiments to determine the overhead that PTPd has when running in a VM and how the number of concurrent PTPd clients impacts the OWD measurement (§4.3). I then presented the datasets collected (§4.4) and analysed several interesting traits concerning one-way delay measurements (§4.5) and packet loss ratios in ten data centres from three cloud providers (§4.6). I also looked at inferring network path symmetry (§4.7) and at finding different paths between VMs in data centres (§4.8). I presented an analysis of latency contributors of the PTPmesh end-to-end-measurements (§4.9). Finally, I discussed the limitations of PTPmesh (§4.10).

Through my work in Chapter 3 and Chapter 4, I show that PTPmesh provides a majority of the features needed by a data centre network monitoring system (§2.3.4). PTPmesh uses PTP measurements to estimate one-way delay and packet loss ratio. The number of probes sent is configurable, it can provide continuous measurements, and does not have significant overhead. Furthermore, it is easy to deploy within VMs by tenants themselves. PTPmesh can infer network conditions for tenant deployments in the cloud. It can keep track of the latency within the data centre and inter-data centre, and can help in detecting network congestion and packet loss.

Chapter 5

Characterising the network latency impact on cloud-based applications performance

Cloud computing has revolutionised the way businesses use computing infrastructure. Instead of building their own data centres, companies rent computing resources from cloud providers (*e.g.*, Amazon AWS, Google Cloud Platform, or Microsoft Azure), and deploy their applications on cloud provider hardware. Previous work [WN10; BS10; XMN+13; MK15] and measurements presented in Sections 2.4 and Chapter 4 have shown that network latency variability is common in multi-tenant data centres. As even small amounts of delay, in the order of tens of microseconds, may lead to significant drops in application performance (§2.5), there is a need to quantify the impact of network latency on typical cloud-based applications' performance. While past work has provided comprehensive performance studies of the effect of CPU cache, memory, OS or virtualisation upon application performance (*e.g.*, [WN10; ZTH+13; MWH14; XLJ+14]), a significant gap exists in evaluating the impact of networking resources, and in particular network latency, upon application performance.

This chapter first presents an experimental methodology to measure application performance under arbitrary changes in network latency. This involves injecting network latency in the network between hosts. Software-based latency injection tools do not provide the required microsecond-granularity, as explained in Section 2.1.4. Thus, I use a custom hardware appliance, NRG, described in Section 2.1.4, that allows per-packet latency control with a precision on the order of tens of nanoseconds. The methodology allows testing different latency magnitudes, and also different variance magnitudes and distributions. The injected latency may represent delay due to increased cabling length which translates into increased propagation delay, or delay caused by the end-host, or queueing within switches due to network congestion.

In this chapter, I study the impact of network latency on application performance through the methodology previously described for a set of cloud applications: domain name system (DNS) [Moc87], key-value store (Memcached) [Mem18] and machine learning applications running on different frameworks (STRADS [KHL+16], Spark [Spa], Tensorflow [ABC+16]).

I discuss the results of my measurement study on how the performance of these cloud-based applications changes under a range of latency magnitudes chosen based on the values measured in Sections 2.4 and Chapter 4. Further, I model the relationship between network latency and application performance.

5.1 Experimental setup

The methodology enables characterising the effect of network latency on applications' performance using a small scale, controlled, environment. It is based on the observation that each host's experience of the network can be collapsed to the link connecting it to the ToR switch. By modifying the properties of the traffic arriving through this link, the host can experience different latency values, as if it were located in different data centres or different locations within a data centre, or as if it were running during different time periods, all of these being conditions which affect the network latency, as shown in Chapter 3.

In each scenario an application component is selected to be run on a host in the setup from Figure 5.1. The application component can be the server (DNS, Memcached), the master (STRADS, Spark, Tensorflow) or a client (Memcached). Between the *selected application component's host* and the other hosts of the setup, I use the NRG appliance to inject a controlled latency value into the system. NRG abstracts the network topology as a single queue, represented by the delays injected in the network. From the selected host's perspective, this queue introduces delay through the link that connects the server to the remaining network. The injected network latency encompasses the different sources of latencies that are present in a networked system: *static* latencies, *e.g.*, propagation delay, or *variable* latencies, *e.g.*, end-host delay, and queueing in switches. The measurements I perform in this chapter use a latency injection model that presumes a constant latency between the server/master and client/worker pairs. In this work I do not consider the impact on performance of variable latency amongst the client-server/worker-master pairings.

The experimental setup in Figure 5.1 is composed of 10 hosts, but the methodology does not depend on this number of hosts. Each host has an Intel Xeon E5-2430L v2 Ivy Bridge CPU with six cores, running at 2.4GHz with 64GB RAM. To ensure experimental reproducibility and reduce variance, CPU-power saving, hyper-threading and frequency scaling features are disabled. The hosts run Ubuntu Server 16.04, kernel version 4.4.0-75-generic. Each host is equipped with an Intel X520 NIC with two SFP+ ports, and is connected at 10Gbps using 2m long Direct-Attach copper cables through an Arista 7050Q switch.

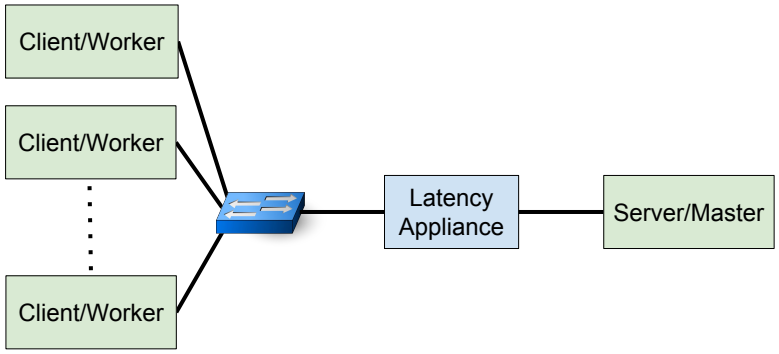


Figure 5.1: Experimental setup to evaluate application performance under changing network latency.

5.2 Selected cloud-based applications

I select five applications whose performance can be analysed on my setup (see Section 5.1). The choice of applications is intended to explore different distributed operating models (server-client and master-worker). The selection does not represent all common data centre applications, and most of the chosen applications are network intensive.

5.2.1 Domain Name System (DNS)

This is the simplest application studied, while being widely used in the cloud. It provides a domain name lookup service. For the server, I use NSD (Name Server Daemon) [Lab18], which is an open source name server, authoritative only. DNSPerf [Nom18] (version 2.1.0.0) is used on the client side to generate requests. For the application performance metric I use the number of requests per second that the name server can achieve. This number is dependent on how the server and client are implemented: if pipelining is used, then the number of requests per second will not be dramatically affected by the injected network latency, but the query latency will be. DNS follows a client-server model, and I focus on the effect of network latency on performance as observed by the server and the client.

5.2.2 Key-value store: Memcached

Memcached [Mem18] is a widely used, in-memory, key-value store for arbitrary data. Clients can access the data stored in a Memcached server remotely over the network. Memcached offers an interface that resembles that of a hash table: the most widely-used operations are insertion (SET command) and retrieval (GET command). Memcached then uses a Least-Recently-Used (LRU) policy to evict items when running out of server memory. I use the open-source version of the Memcached server 1.4.25.

I use the Mutilate [Lev14] Memcached load generator in my evaluation of the impact of network latency on Memcached's application performance, measured in queries per second (QPS). The workload generator is based on a closed system model [SWH06]; that is, each workload generator waits for a reply before sending the next request. I use two workloads generated by the Mutilate benchmark: i) a *read-only workload*: the requests follow an exponential distribution, the key size is 30 byte, and the value size is 200 bytes; the keys are accessed uniformly [LK14]; ii) the *Facebook "ETC" workload*, taken from [AXF+12], which is considered representative of general-purpose key-value stores; the ratio SET:GET is 1:30 ratio. The configuration parameters used in my experiments match the ones in [AXF+12]. I do not use *multi-GET* requests. I do not use the pipelining option for requests, in order not to introduce delays at the server or in the network. If pipelining were used, then the number of queries per second would be less affected by the additional network latency. Memcached follows a client-server model, and I focus on the effect of network latency on performance as observed by the server and the client pool.

5.2.3 Machine Learning applications

STRADS Lasso Regression STRADS [KHL+16; KHL+18] is a distributed framework for machine learning algorithms targeted to moderate cluster sizes between 1 and 100 machines. The STRADS framework uses one coordinator, one scheduler and requires at least two workers. In my setup, the coordinator and the scheduler run on the master machine from Figure 5.1. I evaluate the impact of network latency on the sparse Lasso (least absolute shrinkage and selection operator) Regression [Tib11] application implemented in this framework. The Lasso Regression application works in the following way. Firstly, each worker receives its data partition at the beginning of the run. Then, in each iteration, the scheduler computes the set of parameters that each worker will operate on during that iteration, using their data partition; at the end of an iteration, the scheduler aggregates the results from the workers. The network communication pattern can be represented as a star, with a central coordinator and scheduler on the master server, while workers communicate only with this master server.

The application performance metric is the objective function value (convergence metric) versus time (seconds), also referred to as *convergence time* [KHL+16]. I do not use pipelining, which means executing an iteration of the algorithm with some or all of the parameters stale (from previous iterations). While pipelining reduces the impact of network latency by overlapping network communication with computation, due to the usage of stale parameters and the potential for dependencies between iterations, it may lead to a slower convergence rate [KHL+16]. Since I do not allow the use of stale parameters and the injected network latency does not change the scheduling of the parameters (not influencing the objective function value), the application performance metric can be represented by the job completion time, named in the next sections the *training time*.

The input to the application is represented by a N-by-M matrix and a N-by-1 observation vector, while the model parameters are represented by a M-by-1 coefficient vector. I use a synthetic

workload that I generated, with a number of 10K samples and 100K features, and a total of 500M non-zero values in the matrix, the input data size being 9.5GB.

Spark GLM Regression I use Apache Spark [Spa]'s machine learning library (MLlib), on top of which I run benchmarks from Spark-Perf [Dat18]. I run Spark 1.6.3 in standalone mode. Spark follows a master-worker model. Spark supports broadcast and shuffle, which means that the workers do not communicate only with the master, but also between themselves. I use as application performance metric the *training time*, *e.g.*, the time taken to train a model.

Tensorflow MNIST Tensorflow [ABC+16] is a widely used machine learning framework. I use the MNIST dataset [LC10] for the handwriting recognition task as input data, and Softmax Regression for the training of the model. Tensorflow follows a master-worker model. The application performance metric used is the *training time*, similarly to Spark's performance metric. I use the `synchronise replicas` option, which means that the parameter updates from workers are aggregated before being applied in order to avoid stale gradients. This option is similar to not using the pipeling option in STRADS.

5.2.4 Other applications

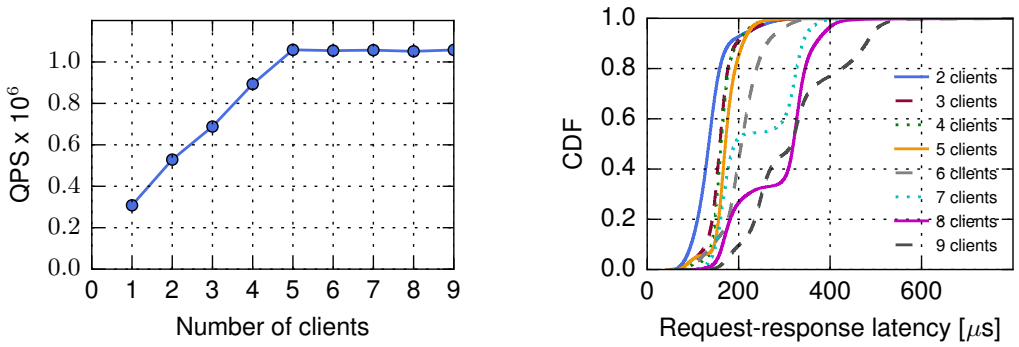
Besides the four applications described previously, I also explore the effect of latency on the performance of other applications. In Section 2.5, I studied the effect of static latency on the Apache Benchmark [Proa] for a single client-server pair. However, for this use case, Apache easily saturates the link, making network bandwidth the bottleneck of the system. In this scenario, even two clients are competing for network resources, thus the study of the latency effect is contaminated by other network effects. I also studied the effect of latency on the TPC- C MySQL benchmark [Lab17] for a MySQL database reporting New-Order transactions per minute as the performance metric (where New-Order is one of the database's tables for the benchmark). Further, I also explore a set of applications for the Spark framework, including *e.g.*, K-Means, Gaussian Mixture Models. I omit those for brevity, as they behave similarly to the GLM Regression application.

5.3 Baseline application performance

Table 5.1 describes the settings of each application. I run each application a sufficient number of times for reproducibility for each latency configuration. I first determine the baseline performance of each application, which is the maximum achievable performance for the selected host. The baseline setup achieves this performance using a minimum number of hosts. The selected host's resources (acting as a server, worker or client) should be saturated, but without overloading the host. There is no other network traffic in the setup.

Application	Host's role	#Hosts	Performance Metric	Runtime Target	Dataset	Dataset Size
DNS [Moc87]	Server	1	Queries/sec	10M requests	*A record*	10M requests
Memcached [Mem18]	Server	5	Queries/sec	10 seconds	FB ETC [AXF+12]	see [AXF+12]
Memcached [Mem18]	Client	1	Queries/sec	10 seconds	FB ETC [AXF+12]	see [AXF+12]
STRADS Regression [KHL+16]	Coordinator	6	Training time	100K iterations	Synthetic	10K samples, 100K features
Spark Regression [Spa]	Master	8	Training time	100 iterations	Spark-perf [1] generator	100K samples, 10K features
Tensorflow [ABC+16]	Master	9	Training time	20K iterations	MNIST [2]	60K examples

Table 5.1: Workloads Setup. #Hosts indicates the minimum number of hosts required to saturate the selected host for which I measure the application performance, or the number of hosts for which I determine the best training time when no latency is added.



(a) QPS (b) Request-response latency

Figure 5.2: Baseline analysis to determine the maximum QPS that can be achieved by the Memcached server.

For example, in the case of a client-server application, where the peak performance of the server is N queries per second, the minimum number of clients required to achieve this performance is k. Any number of clients above k can still achieve N queries, but not more. What I seek is to understand the effect of network latency on the server using k clients, where any loss of performance will be due to latency rather than end-host processing or storage. If I had chosen a number of clients greater than k, the host could have maintained a performance of N queries per second, but the sensitivity to the network would not have been exposed.

For each of the workloads, I conduct multiple experiments required to determine the aforementioned baseline. An example of such baseline analysis is provided in Figure 5.2 for Memcached using the read-only workload. The results are similar for the Facebook "ETC" workload. I vary the number of client machines from 1 to 9, and one of these clients, called the master client, also takes QPS and request-response latency measurements. In this configuration, Memcached achieves a maximum of approximately 1.05M QPS (Figure 5.2(a)) using five clients machines (180 connections total). I use 6 threads and 6 connections per host, thus one client machine creates 36 connections. Increasing further the number of threads or connections per host does

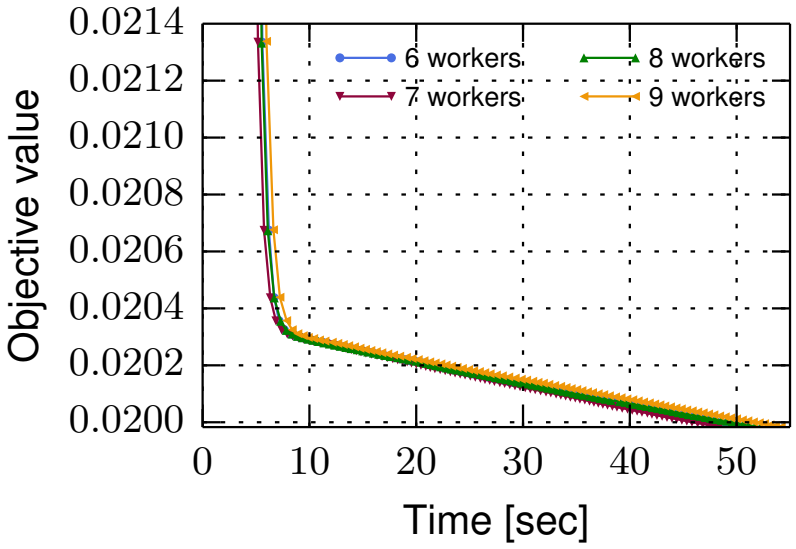


Figure 5.3: Baseline analysis to determine how many worker machines are needed to complete the STRADS Lasso Regression training in minimal time.

not yield any increase in the achieved QPS. From Figure 5.2(a), it can be observed that, as the maximum compute power of the Memcached server is reached, increasing the number of client machines beyond five does not increase the maximum throughput achieved by the system. However, using more than five client machines leads to an increase in the request-response latency per client due to queueing delay at the server (Figure 5.2(b)). This happens because when the server is fully loaded, the `epoll_wait` function used by the Memcached server code returns hundreds of file descriptors, which are processed sequentially, taking a substantial time to process any request [LK14]. This also leads to delays for the new requests that have arrived in the meantime [LK14]. Another cause is L3 cache interference when the server is fully loaded [LK14]. Based on these results, I select the setup that yields maximum performance under minimal request-response latency: five client machines to generate load, including a client machine that takes QPS and latency measurements, and a sixth machine for the selected host acting as a server.

Similarly, when having a master-worker application, I determine the minimum number of workers needed to minimise the training time. The STRADS framework uses one coordinator, one scheduler, and requires at least two workers. The coordinator and the scheduler run on the master server (in Figure 5.1), while the other 9 machines act as workers, each worker uses 6 threads, and the scheduler uses 6 threads as well. Under the given configuration, it is not possible to use less than 6 workers. I vary the number of available workers from 6 to 9 to determine the best configuration. The application runs for $100,000$ iterations. Figure 5.3 shows the objective value (metric for convergence for the ML application model) versus job completion time for various number of workers. Since there is no substantial decrease in execution time, nor a substantial difference in the achieved objective value, I choose to run the experiments using 6 workers. I also explored setting different numbers of threads for the scheduler and workers, however, the

described configuration is the one that offers the best performance on the test setup.

5.4 The effect of static latency on application performance

Static latency most often represents latency due to the distance between two machines, mostly the propagation delay on the fibre, but it may also represent other fixed delays, such as the inherent delay within network apparatus, *e.g.*, NICs, switches. This type of delay can be assimilated mostly to the median latency values within data centres, while the higher percentile values refer to the latency variance. The injection of constant latency can thus be translated to a placement problem, *e.g.*, what is the maximal distance between processing nodes that will not affect the performance of an application. As every $100m$ of fibre is equivalent to $\approx 1\mu$s of RTT, application performance could be improved through the design of different network topologies that reduce propagation delays, or through better cluster scheduling to support application placement constraints related to network latency demands (Chapter 6).

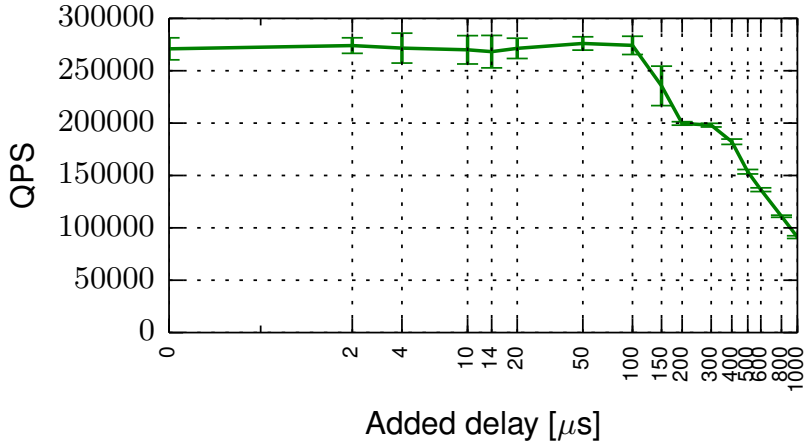
After the baseline performance is determined for each application, I introduce a constant latency value between the selected host and the other hosts using NRG in both directions, client to server (request) and server to client (response), sweeping the range of values between 1μs and 500μs. Thus, the total latency values introduced range between 2μs and 1000μs. These values are in addition to the baseline latency of the networked system. I chose values in this range based on the network latency values that I measured in different cloud providers (§2.4, §4.5). The application performance is measured for each injected latency value. This methodology can be deployed in the public clouds as well, either through using a software-based emulator such as NetEm (§2.1.4), or by leveraging the FPGAs deployed in the data centres by the different cloud providers, for example the Catapult project [CCP+16] from Microsoft Azure or Amazon EC2 F1 instances [3].

I first discuss the results for each application in turn, and then I compare the applications' response to injected latency in terms of performance.

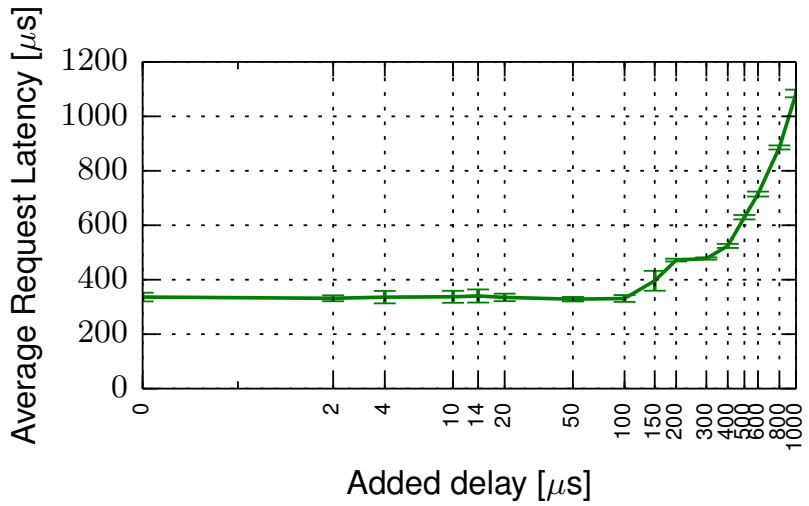
Domain Name System (DNS) The results for DNS are presented in Figure 5.4. The baseline performance obtained on my setup is approximately $270,000$ QPS. Injected latencies of up to 100μs do not affect application performance. Inserting higher latency values leads to drops in performance: 150μs brings 13%performance loss, while 200μs and 300μs lead to 26% performance loss compared to the baseline. Injecting 500μs leads to 44% performance loss, and 1ms leads to 66% performance loss, reaching a QPS of around $91,000$.

Figure 5.4b shows the average query latency for each latency value injected in the setup. The query latency increases as the QPS decreases. The baseline average latency is 344μs, while the average latency when injecting 1ms is 1.1ms. It is interesting to note that the request latency

[3] `https://aws.amazon.com/ec2/instance-types/f1/`



(a) Queries per second

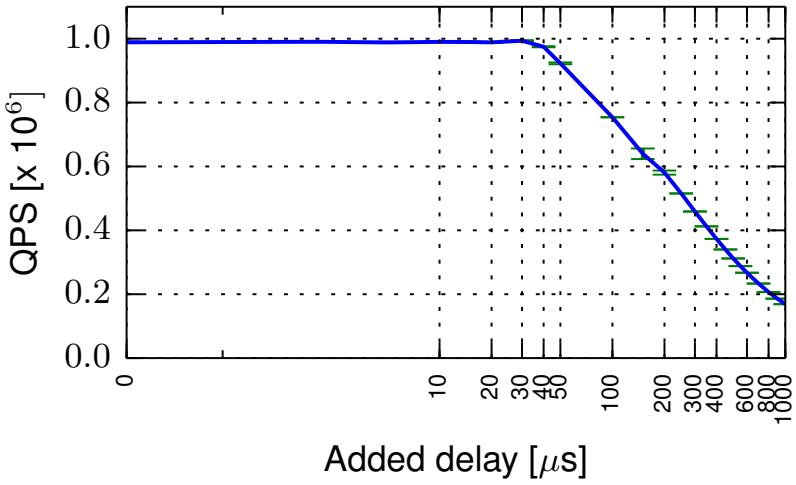


(b) Average query latency

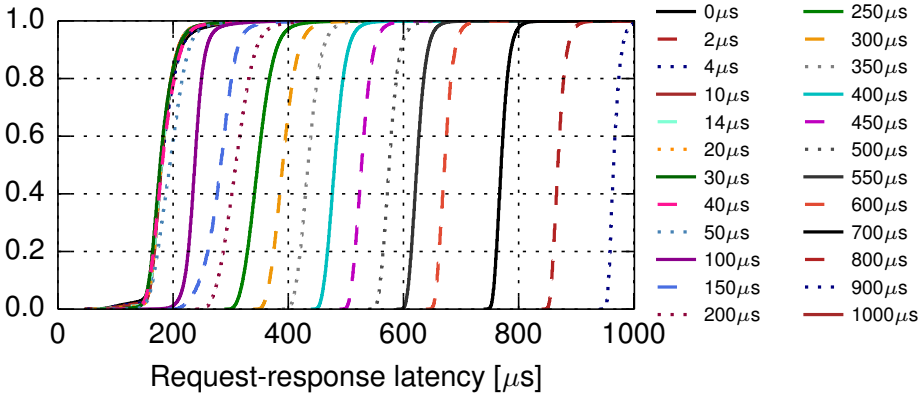
Figure 5.4: DNS QPS and average query latency for static latency injection.

does not increase by the exact amount of latency injected in the network. This is related to the interplay between architectural components (CPU, cache, memory) and OS operations [ZGP+17], masking part of the effects of injected latency.

Key-value store: Memcached In Figure 5.5a, a slight drop from a baseline of 990K QPS for the Facebook "ETC" workload appears with the addition of 40μs, and then a further 6.6% performance drop from the baseline when adding 50μs. Adding a total of 250μs reduces the QPS value to nearly half of the baseline. When adding 1ms, Memcached achieves only 17% of the baseline performance. The baseline request-response latency is at the median 179μs, with additional delay of 500μs it is 575μs at the median, and with 1ms it is 1064μs at the median (Figure 5.5b). Similar to the DNS case, the request-response latency does not increase with the exact amount of latency injected, with the interplay between architectural components (CPU, cache, memory) and OS operations [ZGP+17] masking part of the effects of injected latency.



(a) Queries per second



(b) CDF of request-response latencies

Figure 5.5: Memcached QPS and request-response latency for the Facebook "ETC" workload for static latency injection.

The read-only workload yields very similar results. Memcached's performance drops slightly starting with an injected latency of 20μs. Adding 30μs lowers the throughput by approximately 35K QPS from the 1.05M QPS baseline, while 50μs of additional delay reduces the QPS achieved by more than 100K QPS compared to the baseline performance. Larger injected network latency values lead to a further drop in performance: 250μs of additional latency makes Memcached achieve approximately half of the baseline, while with 1ms added latency it achieves only 16% of the baseline performance.

It is important to note the small amount of network latency that impacts application performance, between 20μs and 40μs, while significant throughput drops appear from 50μs additional delay.

STRADS Lasso Regression The results are presented in Figure 5.6. Injecting only 20μs of latency increases the training time by approximately 2s from a baseline of 50s, while injecting

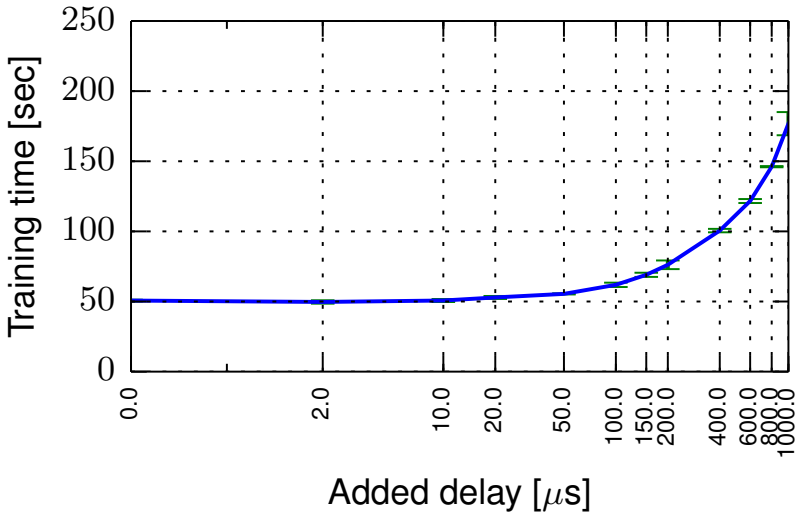


Figure 5.6: STRADS Lasso Regression training time for static latency injection.

100μs increases the training time by 10s, 400μs doubles the training time, reaching approximately 100s. Further, 1ms of network latency leads to a training time of 172s.

Spark GLM Regression In the first set of experiments, the latency is injected only between the master and the workers. Network latencies of up to 200μs do not have an impact on application performance, as observed from Figure 5.7. An additional latency of 500μs leads to a small increase in the training time of 4.4%, while 1 ms results in an increase of 9%, to a runtime of 18.86s compared to the baseline of approximately 17.14s. My results are in line with previous work [ORR+15], which showed that network resources do not have a significant impact on Spark's performance. This is due to a bottleneck in serialisation and deserialisation in the Spark framework that leads to under-utilisation of the network. I also experiment with different numbers of input examples (100,000, 1 million) and iterations (20, 100) for the application, however the behaviour is similar.

Unlike in a parameter server design, workers in Spark also communicate with each other. Thus, I conduct a second set of experiments where I introduce delay also between the workers, not only between the master and the workers. To avoid the complexity of having a NetFPGA SUME card [ZAC+14] installed at each server to deploy NRG, I instead introduce latency with NetEm at each host's incoming and outgoing interfaces. As described in Section 2.1.4, this approach has drawbacks over the hardware-based one. Even so, the results obtained are similar to the experiment where latency was introduced only between the master and the workers: the application performance is not affected for injected latency values of below 500μs RTT.

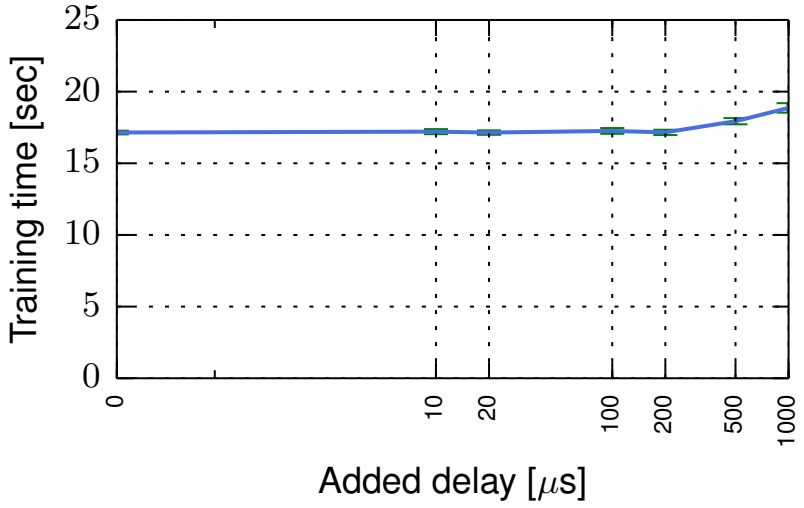


Figure 5.7: Spark GLM Regression training time for static latency injection.

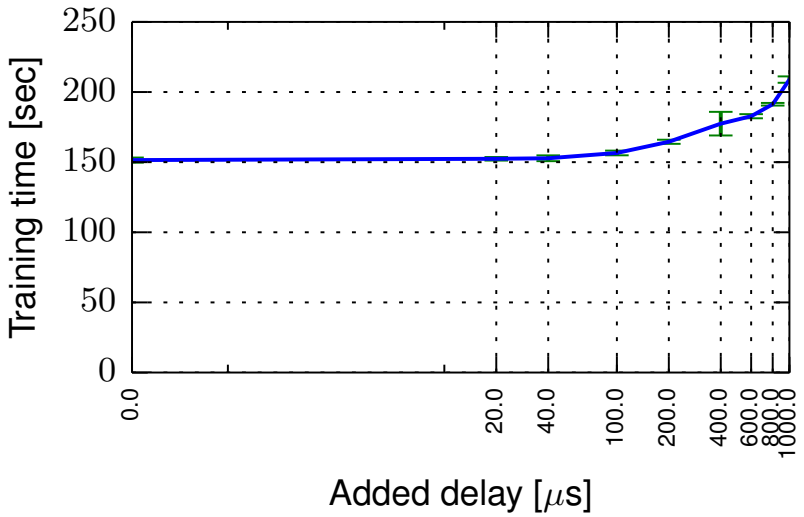


Figure 5.8: Tensorflow handwritten digit recognition training time for static latency injection.

Tensorflow MNIST Figure 5.8 shows the results for Tensorflow handwriting recognition task [4]. The baseline training time is approximately 151s. As little as 40μs affect slightly the application performance. A performance degradation of 3.3% can be noticed for 100μs additional delay. With 600μs additional delay the training time reaches 182s, and 1ms leads to a drop in performance of 27.5%, reaching 208s training time.

[4] `https://github.com/tensorflow/tensorflow/tree/master/tensorflow/examples/tutorials/mnist`

5.4.1 Understanding the effect of static latency on application performance

It is important to understand why applications react differently to network latency. The nature of the application and its purpose define how latency-sensitive the application is. Furthermore, online analysis frameworks, such as SnailTrail [HLL+18], can help to determine for a distributed application the amount of time spent on computation, serialisation, deserialisation or communication between workers, and, more importantly, if or how these periods of time change because of increased network latency.

Latency-sensitive distributed applications are usually synchronous, meaning that the application blocks on waiting to receive data from a different host over the network before proceeding with the next step. This is the case for the different machine learning frameworks studied in this chapter, such as STRADS or Tensorflow. They follow the parameter server design, with workers that exchange messages with the parameter server over the network. These frameworks are generally synchronous: they use, in the current iteration, the parameters computed during the previous iteration. This pattern makes them highly dependent on the network, and especially on network latency. On the other hand, if a degree of parameter staleness is tolerated, the impact of network latency can be mitigated through the use of paradigms like eager stale synchronous parallel (ESSP) [XHD+15], or even asynchronous communication.

Key-value stores, like Memcached, are generally very latency-sensitive. This type of application serves as an intermediate caching layer between the client and the storage system, meaning that the request-response latency is very important, since the store needs to provide fast access to the data. The overall throughput of the Memcached server will decrease when additional latency is injected in the network. To keep the throughput constant on the server side, more requests can be issued by the clients through pipelining (or more clients can be deployed in the system), but this does not change the fact that the additional latency increases the latencies of the requests issued by the clients.

Another aspect defining how latency-sensitive the applications are is how well they are written. The applications should be written to take advantage of all resources. In the case of Spark, small amounts of network latency do not matter, since it has a bottleneck in the serialisation and deserialisation of data, which leads to under-utilisation of the network [ORR+15].

Applications that are throughput intensive, such as Hadoop MapReduce [DG04], are not latency-sensitive, since they are not sensitive to per-packet delivery times [AKE+12]. I conducted experiments with Hadoop MapReduce on the same testbed, and using an application that performs natural join [GSG+15] of two datasets. I found that additional latencies up to 1ms do not increase the job completion time for this application.

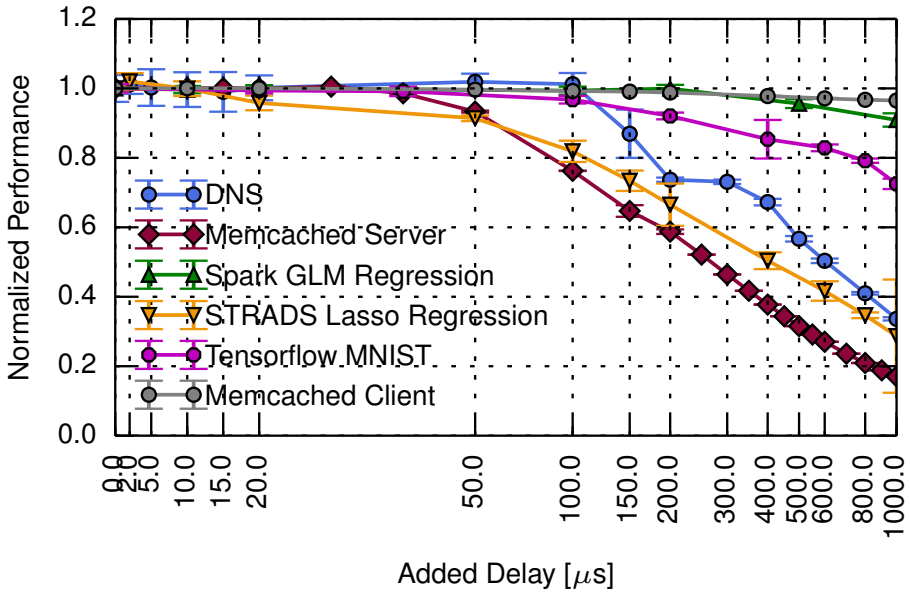


Figure 5.9: The effect of injected static latency on typical cloud applications' performance.

5.4.2 Comparison between applications' performance in relation to network latency

To compare the effect of static latency on the applications, I normalise the performance of each application with respect to its baseline performance by dividing each performance value to the baseline performance value. In the case of Memcached and DNS, the performance curve is descending when injecting more latency, while for Spark, STRADS and Tensorflow the curve is ascending when injecting more latency. To compute the percentage by which the performance decreases when the amount of injected latency increases, I treat the performance values and baseline performance value as inverse proportional values. To have a descending curve for all the applications, in the second case I divide the baseline performance value to each performance value. The baseline performance is marked as 1, and the ratio between the measured performance at each latency point and the baseline performance is shown in Figure 5.9. The x-axis is the static latency added (in μs, for total latency injected in both directions), while the y-axis is the normalised performance. It must be noted that each application has a different performance metric, as described in Table 5.1.

The results show that all applications are sensitive to latency, though on very different scales. For STRADS Lasso Regression, performance degradation can be observed when as little as 20μs are added to the RTT between hosts, while GLM Regression on Spark is not affected by less than 500μs RTT. The Memcached sever is also very sensitive to latency: 100μs are enough for over 20% performance degradation, and the performance is halved for 250μs. DNS is also affected by latency at the scale of hundreds of microseconds. Tensorflow MNIST is also latency sensitive, although to a smaller degree when compared to DNS, Memcached and

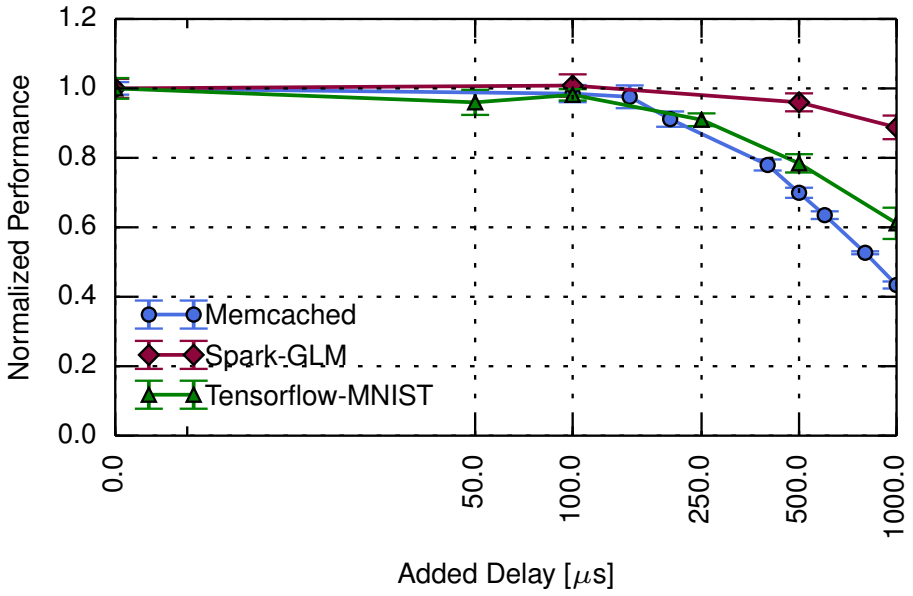


Figure 5.10: The effect of injected static latency on typical cloud applications' performance running on cloud hardware.

STRADS Lasso Regression. On the other hand, Spark is not dramatically affected by network latency, meaning that Spark applications have more choices for their physical placement in a data centre, being less influenced by the network conditions.

For completeness, I also study a different scenario, where I inject static latency between only one Memcached client and the Memcached server. Figure 5.9 shows that in this situation there is a small reduction in the overall aggregated performance of the five clients. As it can be seen from Figure 5.2, for five clients, the first client contributes more than one fifth of the total QPS, while the rest of the four clients contribute less than a fifth of the total QPS. This explains why the reduction in the achieved performance is smaller than a fifth when one client's requests are delayed, considering that the client still sends requests contributing to the overall performance, even if these requests are delayed.

This sensitivity to latency demonstrates two orders of magnitude difference between applications. As previous work has shown 10μs to be the scale of latency between two hosts connected back-to-back (§2.6) [ZGP+17; ERW+14], it follows that DNS, Memcached, and STRADS Lasso Regression are very sensitive to their physical host allocation. These applications need a high degree of network locality. Thus, the components of these distributed applications should be ideally placed close to each other, preferably within the same rack.

While the impact of latency on performance evaluated in Section 5.4 is conducted on a specific setup, the results offer an intuition on the application behaviour that can be generalised to other setups and scenarios as well. Even though there are differences between computing platforms, the results have the same scale and follow the same trends. I exemplify this statement by evaluating three of the selected applications (Memcached, Spark GLM Regression and Tensorflow

MNIST) on a different setup in a data centre in Microsoft Azure. The setup has one server/master VM and five clients/workers VMs. The VM type is Standard E16s v3 with 16 virtual CPUs and 128 GB memory. I insert network latency with NetEm at every host. Given that NetEm is not suitable to inject small latencies of tens of microseconds, as described in Section 2.1.4, I use only larger latencies of over 100μs. The general trend in Figure 5.10 is the same as in Figure 5.9 for the selected applications. In the case of Spark GLM Regression and Tensorflow MNIST, the drop in performance on this setup is steeper than on my local testbed. On the other hand, the Memcached server is less affected on this setup compared with the local testbed. These differences can be the result of any or all of the following factors: virtualisation, different number of hosts, host specifications, different network topology, varying network utilisation due to the shared network in the cloud, latency injection through software emulation instead of through a hardware-based solution. To cover more scenarios resulted from the interplay between all these factors, a system that benchmarks the application performance under different configurations and network conditions could be developed.

5.5 Functions that predict application performance based upon network latency

The relationship between network latency and application performance can help cloud customers to determine the performance their application can achieve under certain network conditions and can guide cloud operators in selecting the network latency ranges that best suit the needs of their customers. By measuring dynamically the network latency in the data centre and having a model of the application performance dependent upon network latency, the expected application performance under the measured network conditions can be determined. Using the normalised performance curves built in Section 5.4.2, I construct a *function that predicts application performance dependent upon network latency* for each application. To model the relationship between network latency and application performance, I use SciPy[5]'s `curve_fit` function, which uses non-linear least squares to fit a function p to the experimental data. The `curve_fit` returns optimal values for the parameters, so that the sum of the squared error of $p(x\_data, parameters) - y\_data$ is minimised. The function has one independent variable, the static latency, and the dependent variable is the application's performance metric. I additionally use the standard deviation of the results as a parameter for the `curve_fit` function.

The relationship between network latency and application performance for Memcached and DNS can be constructed as

$$QPS = p(static\_latency) \tag{5.1}$$

For the other applications (STRADS, Spark, Tensorflow), the training time is the application performance metric, thus the relationship between network latency and application performance

[5] `https://www.scipy.org/`

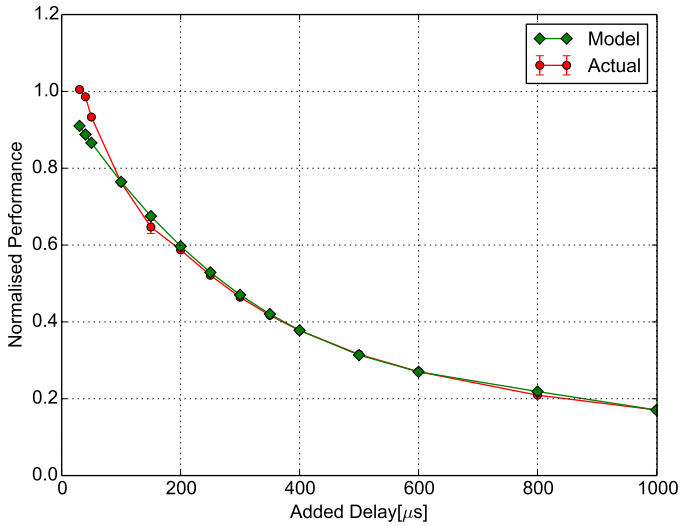


Figure 5.11: Polynomial function fitted to Memcached experimental data.

is:

$$Training\_time = p(static\_latency) \tag{5.2}$$

However, I model the relationship between network latency and normalised application performance:

$$Normalised\_performance = p(static\_latency) \tag{5.3}$$

Memcached I fit a polynomial function based on the results shown in Figure 5.9, where the independent variable is the static latency, and the dependent variable is the application performance. The resulting model is shown in Figure 5.11 and in Equation 5.4. This model does not capture the baseline performance, nor small static latency values. Therefore, the model needs to have two functions: a constant function, whose value is the baseline performance, and a polynomial function fit on the experimental data. The first function gives the performance up to the threshold latency value beyond which the application performance starts to drop, *e.g.*, 40μs. The Figure 5.11 does not show the first function.

$$p(x) = \begin{cases} 1, x < 40 \\ 1.067 - 3.093 \times 10^{-3} \times x + 4.084 \times 10^{-6} \times x^2 - 1.898 \times 10^{-9} \times x^3, x \geq 40 \end{cases} \tag{5.4}$$

STRADS Lasso Regression I fit a polynomial function to the results shown in Figure 5.9, where the independent variable is the static latency, and the dependent variable is the normalised performance. The results are shown in Figure 5.12 and in Equation 5.5. The first function is the constant baseline performance up to 20μs.

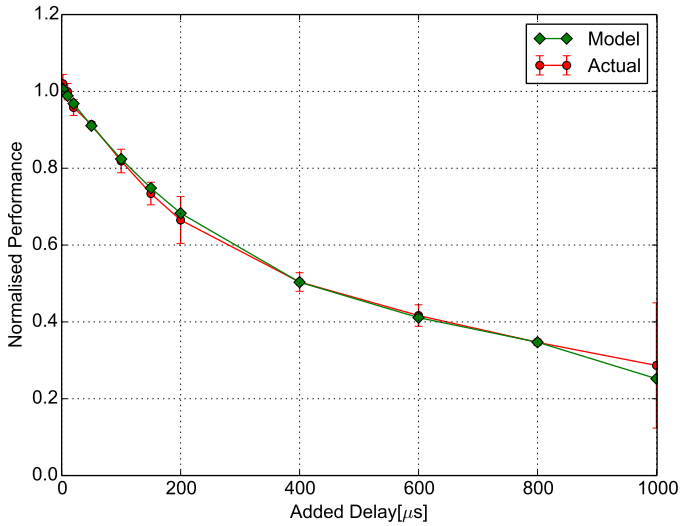


Figure 5.12: Polynomial function fitted to STRADS Lasso Regression experimental data.

$$p(x) = \begin{cases} 1, x < 20 \\ 1.009 - 2.095 \times 10^{-3} \times x + 2.571 \times 10^{-6} \times x^2 - 1.232 \times 10^{-9} \times x^3, x \geq 20 \end{cases} \tag{5.5}$$

Spark GLM I fit a linear function to the results shown in Figure 5.9, where the independent variable is the static latency, and the dependent variable is the normalised performance. The results are shown in Figure 5.13 and in Equation 5.6. The first function is the constant baseline performance up to 200μs.

$$p(x) = \begin{cases} 1, x < 200 \\ -1.161 \times 10^{-4} \times x + 1.0199, x \geq 200 \end{cases} \tag{5.6}$$

Tensorflow MNIST I fit a polynomial function to the results shown in Figure 5.9, where the independent variable is the static latency, and the dependent variable is the normalised performance. The results are shown in Figure 5.14 and in Equation 5.7. The first function is the constant baseline performance up to 40μs.

$$p(x) = \begin{cases} 1, x < 40 \\ 1.005 - 5.146 \times 10^{-4} \times x + 5.837 \times 10^{-7} \times x^2 - 3.46 \times 10^{-10} \times x^3, x \geq 40 \end{cases} \tag{5.7}$$

Finding a general relationship between an application's performance and network latency is not

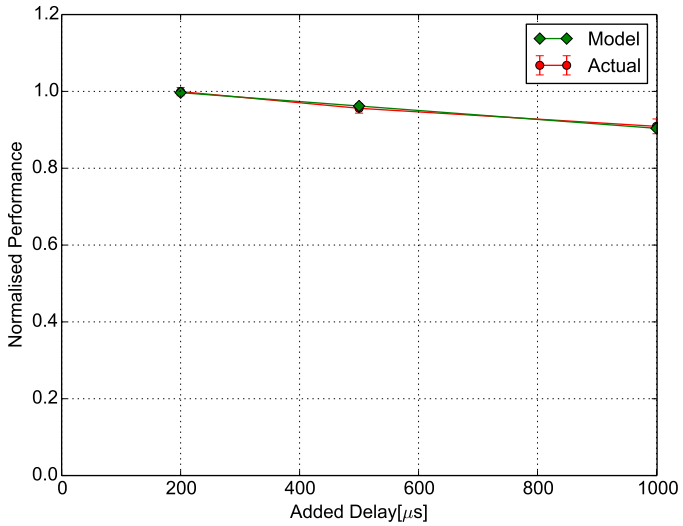


Figure 5.13: Polynomial function fitted to Spark GLM experimental data.

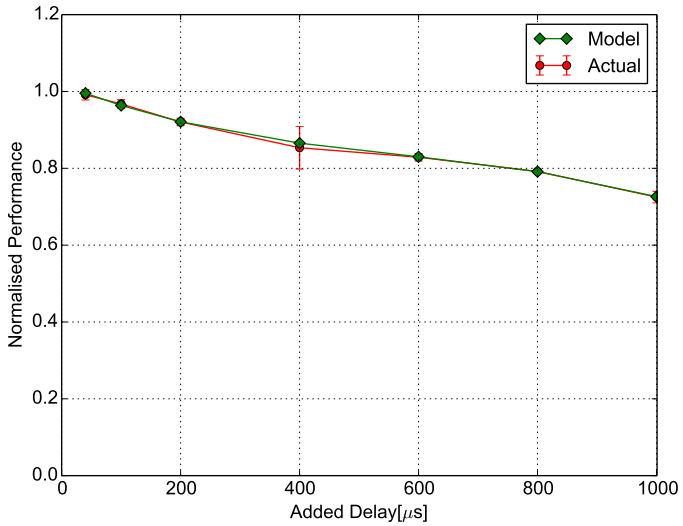


Figure 5.14: Polynomial function fitted to Tensorflow handwritten digit recognition experimental data.

easy. I sought to limit the influence of other factors (OS impact on application, number of cores, number of machines in the setup) on the measured application performance, leaving only the effect of network latency on the application performance.

5.6 Summary

The rapid increase in cloud computing use makes it important to consider how network conditions in data centres affect an applications performance. In this chapter, I studied the effects of network latency on typical cloud applications, ranging from DNS to distributed machine learning applications (§5.2). I performed extensive measurements by artificially injecting controlled network latency in an experimental setup described in §5.1, quantifying the impact of

network latency on application performance (§5.4). The results of the experiments show that different applications react differently to changing network latency (§5.4.2). The fact that network latency affects performance is well known [Che96; Bar14; DB13], yet the results of the experiments performed in this chapter show the extent of sensitivity to latency for some applications: 10μs latency in each direction enough to have a noticeable effect, and 50μs latency in each direction enough to significantly negatively impact the performance. Given that only a few years ago this was the scale of latency within the host alone [ROS+11], it means optimising end-host latency continues to be of importance. A second noteworthy aspect is the distance between servers. With the longest cable length reaching 900m [GBQ+14], equivalent to 9μs RTT on the fibre alone, and expected to grow, performance can be noticeably affected. Scaling the data centre comes at the cost of additional hops between servers, which means that passing through more switches increases latency further. This has ramifications for workload placement when trying to meet application performance guarantees. In the next chapter, I show how predictions of the application performance (§5.5) aid cluster management software to schedule the applications to ensure optimal application performance given the current network conditions.

Chapter 6

NoMora: latency-driven, application performance-aware, cluster scheduling

With more and more businesses and government institutions moving their operations to the cloud, a lot of attention has been devoted to providing a fast data centre network with predictable application performance for customers. An important factor in achieving predictable application performance is understanding the networking requirements of the application in terms of bandwidth and latency. Once these requirements have been determined, they have to be incorporated into the data centre management stack. This can be done in-network, through scheduling [ARR+10; BAA+11; POB+14] or prioritising [GSG+15; AYS+13; DKB+14; CZS14] the application's flows, and-or at the end-host, through bandwidth allocation [KJN+15]. In these situations, the placement of the application's tasks is assumed to be known before incorporating its network resource demands. If the tasks' placements are not known a priori or if they can be changed, the network resource demands can be incorporated at a higher level in the data centre management stack, namely in the cluster scheduler. In Section 2.7, I summarised previous work that incorporated network bandwidth and tail latency demands in the cluster scheduler's decisions. However, none of the cluster schedulers have placed an application's tasks according to their expected performance as predicted by the current network conditions.

In Chapter 5, I demonstrated that network latency affects typical cloud applications' performance through experiments performed on a custom testbed, and I showed how the experimental results can be used to build functions that predict application performance based upon network latency for typical cloud applications. These functions can be abstracted in a way in which they can be understood by a cluster scheduler.

In this chapter, I use these functions in a cluster scheduling architecture, NoMora[1], extending the Firmament [GSG+16] cluster scheduling framework. The core of NoMora is a cluster scheduling policy that places the tasks of an application (job) taking into account the expected

[1] *mora* means delay in Latin, so the name refers to applications being scheduled to not have network delay affecting their performance

performance based on the measured network latency between pairs of hosts in the data center. Furthermore, if a tenant's application experiences increased network latency due to unexpected conditions, *e.g.*, network congestion, and thus lower application performance, their job's tasks may be migrated to a better placement. As shown in Chapter 4, network latency values can change substantially over time, meaning that applications can achieve better performance if their placement within the data centre takes into account the current network conditions.

While incorporating network latency demands at the cluster scheduler level can be viewed as being a coarse-grained solution compared to the granularity offered by in-network flow management solutions, I show that such an approach is feasible for improving application performance.

6.1 Background

NoMora extends the Firmament cluster scheduler [GSG+16]. I chose to extend Firmament because it is a centralised scheduler that considers the entire workload across the whole cluster, making it straightforward to incorporate the network latency measured between every pair of hosts in the cluster, and due to its low latency (sub-second) task placement (§2.7.2.1).

6.1.1 The cluster scheduling problem modelled as a flow network

In this section, I give an overview of how the cluster scheduling problem is mapped to the minimum-cost maximum-flow optimisation problem, as described in Quincy [IPC+09] and Firmament [GSG+16].

6.1.1.1 Flow network

Firstly, I provide a high-level overview of the structure of the flow network, which can be seen in Figure 6.1. By *flow network* I refer to a directed graph where each arc has a capacity and and a cost to send flow across that arc. Each submitted task $T_{i,j}$, representing task j of job J_i, is represented by a vertex in the graph, and it generates one unit of flow. The sink S drains the flow generated by the submitted tasks. A task vertex needs to send a unit of flow along a path composed of directed arcs in the graph to the sink S. The path can pass through a vertex that corresponds to a machine (host) M_m, meaning the task is scheduled to run on that machine, or it can pass through a special vertex for the unscheduled tasks of that job U_i, meaning that the task is not scheduled. In this way, even if the task is not scheduled to run, the flow generated by this task is routed through the unscheduled aggregator to the sink.

The graph can have an arc between every task and every machine, but this would make prohibitive the computation of an optimal scheduling solution in a short time, as the graph would scale linearly with the number of machines in the cluster. To reduce the number of arcs in the

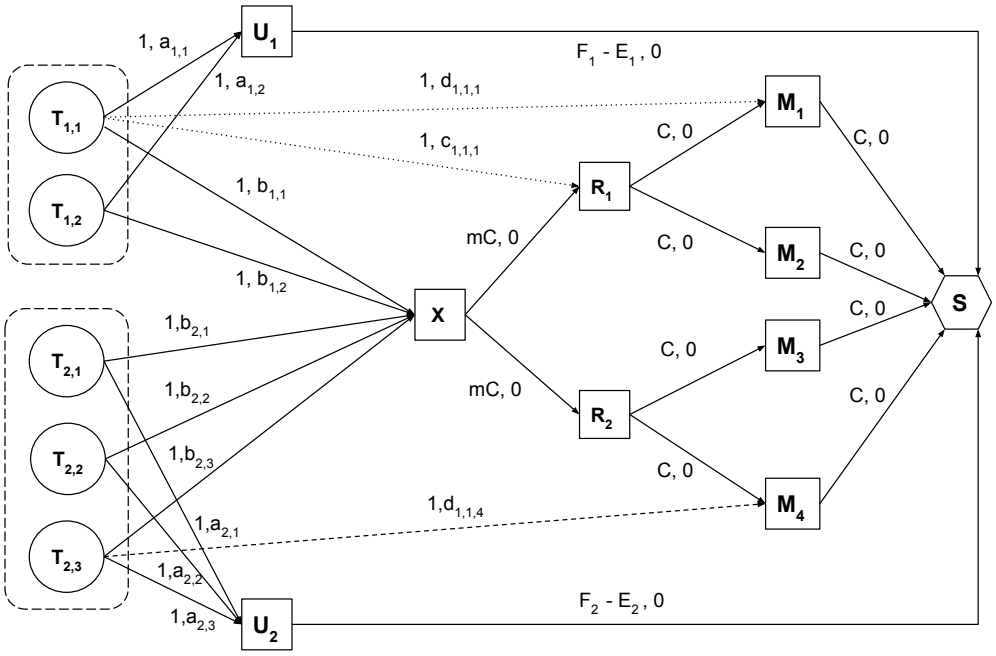


Figure 6.1: A general flow network with annotated capacities and costs on arcs. Job J_1 has tasks $T_{1,1}$ and $T_{1,2}$. Job J_2 has tasks $T_{2,1}$, $T_{2,2}$ and $T_{2,3}$. The unscheduled aggregator is U_1. The machines in the cluster are M_1, M_2, M_3 and M_4. Rack aggregators are R_1 and R_2. The cluster aggregator is X. The sink vertex is S.

graph, a cluster aggregator X and rack aggregators R_r have been introduced in Quincy, inspired by the topology of a typical data centre. The cost of the arc between a task and the cluster aggregator is the maximum cost across all of the machines in the cluster. Similarly, the cost of the arc between a task and a rack aggregator R_r is the maximum cost across all of the machines in that rack. It can be easily seen that the costs to the cluster and rack aggregators serve as a conservative approximation, providing an upper bound for a set of resources that are grouped together.

The cluster and rack aggregators are implemented as equivalence classes in Firmament. Firmament defines the notion of *equivalence classes* for elements whose behaviour is similar, and can be defined for both tasks and machines. The benefit of defining equivalence classes is that it reduces the number of arcs for a pair of equivalence classes with sizes n and m from $O(nm)$ to $O(n+m)$.

To summarise, the flow network has the following types of vertices:

1. task vertices $T_{i,j}$ - a task that is submitted, it can be scheduled or unscheduled,
2. machine vertices M_i - a machine in the cluster,
3. aggregator vertices - unscheduled aggregator vertices U_i (there is one for each job, all the unscheduled tasks in the job are connected to the unscheduled aggregator), rack aggregator vertices R_i (which connects all machines in a rack), the cluster aggregator vertex X (all the rack aggregators are connected to it), and

4. a sink vertex S - the flow coming from the task vertices is drained in the sink vertex.

The flow network has the following types of directed arcs:

1. an arc from a task to a resource (machine, rack aggregator or cluster aggregator), referred to as *preference arcs*,
2. an arc from a task to a machine on which the task is running,
3. an arc from a task to the unscheduled aggregator of the job's task,
4. an arc from a resource to another resource (cluster aggregator to rack aggregator, rack aggregator to machine, machine to machine), and
5. an arc from a machine or from an unscheduled aggregator to the sink.

6.1.1.2 Capacity assignment

Each arc in the flow network has a capacity c for flow, bounded by c_{min} and c_{max}. In Firmament and Quincy, c_{min} is usually zero, while c_{max} depends on the type of vertices connected by the arc and on the cost model. Given that the minimum capacity is zero, it is omitted from here onwards [Sch16], with only the maximum capacity values being presented.

The capacity of an arc between a task and any other vertex is 1. If a machine has C cores and a rack has m machines, the capacity of an arc between a rack aggregator and a machine is C, and the capacity of an arc between the cluster aggregator and a rack is $C \times m = Cm$. The capacity of an arc between a machine and the sink is C.

The capacity between an unscheduled aggregator U_i and the sink S is represented by the difference between the maximum number of tasks to run for a job J_i, F_i, and the minimum number of tasks to run for job J_i, E_i, with $0 \leq E_i \leq F_i \leq N_i$, where N_i is the total number of tasks in job J_i. These limits can be used to ensure a fair allocation of runnable tasks between jobs [IPC+09; Sch16].

6.1.1.3 Cost assignment

The cost on an arc represents how much it costs to schedule any task that can send flow on this arc on any machine that is reachable via this arc.

Task to machine arc The cost on the arc between a task vertex $T_{i,j}$ and a machine vertex M_m is denoted by $d_{i,j,m}$, and is computed according to information regarding the task and machine. In most cases, this cost is being decreased by how much the task has already run, $\beta_{i,j}$.

Task to resource aggregator arc The cost on the arc between a task vertex $T_{i,j}$ and a rack aggregator vertex R_r, denoted $c_{i,j,r}$, represents the cost to schedule the task on any machine

within the rack, and is set to the worst case cost amongst all costs across that rack. The cost on the arc between a task vertex $T_{i,j}$ and the cluster aggregator X, denoted by $b_{i,j}$, represents the cost to schedule the task on any machine within the cluster, and is set to the worst case cost amongst all costs across the cluster.

Task to unscheduled aggregator arc The cost on the arc between a task vertex $T_{i,j}$ and the unscheduled aggregator U_i, denoted by $a_{i,j}$, is usually larger than any other costs in the flow network. The cost on this arc increases as a function of the task's wait time, in order to force the task to be scheduled, and it is scaled by a constant *wait time factor* ω, which increases the cost of tasks being unscheduled.

Preemption If preemption is enabled, the scheduler can preempt a task that it is running on a machine, which means the flow pertaining to that task is routed via the unscheduled aggregator, or migrate the task to a different machine, meaning that the flow is routed via that new machine's vertex. If preemption is not enabled, then a scheduled task will have in the flow network only the arc to the machine that it is currently running on, with all the other arcs being removed once the task is scheduled.

6.1.2 Firmament overview

I present in Figure 6.2 the architecture of Firmament as described in [Gog17], and I briefly describe its main components. A coordinator process runs on each machine in the cluster. Its roles are scheduling tasks, monitoring tasks, and collecting resource utilisation statistics and information about the machine's hardware specifications. This information is sent to the master coordinator process. The master coordinator process schedules the tasks and assigns them to worker coordinator processes that run on machines. It also aggregates the information received from the worker coordinator processes, storing the information into the knowledge base. The master coordinator process builds a cluster resource topology based on the machines' hardware specifications. The knowledge base also stores information for each task and builds task profiles. The task profiles and the resource topology are used by the master coordinator's scheduler in scheduling policies to make placement decisions.

Firmament exposes an API to implement scheduling policies, that may incorporate different task constraints. A scheduling policy defines a flow network representing the cluster, where the nodes define tasks and resources, as described in Section 6.1.1.1. The policy can also use task profiles to guide the task placement through preference arcs to machines that meet the criteria desired by the task. Events such as *task arrival*, *task completion*, *machine addition to the cluster*, or *machine removal from the cluster*, change the flow network. When cluster events change the flow network, Firmament's min-cost flow solver computes the optimal flow on the updated flow network. The updates to the flow network caused by the cluster events are not applied while the solver runs, but only after the solver finishes computing the optimal solution. After the solver finishes running, Firmament extracts the task placements from the optimal flow,

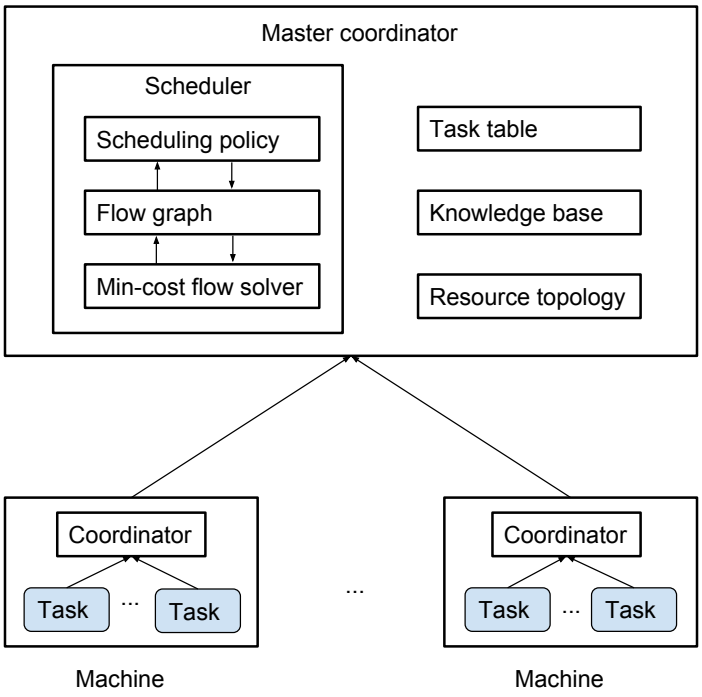


Figure 6.2: Firmament architecture.

and applies these changes in the cluster.

Firmament [2] supports several cost models: trivial cost model (random placement), load-spreading policy (load balance the tasks across machines), Whare-Map cost model (avoids interference and exploits machine heterogeneity), coordinated co-location (CoCo) cost model (extends the Whare-Map cost model), green cost model (uses power consumption as input) and network bandwidth-aware cost model (avoids oversubscribing the end-host network interface).

6.2 NoMora

6.2.1 Architecture

I combine the following three elements in the NoMora cluster scheduling architecture, as seen in Figure 6.3, where (1) and (2) offer inputs to (3):

1. functions that predict application performance dependent upon network latency;
2. network latency measurement system;
3. the latency-driven application performance-aware cluster scheduling policy implemented on top of the Firmament cluster scheduler.

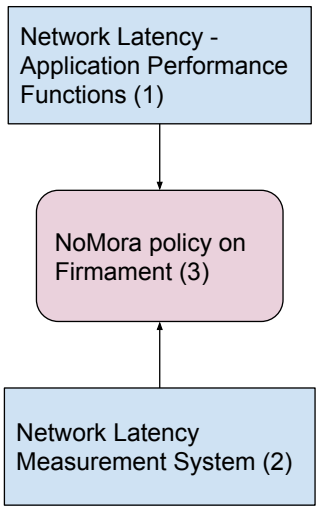


Figure 6.3: NoMora architecture.

The functions that predict application performance dependent upon network latency were determined in Chapter 5. The second component of the architecture is the network latency measurement system. Systems such as PTPmesh [PM17] (described in Chapter 4), Pingmesh [GYX+15] or NetNORAD [ALZ16] can provide the most recently measured network latency between hosts. Due to the scale of data centres, these systems do not send probes between every pair of hosts, but instead choose fewer hosts to ensure the largest coverage. Additionally, they set a minimum probing interval to bound the network traffic generated. The data collected by these systems is fed to the cluster scheduler to aid in making task placement or migration decisions. Default latency values can be determined based on the network topology of the data centre to be used instead of the actual measured latencies if these are unavailable. The third component of the system, the policy, is discussed in the following section.

6.2.2 Latency-driven, application performance-aware, policy

I propose a new *latency-driven, application performance-aware,* policy whose goal is to place distributed applications in a data centre in a manner that gives them improved application performance. This generally leads to grouping tasks as close as possible, in a rack or on the same machine, for the applications for which latency matters, such as Memcached or machine learning frameworks (STRADS, Tensorflow). For the tasks that do not fit within the same rack or on the same machine, the policy finds the machine that offers the best application performance amongst the available placements. On the other hand, applications like Spark, for which additional latency of up to one millisecond does not affect substantially its performance, will have more freedom when being placed within the data centre. Furthermore, if the network conditions

[2] `https://github.com/camsas/firmament`

change, a task whose performance degrades can be migrated to a better placement.

Since the applications I studied in Chapter 5 are client-server applications or worker-master applications, I consider that the server/the master has a special role, because it has to be running before the clients/workers. I call the server (for client-server applications) / the master (for master-workers applications) the *root task*. Thus, the policy needs to schedule the root task first. The root task is scheduled immediately in any place available in the cluster. The other tasks of the job (clients/workers) are not scheduled until the root task is scheduled. While this adds delay in scheduling for these tasks, the delay is minimal, since they will be scheduled in the next scheduling round based on the placement of the root task. In my policy, a task's placement does not depend on a machine's architectural properties (CPU, RAM, etc.) or on the properties of the other tasks that run on the machine, but on the placement of another task (the root task), and on the network latency between a machine considered for the task's placement and the root task's placement. My policy uses, for assigning costs to arcs, the application performance predictions and network latency measurements between hosts to determine the expected application performance.

In summary, the placement of a task follows these steps:

1. the root task is scheduled on any available machine;

2. if a task that is not a root task enters the system at the same time as the root task, or before the root task is scheduled, it will not be scheduled, waiting instead;

3. if the root task is scheduled, then a new task's placement is determined based on the application performance prediction, and current network latencies to the root task's placement.

6.2.2.1 Flow network

The flow network can be seen in Figure 6.4. Arcs are defined between a task and the potential machines on which it can run, and each arc has a cost computed using the application performance predictions dependent upon network latency for each application.

When a job is submitted, the root task $T_{i,0}$ is assigned a single arc to the cluster aggregator, with a cost of 0, which means that the root task will be scheduled immediately on any available machine. After it is scheduled, the root task will have an arc from the root task to the machine it is running on. The other tasks of the job will wait for the root to be scheduled first, and they do not have any arcs initially. After the root task is scheduled, each task $T_{i,j}$ will have preference arcs to the cluster aggregator X, to rack aggregators R_r and machines M_m based on the cost to schedule the task on those resources and on the parameters of the policy.

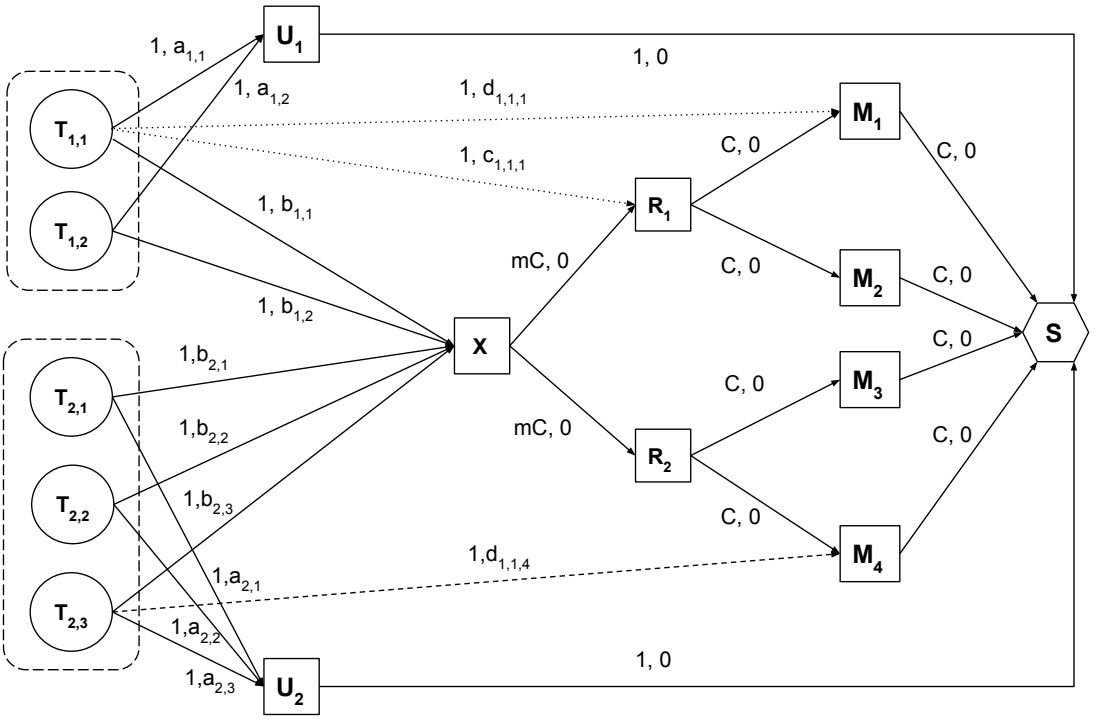


Figure 6.4: NoMora flow network with annotated capacities and costs on arcs.

6.2.2.2 Cost assignment

The cost assignment is also called *cost model* [GSG+16]. NoMora's cost assignment is similar to Quincy's cost assignment [IPC+09]. The Quincy policy considers in the cost computation data locality, task wait time (the time a task waits before being scheduled) and how much time a task has run before being preempted (*preemption cost*). These factors are considered because having good data locality reduces the job's runtime in the scenario considered by Quincy, but finding a good placement can mean waiting more time until a suitable machine is free (the task wait time increases), or preempting a task that is already running (if this task is restarted from the beginning on another machine, then the time the task has already run is lost). Instead of considering data locality, in NoMora I consider the cost to the root task computed based on application performance prediction dependent upon network latency for a job (as built in Chapter 5, Section 5.5), combined with measured network latency between the root task machine and the machine under consideration. Similarly to the Quincy cost model, in NoMora I also factor the task wait time when computing the cost of the arc to the unscheduled aggregator, and, if preemption is enabled, the preemption cost in the arc cost computation.

Table 6.1 provides on overview of the costs NoMora assigns to different arcs.

Assuming the root task is running on machine M_{root} and a task j, $T_{i,j}$, of job J_i can be scheduled on machine M_m, then the cost of the arc from $T_{i,j}$ to M_m is:

$$d_{i,j,m} = cost(T_{i,j}, M_m) = \frac{1}{p(max(latency(M_{root}, M_m)))} \tag{6.1}$$

Arc	Capacity	Value	Cost
$T_{i,j} \to U_i$	1	$a_{i,j}$	Cost of leaving $T_{i,j}$ unscheduled
$T_{i,j} \to X$	1	$b_{i,j}$	Cost of scheduling $T_{i,j}$ on the worst machine
$T_{i,j} \to R_r$	1	$c_{i,j,r}$	Cost of scheduling $T_{i,j}$ on the worst machine in rack R_r
$T_{i,j} \to M_m$	1	$d_{i,j,m}$	Cost of scheduling $T_{i,j}$ or continuing to run on machine M_m
$X \to R_r$	mC	0	-
$R_r \to M_m$	C	0	-
$M_m \to S$	C	0	-
$U_i \to S$	1	0	-

Table 6.1: Arcs in the NoMora flow network with their capacities and costs.

where $p(max(latency(M_{root}, M_m)))$ is the expected application performance for the measured network latency between machine M_{root} and machine M_m, as determined in Section 5.5. I invert the performance because, when the performance is smaller, the cost assigned to the arc is higher, making the machine to which the arc points to less desirable for running the task on. Since in data centres typically there are multiple paths between two machines, in order to be conservative, I use the maximum latency value measured between the two machines because, due to ECMP, I cannot know which of the available paths the application's flows will take.

Preempting a task presents a trade-off between migrating the task to a better placement and the amount of time the task has already run (on the current machine or on a different one) [ASR+10]. If preemption is enabled, the amount of time the task has already run, $\beta_{i,j}$, can be subtracted from the $cost(T_{i,j}, R_r)$. This leads to less task migrations happening, because it becomes less advantageous to preempt a task and restart it on another machine after migration the more time the task is running, essentially wasting the work that has already been done.

$$d_{i,j,r} = cost(T_{i,j}, M_m) - \beta_{i,j} \tag{6.2}$$

Similarly, the cost of the arc from $T_{i,j}$ to rack R_r is the cost to the worst cost machine in rack r:

$$c_{i,j,r} = cost(T_{i,j}, R_r) = \max_{m \in r} \frac{1}{p(max(latency(M_{root}, M_m)))} \tag{6.3}$$

where $p(max(latency(M_{root}, M_m)))$ is the expected normalised application performance for the measured network latency between machine M_{root} and a machine M_m in rack r. Similarly, to be conservative due to ECMP, I take the maximum value of the latencies between M_{root} and M_m.

The cost to the cluster aggregator X is the cost to the worst cost rack, which is obtained by taking the maximum of the costs to racks.

$$b_{i,j} = \max_r c_{i,j,r} \tag{6.4}$$

The cost to the unscheduled aggregator U_i is computed using the task's wait time, $\alpha_{i,j}$, scaled by a constant factor ω (§6.1), to which a constant cost factor γ, that is larger than any other possible arc costs, is added.

$$a_{i,j} = \omega \times \alpha_{i,j} + \gamma \tag{6.5}$$

The costs on the arcs are rounded to two significant digits, and then multiplied by a factor of 100, since the costs must be integer numbers for the solver to understand. For a normalised performance of 1, the cost is $\frac{1}{1} \times 100 = 100$. For a normalised performance of 0.1, the cost is $\frac{1}{0.1} \times 100 = 1000$. The cost to the unscheduled aggregator is offset by γ, greater than all the other possible costs.

Since the network latency is not constant in a data centre, as shown in Chapter 4, the costs associated with the arcs are updated based on the latest measured network latency values, and as a result, the preference arcs for the tasks are updated. If preemption is enabled, the cost of the arcs for a running task will also be updated.

6.2.2.3 Cost model parameters

The cost model has two main parameters:

- threshold for the cost on an arc to a machine in order for that machine to be on the preferred list of machines on which the task can run, p_m,
- threshold for the cost on an arc to a rack in order for that rack to be on the preferred list of racks in which the task can run, p_r.

The first preference list comprises the machines on which the application may run to achieve the desired performance. This list should be kept small for the task scheduling to take a reasonable amount of time. But having a small preference list means the application's placement options are limited. To mitigate this, the second preference list, which comprises the racks on which the application may run, was introduced. The second list is smaller than the first one, since the number of racks is smaller than the number of machines. This allows a bigger threshold to be set for the second parameter of the model, offering more placement options for the application's tasks, while keeping the first preference list small.

6.3 NoMora evaluation

I evaluate NoMora in simulation, using the same simulator as for Firmament's evaluation, extended to provide network latency measurements between pairs of machines, and to update them during the simulation. Secondly, I added application performance predictions dependent

upon network latency per job and per task (same function for all tasks of a job). Finally, I implemented the policy that uses these predictions and the latency measurements to compute task placements.

Cluster workloads As explained in Section 2.7.1, no information about the network communication patterns between a job's tasks, nor about their sensitivity to network latency are provided in public cluster workloads. Thus, I have assigned the network latency to application performance functions determined in Chapter 5, Section 5.5 to the jobs in the Google workload. I did not include the single task jobs, as they do not communicate with any other task. I used 24 hours of the trace.

Application performance predictions dependent upon network latency The predictions are discretised in steps of 10μs, and are stored in a hash table for each job. The network latency value between two machines is rounded to the nearest latency value for which the prediction function has an entry in the hash table. For the latency values in the used traces that are outside the interval of defined values, I use the smallest performance value defined for that function. The different prediction functions are assigned randomly in different proportions to the jobs. For the experiments presented in this section, 50% of the jobs use the Memcached prediction, 25% of the jobs use the STRADS prediction and 25% the Tensorflow prediction. This scenario is one of the most challenging, as Memcached is the most latency sensitive application that I studied. I did not use the Spark prediction, which is almost constant, as it would not be challenging to place such jobs. Given the functions built in Chapter 5, for which the normalised performance does not drop below 0.1, I set $\gamma = 1001$ for the simulation.

Network latency measurements The simulator leverages the network latency measurements collected in Chapter 4. With 18 week-long traces collected, I further divide each trace in 7 (for each day of the week), and I assign them to machine pairs considering the physical distance between servers as a criterion. Assuming a typical fat-tree topology for a data centre [ALV08] and based on the latency values measured in Microsoft Azure by [RBB+18], I use the traces with the lowest values for machine pairs located in the same rack (6 traces - GCE), the traces with intermediate values for machine pairs located within the same pod (6 traces - Azure), and the traces with the largest values for machines located in different pods (6 traces - EC2). These traces are used to provide the latency values between hosts for the duration of the simulation, which is one day. Since I do not have different traces for each machine pair, I scale the values of each trace using a coefficient between 0.8 and 1.2, selected randomly for intra-pod and inter-pod values. For the traces within the rack, I scale them between 0.5 and 1. For the latency values between cores on the same server I use a small constant. Latency values from traces are provided every second in the simulation, in total $86,400$ per day between every pair of hosts in the cluster.

Topology I use the Google trace which has 12,500 machines [RTG+12]. The machines used in the workload are grouped into racks and pods at the beginning and during the simulation. I set the number of machines per rack to 48, and the number of racks per pod to 16. The results will be influenced by the number of hosts per rack and the number of racks per pod due to the

assignment of the different cloud latency traces. These two numbers were chosen to reproduce a small cluster. However, if there are more hosts per rack and more racks per pod, then there will be a greater chance to fit all the tasks of a job in the same rack or in the same pod, meaning the job will have a good overall application performance. The size of a job in terms of tasks should thus be taken into consideration when designing novel data centre topologies.

I performed experiments with the settings of the Facebook data centre topology (192 hosts per rack and 48 racks per pod) [And14], but for a cluster of only 12,500 machines as the one presented in the Google trace, it means there is only one complete pod and an incomplete one, with a total of approximately 260 racks. The overall application performance in this case is very high due to the small network latencies assigned between the hosts within the same rack.

Adjusting job runtime in simulation In Table 5.1, I presented the performance metric of each application. For some applications (Memcached, DNS) I use the number of queries per second, while for the rest (Spark, STRADS, Tensorflow) I use the training time, which is equivalent to the job's runtime. In the second case, it it hard to adjust in simulator the expected job runtime based on the measured latency at every second. As such, I do not adjust the job runtimes. I use instead the normalised value of the application performance as performance metric for all jobs, and I adjust this performance metric based on the current network conditions.

6.3.1 Evaluation metrics

Through the evaluation of the NoMora cluster scheduling policy, I seek to answer the following questions:

- does NoMora's placement improve application performance compared to a random placement policy and a load-spreading policy?
- how long does it take to compute a placement solution?
- how long does a task have to wait before starting to run?

In order to know if my policy improves application performance compared to other policies, I compute the **average application performance**. This metric measures NoMora's task placement quality. It is computed as the application performance determined by the network latency in every measurement interval divided by the maximum application performance that could be achieved in every measurement interval, and it is computed for the job's total runtime.

The next two metrics used to evaluate NoMora were also used to evaluate Firmament.

The **algorithm runtime** is the time it takes for Firmament's min-cost max-flow algorithm to run. The *Flowlessly* solver is the min-cost max-flow library utilised in Firmament, and combines several techniques to reduce the solver runtime. I compare my policy's runtime with other Firmament policies' runtimes to ensure that my policy is scalable when run by a centralised

cluster scheduler. Additionally, the time it takes to compute the applications' placements in a round gives an indication of the time interval that is needed between latency measurements. For example, if the algorithm runtime would be in the order of minutes, running latency measurements every few seconds would not be useful, since the measurements accumulated over the scheduler's runtime would not be used by the scheduler. On the other hand, if the algorithm runtime is in the order of milliseconds, running latency measurements every second or few seconds is useful, the scheduler being able to use the most recent measurement data.

The **task placement latency** is the time between task submission and task placement. This metric includes the task wait time, which should be as short as possible, but also the time it takes for Firmament to update the flow network, Flowlessly's runtime and the time to iterate over the placements computed by Flowlessly. In the context of my policy, the metric also captures by how long the tasks are delayed when they are waiting for their root task to be placed first before them.

Another metric that I looked at is the *task response time*, which is the time between a task's submission and its completion.

6.3.2 Placement quality

I compare the NoMora policy, using different parameter values for the cost model (§6.2.2.3), with a random policy that uses fixed costs (tasks always schedule if resources are idle), and a load-spreading policy that balances the tasks across machines. I enable preemption only for the NoMora policy, since the two other policies would not benefit from preemption due to their different scheduling goals. The random policy schedules tasks if resources are available, thus migrating a task does not make sense, since the task is already running. The load-spreading policy schedules tasks based on current task counts on machines, thus potential task load imbalance can be handled by scheduling new tasks on less loaded machines instead of migrating already running tasks on the less loaded machines.

The results for the average application performance for different policies can be seen in Figure 6.5. I compute the area marked by the y-axis, the CDF, and the straight horizontal line with $y = 1$, for each policy. According to this computation, the maximum area corresponding to the maximum average application performance across applications is 100%, and it would be obtained for a vertical line at $x = 100\%$. Next, I subtract from the NoMora policies areas the random and load-spreading areas to assess the placement improvement given by the NoMora policy. The total area for the random policy is 47.2%, while for the load-spreading policy it is 46.8%. For the NoMora policy with parameters $p_m = 105$ and $p_r = 110$ the area is 60.2%, NoMora with parameters $p_m = 105$ and $p_r = 110$ and preemption enabled is 59%, NoMora with parameters $p_m = 110$ and $p_r = 115$ is 51.85%, and finally, NoMora with parameters $p_m = 105$ and $p_r = 110$, and preemption enabled with $\beta_{i,j} = 0$, is 89.6%. The maximum overall improvement without preemption enabled is 13% over the random policy and 13.4% over the

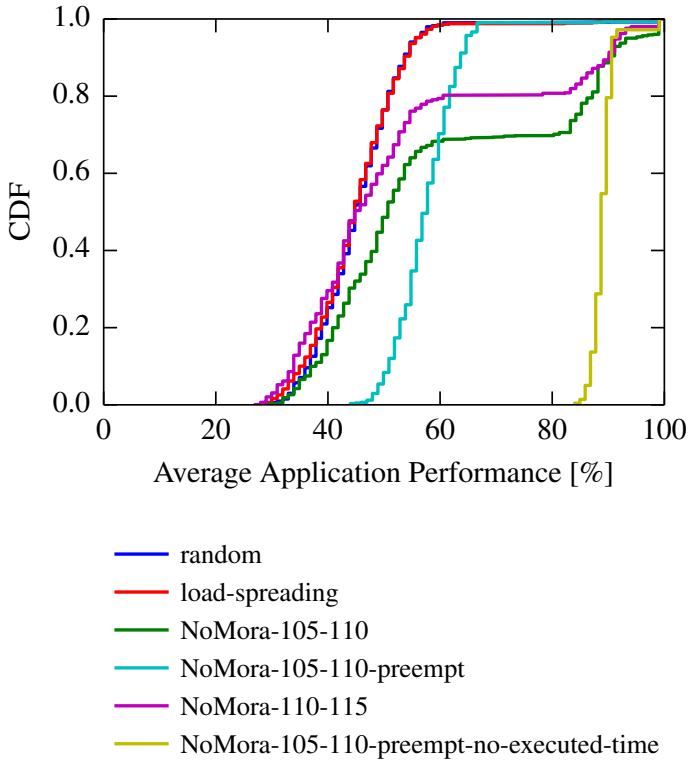


Figure 6.5: Average application performance for different policies on the Google workload.

load-spreading policy, and is obtained for NoMora with parameters $p_m = 105$ and $p_r = 110$. If preemption is enabled and $\beta_{i,j} = 0$ (the time already executed by a task is not considered in the arc cost computation), the improvement is considerable, 42.4% over the random policy, and 42.8% over the load-spreading one.

The improvement in average application performance is not substantial when preemption is not enabled because of the root task's random placement, and also because of a smaller number of available places a task can be scheduled because of long-running jobs that are set up at the beginning of the trace. The tasks of the jobs are placed in the best available slots in relation to the root task's placement. In this way, I constrain the available placements, and the policy searches for placements in relation to a known location rather than trying to find the a placement for all the tasks of a job simultaneously. I further explain the reason behind this design decision and its implications in Section 6.4.

It can be seen that the CDF of NoMora with *preemption enabled* has a different shape than the other policies. This is due to task preemption, which can correct the initial placement if it is not good (because of the random placement of the root task), and it can also migrate tasks when their current placement is not good anymore. The improvement provided by the NoMora policies without preemption is evident from Figure 6.5, but it can also be seen that the CDFs start at approximately the same value (27%-28% average application performance), and have an initially similar shape to the random and load-spreading CDFs. On the other hand,

for NoMora with preemption enabled, the minimum average application performance is 44% and 84% respectively, which means that the improvement in application performance happens across all jobs due to migration to better placements.

6.3.3 Algorithm runtime

The algorithm runtime depends on the number of arcs from each task to the resources, but also on the cluster size. As the number of arcs or the cluster size increases, so does the algorithm runtime. The two parameters of the cost model (§6.2.2.3) influence the number of arcs the graph has between task nodes and machine nodes or rack nodes, and hence the algorithm runtime, which depends on the flow network size and on the number of tasks considered per scheduling round. If the thresholds are lower, the preference lists will be smaller. In this case, the applications' performance will be higher (only high-quality placements considered), but they will have less placement options to be scheduled, and thus the wait time may increase. The tasks will have to wait for the machines that offer the performance desired to have empty slots. However, setting a high threshold means the preference lists will be larger, which could lead to an increase in the algorithm runtime. On the other hand, more placement options will be available for the tasks to be scheduled, reducing their wait time. In practice, the algorithm runtime may not necessarily increase. With more placement options available, the tasks may be scheduled sooner, thus leading to less tasks being scheduled per round, resulting in a decrease in the algorithm runtime per scheduling round.

Figure 6.6 presents results for the algorithm runtime for the load-spreading policy, random policy and for the NoMora policy with and without preemption on the Google workload. The two parameters of the cost model are set as in the previous experiment. The random and load-spreading policies have a similar algorithm runtime, with a median runtime of 108ms-109ms. However, the two policies differ at the tail: for the random policy the 99$^{\text{th}}$percentile is 661ms and a maximum of 18.89s, while for load-spreading policy the 99$^{\text{th}}$percentile is 974ms and a maximum of 25.88s. For NoMora with parameters $p_m = 105$ and $p_r = 110$, the median algorithm runtime is 93ms (99$^{\text{th}}$percentile is 248ms, and maximum is 6.13s), an improvement of 1.61$\times$ for the median runtime, and 2.66$\times$ and 3.92$\times$ at the 99$^{\text{th}}$percentile, compared to the baselines. For NoMora with $p_m = 110$ and $p_r = 115$, median runtime is 72ms (99$^{\text{th}}$percentile is 486ms, and maximum is 39.55s). On the other hand, the maximum value for the algorithm runtime is considerable larger in the case of NoMora policies.

NoMora with parameters $p_m = 105$ and $p_r = 110$ with preemption enabled takes a considerable longer amount of time, because of the higher number of arcs in the flow network compared to the case when preemption is not enabled (the arc preferences of the tasks that are running are not removed, unlike when preemption is not enabled), and the updates made to the flow graph (adding or changing running arcs to resources), further resulting in a larger number of tasks considered per scheduling round. This also translates into a larger task placement latency (§6.3.4).

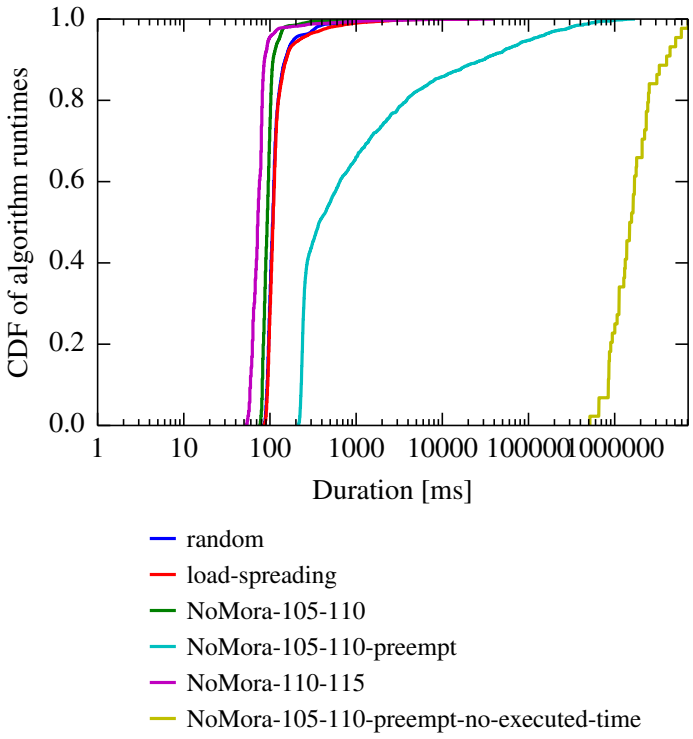


Figure 6.6: Algorithm runtime for different policies on the Google workload.

As can be seen from Figure 6.7, the percentage of migrated tasks in the first case (NoMora with preemption enabled and already executed time for a task considered in the arc computation) is on average 0.022% per scheduling round, with a 99$^{\text{th}}$percentile of 0.5%. If $\beta_{i,j} = 0$ (already executed time is not considered in the arc computation), a considerable number of task migrations take place: an average of 7.1% per scheduling round, with a 99$^{\text{th}}$percentile of 10.07%. This happens because the time a task has already run is ignored in the arc cost computation, meaning that the cost is based solely on the expected application performance under the given network conditions. The median algorithm runtime time is 373ms, the 99$^{\text{th}}$percentile is 511s and the maximum is 1719s, which is 3.45× larger than the baseline for the median runtime, and 773× larger than the baseline for the 99$^{\text{th}}$percentile. In the second case ($\beta_{i,j} = 0$), the median running time is 1532s, the 99$^{\text{th}}$percentile 6610s, and the maximum is 7118s. This significant algorithm runtime means that preemption should be used with care. For example, only certain applications that explicitly demand to be migrated should be migrated, or migration can be triggered only if the application performance drops below a certain threshold.

6.3.4 Task placement latency

Figure 6.8 presents the task placement latency, which is at the median 436ms, 90$^{\text{th}}$percentile is 312ms and 99$^{\text{th}}$percentile is 1.9s for the random policy; median is 498ms, 90$^{\text{th}}$percentile is 4.3s and 99$^{\text{th}}$percentile is 25s for the load-spreading policy, median is 278ms, 90$^{\text{th}}$percentile is 1.23s and 99$^{\text{th}}$percentile is 5.8s for the NoMora policy with parameters $p_m = 105$ and $p_r = 110$;

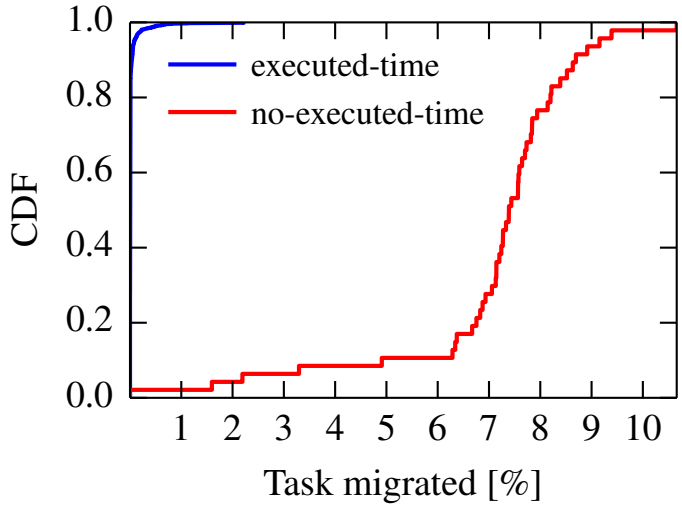


Figure 6.7: Percentage of migrated tasks for NoMora policy with preemption (parameters 105 and 110) on the Google workload.

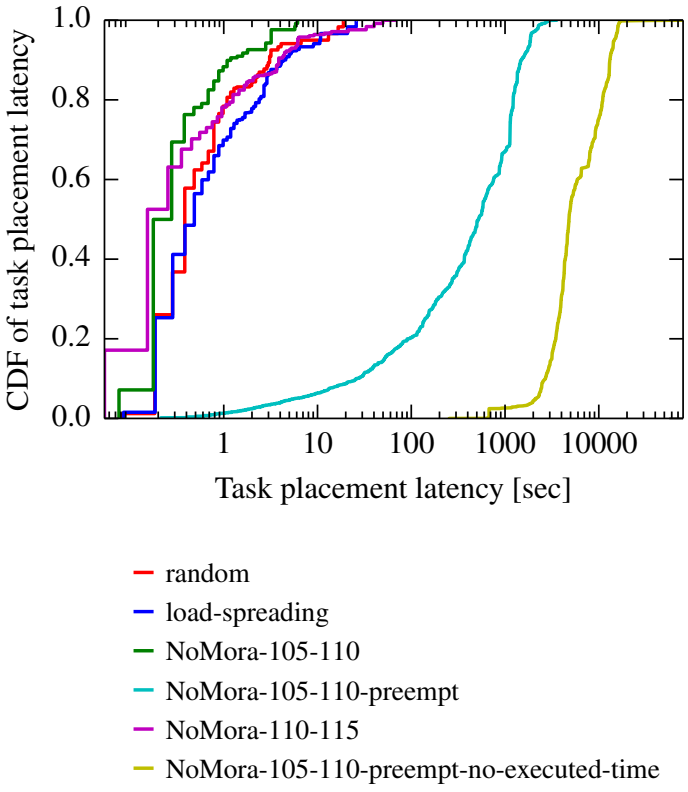


Figure 6.8: Task placement latency for different policies on the Google workload.

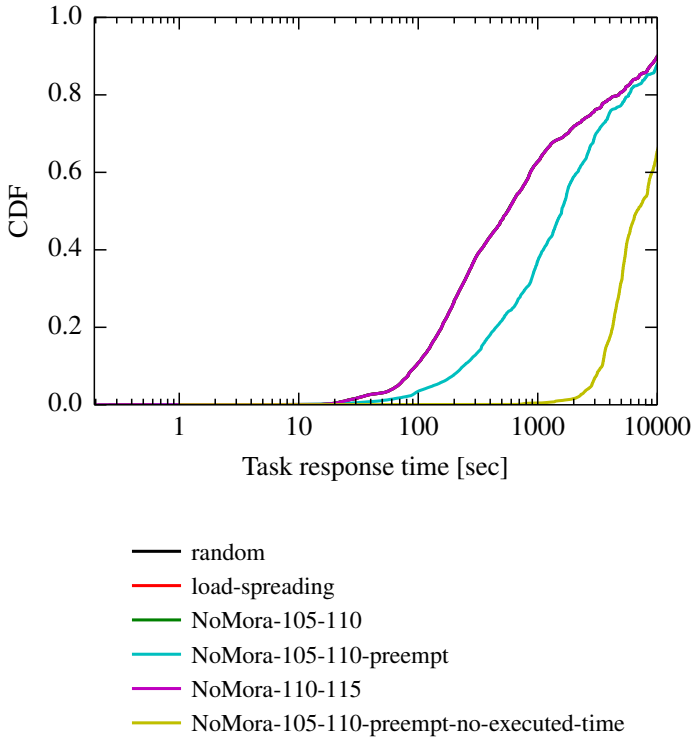


Figure 6.9: Task response time for different policies on the Google workload.

median is 185ms, 90$^{\text{th}}$percentile is 1s and 99$^{\text{th}}$percentile is 5.6s for the NoMora policy with parameters $p_m = 110$ and $p_r = 115$; median is 519s, 90$^{\text{th}}$percentile is 1484s and 99$^{\text{th}}$percentile is 2458s for the NoMora policy with preemption enabled and parameters $p_m = 105$ and $p_r = 110$; and median is 4812s, 90$^{\text{th}}$percentile is 13077s and 99$^{\text{th}}$percentile is 16251s for the NoMora policy with preemption enabled, $\beta_{i,j} = 0$ and parameters $p_m = 105$ and $p_r = 110$.

The NoMora policy with parameters $p_m = 105$ and $p_r = 110$ improves the median task placement latency by 1.56× compared to the random policy and by 1.79× compared to the load-spreading policy. The NoMora policy with parameters $p_m = 110$ and $p_r = 115$ improves the median task placement latency by 2.35× compared to the random policy and by 2.69× compared to the load-spreading policy.

In Figure 6.9, it can be seen that the NoMora policy with preemption degrades the task response time, because of longer task placement latencies (Figure 6.8). The CDF is truncated to 10,000s, because the trace includes long-running jobs that span the whole trace.

6.4 Limitations

Root task placement In the current NoMora policy, the root task is placed before the other tasks in any available slot in the cluster. This can negatively impact the overall job performance, but without gang scheduling it is difficult to decide where to place the root task, since what actually

needs to be placed is the whole task graph. With gang scheduling, the root task would not be placed independently of the other tasks, but instead would be placed at the same time. The current implementation considers that a job has only one root task, but it is straightforward to extend it to place multiple root tasks, and to compute the placement of the other tasks based on the best root task placement of all root tasks.

Changes in network traffic NoMora accounts for the latency changes introduced by the traffic from tasks already placed in previous rounds. It does not account for the changes produced by tasks placed in the same round, *e.g.*, the network latency measurements are not updated during the scheduling algorithm runtime. If a task's placement proves to be suboptimal due to the traffic generated by the tasks placed in the same scheduling round, the task can be migrated to a better placement. Still, the scheduler runs in an online fashion, meaning it runs as soon as a task enters the system. If the scheduler algorithm runtime is low, then the number of tasks scheduled in a round will be kept small. If there is a significant number of tasks that arrive concurrently, then the policy may place tasks suboptimally. Gog [Gog17] analysed the workload in the public Google cluster trace to determine how many tasks the scheduler must deal with in a scheduling round depending on how fast the scheduler runs. Gog found that a scheduler algorithm that completes every 0.5s has to process less than ten events in over 60% of cases, and less than 100 events in 95% of cases. NoMora's policy median runtime is less than 100ms (Section 6.3), which means that in most cases the scheduler deals with a small number of tasks per scheduling round.

As the scheduler places more tasks that send traffic, the latency between hosts might increase due to higher network utilisation. Conversely, when tasks finish running, latency between hosts might decrease due to lower network utilisation. The current simulator does not account for these changes in the network's state. A *correction component* can be added to the simulator to model these changes. I note that the traces collected in different data centres over a week already include the changes observed in network latency in a real-world setting, but these changes are not correlated with the arrivals and departures of the jobs in the traces I use for the simulation. Thus, the simulation is subject to the limitations of the cloud measurements collected.

6.5 Summary

In this chapter, I introduced latency-driven, application performance-aware cluster scheduling, and NoMora, a cluster scheduling framework that implements this type of policy. It exploits functions that predict application performance based upon network latency and dynamic network latency measurements between hosts to place tasks in a data centre, providing them with improved application performance. I first gave an overview of cluster scheduling modelled as minimum cost maximum flow optimisation over a flow network, and of the Firmament cluster scheduler framework (§6.1). Next, I presented NoMora, a cluster scheduling architecture, and I described its cost policy (§6.2). I described the metrics for evaluating NoMora, and I presented

the results of the evaluation (§6.3). Finally, I discussed the limitations of my approach (§6.4).

The overall application performance improvement given by NoMora depends on the workload, network topology and on the network conditions in the data centre. Using the Google workload augmented with cloud latency measurements from Chapter 4 and with predictions from Chapter 5, I show that the NoMora policy improves the overall average application performance by up to 13.4% compared to the baselines, and improves the task placement latency by a factor of 1.79× and the median algorithm runtime by 1.16× compared to the baselines (§6.3). This demonstrates that application performance can be improved by exploiting the relationship between network latency and application performance, and the current network conditions in a data centre, while preserving the demands of low-latency cluster scheduling.

Chapter 7

Conclusions and future work

With the growth of the cloud business, it is of paramount importance to provide tenants with predictable application performance. In this dissertation, I made several important contributions towards enabling *application performance-aware data centres* through network measurements by considering the impact of network latency on application performance when scheduling applications on hosts.

- In **Chapter 3**, I investigated through experiments how the Precision Time Protocol (PTP) through an open-source software implementation, PTPd, can be used to measure network conditions, network latency and packet loss.

  I showed that PTPd represents a practical approach for measuring network conditions, offering estimated one-way delay and packet loss ratio measurements. PTPd is a widely available software, easy to deploy, and it should be noted that a hardware-enabled PTP implementation would perform even better.

- In **Chapter 4**, I introduced *PTPmesh*, a network monitoring tool for cloud tenants, which uses the PTPd software as building block. I deployed PTPmesh in several data centres across the world from different cloud providers (Amazon AWS, Google Compute Platform, Microsoft Azure), where I conducted measurement campaigns. I presented a detailed analysis of the measurement data, revealing different latency magnitude, latency variance and packet loss characteristics for data centres from different geographical regions and from different cloud providers. Normal latencies in data centres vary between tens to hundreds of microseconds, while unusual conditions can lead to latencies of milliseconds.

  I showed that PTPmesh, a network monitoring tool for cloud customers, can be deployed in different cloud providers. PTPmesh performed measurements in data centres across different cloud providers, identifying considerable latency variance between data centres.

- In **Chapter 5**, I showed that even small amounts of network latency in the order of tens or hundreds of microseconds can impact negatively the application performance for certain

typical cloud applications (DNS, key-value store, machine learning frameworks). I studied the effect of injecting different amounts of latency in the network on several typical cloud applications' performance through experiments conducted on a custom testbed. I showed that DNS, Memcached, STRADS, and Tensorflow are sensitive to increases in latencies of only tens of microseconds, while Spark is not affected by up to 500μs. I built predictions of application performance dependent upon network latency based on the experimental results.

I showed that network latency is an important factor to consider for the performance of distributed applications of the style used in data centres, and I built functions that predict application performance dependent upon network latency for these applications.

- In **Chapter 6**, I introduced and described the NoMora cluster scheduling architecture, which extends the Firmament cluster scheduling framework. NoMora uses functions that predict the application performance based upon network latency, combined with network latency measurements between pairs of hosts in a data centre offered by a measurement system, to place or migrate applications in order to provide them with improved application performance. I showed using a Google cluster trace that NoMora improves overall average application performance by up to 13.4%, while decreasing task placement latency by a factor of 1.79×.

  I showed that latency-driven, application performance-aware, cluster scheduling has utility in data centres, leading to improvement in overall average application performance.

7.1 Future work

Alongside my contributions, I have also identified a number of opportunities for future work.

7.1.1 Extending PTPmesh

Data storage and analysis The network measurement data collected must be aggregated in order to extract meaningful information. Systems such as Pingmesh [GYX+15] and NetNORAD [ALZ16] have additional components for data storage (Cosmos [CJL+08] in the case of Pingmesh, Scribe [Joh08] in the case of NetNORAD) and analysis (SCOPE [CJL+08] in the case of Pingmesh, Scuba [Abr12] in the case of NetNORAD). Currently, PTPmesh's data processing takes place separately at each client, but the data storage and analysis components of Pingmesh and NetNORAD can be adapted to aggregate and process PTPd data logs.

Furthermore, PTP measurement data could be post-processed to remove the anomalies in the one-way delay values introduced by the PTP convergence time period, offering more accurate one-way delay measurements.

Hierarchical design PTPmesh could be provided as a data centre-wide service. In this case, its design would resemble that of Pingmesh or NetNORAD. The probing scheme would not be linked to tenant networks, but to the underlying physical network infrastructure. It could additionally use PTP-enabled NICs. PTPmesh's design would be hierarchical: probing would take place between the each pair of servers under a ToR, between each pair of ToRs (through designated servers below ToRs), and at a higher level between each pair of data centres (§2.3.4).

Root cause analysis As already stated, determining the root cause of latency spikes, end-host issue or network fault, is difficult [RBB+18]. Access to the virtualisation layer would be needed to discern between these two types of issues. Collecting different types of measurement data, both at the end-host and in-network [ACL+16], could be a promising approach to solving this challenge. Additionally, the use of monitoring algorithms implemented on programmable data planes [PAM17; NSN+17; LMK+16a; LMK+16b] could make troubleshooting the network significantly easier.

7.1.2 Extending the application performance prediction

Measuring application performance under different network conditions A system that benchmarks the application performance under different configurations and network conditions (different latency magnitudes and variances) would help in building a complete application performance profile. Such a system would be used to predict the application performance under different circumstances. The predictions can then be used to inform cloud operators and users in order to find the optimal placement under given network conditions. Such systems have been proposed in the past, which consider the application semantics, application communication pattern, and also network bandwidth used by the application [EDC+09; VCC11; HB11; JBC+12; JKW15; OAB15; VYF+16], but have not included in their predictions the application's network latency demands.

Extending the application performance prediction model By fitting a naive model to the experimental datasets in Section 5.5, I took a first step towards building a more general model. The model should show how different latency magnitudes and variances influence the application performance. More complex models using several features for each application, such as packet size, packet inter-arrival time, flow duration, or burst length [MZ05], could be derived using recent machine learning techniques [XHG18]. Going further, other aspects, such as type of CPU, core count, memory requirements, disk throughput, network bandwidth demands could be incorporated in the model, building a comprehensive application performance model, with the help of the data obtained from a benchmark system as the one described in the previous paragraph.

7.1.3 Extending NoMora

Task constraints The NoMora cost model does not incorporate other constraints, such as task interference, or constraints about other resources needed by the task, such as CPU, memory, disk throughput, or network bandwith. The cost model can be extended with other Firmament cost models to accommodate such constraints. For example, the cost function for determining costs on arcs could compute an aggregate cost over all the constraints related to resources.

Gang scheduling Gang scheduling means all the application's tasks are placed simultaneously. Certain applications (*e.g.*, Spark, Tensorflow, STRADS) whose computation is done by all tasks together would benefit from having all their tasks placed simultaneously, reducing the task wait time and overall job runtime. Firmament supports gang scheduling, where all task nodes are connected to an aggregator node, and this aggregator node is connected to the rest of the flow network as in the usual case.

In the current NoMora policy, the root task is placed first, and the optimal placement of the other tasks is computed based on the placement of the root task. With gang scheduling, all tasks must be placed at the same time, so the cost between the root task and the other tasks is not known beforehand. Thus, the cost determined by the network latency between every two machines in the data centre must be encoded in the flow network. One way to do this would be to have equivalence classes that comprise machines whose network latency between each other is within a certain range. If a job is gang scheduled in such an equivalence class, then all the tasks of the job will be experiencing the network latency of the equivalence class. All the tasks of a job can be gang scheduled in one equivalence class if its size is sufficient, or across multiple equivalence classes. Since the network latency between machines is not constant, the equivalence classes would change frequently, meaning the flow network would also change frequently, which would add additional overhead to computing task placement.

Flow scheduling NoMora can be coupled with approaches like FastPass [POB+14] to choose amongst the network paths that are available to a placed application's packets. Due to the existence of multiple network paths between hosts in data centres, my policy utilises the maximum network latency in the cost computation to be conservative. But once an application is placed, the packets of the application can be assigned one of the paths with the lowest latency amongst the ones that are available to the application using a centralised allocator such as FastPass. FastPass computes the time when each packet will be sent and the path the packet will take. With the network latency on paths known, these values can serve as an additional input for the FastPass computation to chose paths for an application's packets which would offer the best application performance.

Varys [CZS14], Baraat [DKB+14], Aalo [CS15] are systems for coflow scheduling. A coflow is formed of flows that originate from the same task, and whose flow scheduling is done together to improve application performance. These systems can optimise the scheduling of the tasks' coflows after the job's tasks had been placed by NoMora [JBM+15]. Moreover, approaches

such as pFabric [AYS+13] or QJump [GSG+15] can be used to futher improve the performance for certain applications by assigning higher priorities to their network traffic.

Lastly, an integration of the cluster scheduler and of the flow scheduler into a single monolithic scheduling architecture could be designed.

7.2 Concluding remarks

Network latency is important for cloud customers' application performance [MK15], but it is often an unaddressed issue. Recalling the thesis of this dissertation, I sum up the contributions I made:

1. I investigated the use of PTP measurements for estimating one-way delay and packet loss ratio in data centres. I showed that PTPd measurements can be used to measure network conditions in data centres.

2. I presented PTPmesh, a practical and easily deployable monitoring tool for tenants to access one-way delay and packet loss ratio measurements, and I showed that it can be deployed in different cloud providers to gain insights into current network conditions. Furthermore, I conducted a measurement study of network conditions in data centres from multiple cloud providers using PTPmesh, showing differences in latency magnitude, latency variance and packet loss ratios between data centres.

3. I showed that network latency in the order of tens to hundreds of microseconds matters for typical data centre distributed applications, and I built functions that predict application performance dependent upon network latency.

4. I demonstrated notable application performance improvement by combining application performance prediction functions with a network latency measurement infrastructure in a practical cluster scheduling architecture, NoMora.

Bibliography

[AA15] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 0.91. Arpaci-Dusseau Books, May 2015 (cited on page 49).

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. "TensorFlow: A System for Large-scale Machine Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 265–283 (cited on pages 19, 38–39, 73, 113, 117–118).

[ABK+14] Sebastian Angel, Hitesh Ballani, Thomas Karagiannis, Greg O'Shea, and Eno Thereska. "End-to-end Performance Isolation Through Virtual Datacenters". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Broomfield, CO: USENIX Association, 2014, pp. 233–248 (cited on pages 69–70).

[Abr12] Lior Abraham. *Under the hood: Data diving with Scuba*. `https://www.facebook.com/notes/facebook-engineering/under-the-hood-data-diving-with-scuba/10150599692628920/`. [Online; accessed December 2018]. Facebook, Apr. 2012 (cited on page 156).

[ACC+18] Behnaz Arzani, Selim Ciraci, Luiz Chamon, Yibo Zhu, Hongqiang (Harry) Liu, Jitu Padhye, Boon Thau Loo, et al. "007: Democratically Finding the Cause of Packet Drops". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 419–435 (cited on pages 47–48).

[ACL+16] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. "Taking the Blame Game out of Data Centers Operations with NetPoirot". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 440–453 (cited on pages 48, 157).

[ACR+10] Hussam Abu-Libdeh, Paolo Costa, Antony Rowstron, Greg O'Shea, and Austin Donnelly. "Symbiotic Routing in Future Data Centers". In: *SIGCOMM Comput. Commun. Rev.* 40.4 (Aug. 2010), pp. 51–62 (cited on page 37).

[AED+14] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Vinh The Lam, et al. "CONGA: Distributed Congestion-aware Load Balancing for Datacenters". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, pp. 503–514 (cited on page 18).

[AGM+10] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, et al. "Data Center TCP (DCTCP)". In: *Proceedings of the ACM SIGCOMM 2010 Conference*. SIGCOMM '10. New Delhi, India: ACM, 2010, pp. 63–74 (cited on pages 37–40, 42–43, 65, 73).

[AKE+12] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. "Less is More: Trading a Little Bandwidth for Ultra-low Latency in the Data Center". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 19–19 (cited on pages 37, 125).

[ALC+17] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. "Cherrypick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics". In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI'17. Boston, MA, USA: USENIX Association, 2017, pp. 469–482 (cited on page 66).

[ALV08] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. "A Scalable, Commodity Data Center Network Architecture". In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. SIGCOMM '08. Seattle, WA, USA: ACM, 2008, pp. 63–74 (cited on pages 35–36, 61, 90, 144).

[ALZ16] Aijay Adams, Petr Lapukhov, and James Hongyi Zeng. *NetNORAD: Troubleshooting networks via end-to-end probing*. `https://code.facebook.com/posts/1534350660228025/netnorad-troubleshooting-networks-via-end-to-end-probing/`. [Online; accessed December 2018]. 2016 (cited on pages 29, 45–47, 89–90, 139, 156).

[Ama18] Amazon. *Amazon Enhanced Networking*. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/enhanced-networking.html`. [Online; accessed December 2018]. 2018 (cited on page 100).

[And14] Alexey Andreyev. *Introducing data center fabric, the next-generation Facebook data center network*. `https://code.fb.com/production-engineering/introducing-data-center-fabric-the-next-generation-facebook-data-center-network/`. [Online; accessed December 2018]. 2014 (cited on pages 36, 145).

[APG+18] George Amvrosiadis, Jun Woo Park, Gregory R. Ganger, Garth A. Gibson, Elisabeth Baseman, and Nathan DeBardeleben. "On the diversity of cluster workloads and its impact on research results". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 533–546 (cited on page 63).

[ARR+10] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. "Hedera: Dynamic Flow Scheduling for Data Center Networks". In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. NSDI'10. San Jose, California: USENIX Association, 2010, pp. 19–19 (cited on pages 18, 133).

[ASR+10] S. Akoush, R. Sohan, A. Rice, A. W. Moore, and A. Hopper. "Predicting the Performance of Virtual Machine Migration". In: *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. Aug. 2010, pp. 37–46 (cited on page 142).

[AWS] Amazon AWS. *Weave Net Multicast for AWS EC2*. `https://aws.amazon.com/marketplace/pp/B071RMCZ1X`. [Online; accessed December 2018] (cited on page 91).

[AWS18a] Amazon AWS. *Amazon Elastic Block Store*. `https://aws.amazon.com/ebs/`. [Online; accessed December 2018]. 2018 (cited on page 92).

[AWS18b] Amazon AWS. *Announcing General Availability of Amazon EC2 Bare Metal Instances*. `https://aws.amazon.com/about-aws/whats-new/2018/05/announcing-general-availability-of-amazon-ec2-bare-metal-instances/`. [Online; accessed December 2018]. 2018 (cited on page 83).

[AXF+12] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. "Workload Analysis of a Large-scale Key-value Store". In: *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '12. London, England, UK: ACM, 2012, pp. 53–64 (cited on pages 39, 73, 116, 118).

[AYS+13] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. "pFabric: Minimal Near-optimal Datacenter Transport". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: ACM, 2013, pp. 435–446 (cited on pages 18, 65, 133, 159).

[Azu] Microsoft Azure. *Azure Virtual Network frequently asked questions (FAQ)*. `https://docs.microsoft.com/en-us/azure/virtual-network/virtual-networks-faq`. [Online; accessed December 2018] (cited on page 91).

[BAA+11] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. "MicroTE: Fine Grained Traffic Engineering for Data Centers". In: *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*. CoNEXT '11. Tokyo, Japan: ACM, 2011, 8:1–8:12 (cited on pages 18, 133).

[BAC+17] Ilker Nadi Bozkurt, Anthony Aguirre, Balakrishnan Chandrasekaran, P. Brighten Godfrey, Gregory Laughlin, Bruce Maggs, and Ankit Singla. "Why Is the Internet so Slow?!" In: *Passive and Active Measurement: 18th International Conference, PAM 2017, Sydney, NSW, Australia, March 30-31, 2017, Proceedings*. Springer International Publishing, 2017, pp. 173–187 (cited on page 48).

[BAM10] Theophilus Benson, Aditya Akella, and David A. Maltz. "Network Traffic Characteristics of Data Centers in the Wild". In: *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*. IMC '10. Melbourne, Australia: ACM, 2010, pp. 267–280 (cited on pages 40–43).

[Bar14] Luiz André Barroso. "Landheld Computing". In: *IEEE International Solid State Circuits Conference (ISSCC)*. Keynote. 2014 (cited on pages 55, 132).

[BCK+11] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. "Towards Predictable Datacenter Networks". In: *Proceedings of the ACM SIGCOMM 2011 Conference*. SIGCOMM '11. Toronto, Ontario, Canada: ACM, 2011, pp. 242–253 (cited on pages 68–71).

[BDF+03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, et al. "Xen and the Art of Virtualization". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*. SOSP '03. Bolton Landing, NY, USA: ACM, 2003, pp. 164–177 (cited on page 110).

[BEL+14] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, et al. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 285–300 (cited on page 66).

[BGK+13] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, et al. "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: ACM, 2013, pp. 99–110 (cited on page 46).

[BH18] Luiz Andre Barroso and Urs Hoelzle. *The Datacenter As a Computer: Designing Warehouse-Scale Machines, Third Edition*. 3rdst. Morgan and Claypool Publishers, 2018 (cited on page 17).

[BJK+13] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. "Chatty Tenants and the Cloud Network Sharing Problem". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 171–184 (cited on pages 70–71).

[BMP+17] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. "Attack of the Killer Microseconds". In: *Commun. ACM* 60.4 (Mar. 2017), pp. 48–54 (cited on pages 18, 49, 73).

[Bra89] R. Braden, editor. *Requirements for Internet Hosts - Communication Layers*. United States, 1989 (cited on page 23).

[BS10] Sean Kenneth Barker and Prashant Shenoy. "Empirical Evaluation of Latency-sensitive Application Performance in the Cloud". In: *Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*. MMSys '10. Phoenix, Arizona, USA: ACM, 2010, pp. 35–46 (cited on page 113).

[Bur06] Mike Burrows. "The Chubby lock service for loosely-coupled distributed systems". In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2006 (cited on page 38).

[CAK12] Yanpei Chen, Sara Alspaugh, and Randy Katz. "Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads". In: *Proc. VLDB Endow.* 5.12 (Aug. 2012), pp. 1802–1813 (cited on page 40).

[CB06] Kendall Correll and Nick Barendt. "Design Considerations for Software Only Implementations of the IEEE 1588 Precision Time Protocol". In: *In Conference on IEEE 1588 Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*. 2006 (cited on pages 33–34).

[CBA+14] S.R. Chowdhury, M.F. Bari, R. Ahmed, and R. Boutaba. "PayLess: A low cost network monitoring framework for Software Defined Networks". In: *2014 IEEE Network Operations and Management Symposium (NOMS)*. May 2014, pp. 1–9 (cited on page 26).

[CBM+17] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: ACM, 2017, pp. 153–167 (cited on pages 63, 68, 110).

[CBR+15] Paolo Costa, Hitesh Ballani, Kaveh Razavi, and Ian Kash. "R2C2: A Network Stack for Rack-scale Computers". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 551–564 (cited on page 62).

[CCP+16] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, et al. "A Cloud-scale Acceleration Architecture". In: *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-49. Taipei, Taiwan: IEEE Press, 2016, 7:1–7:13 (cited on page 120).

[CDE+12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, et al. "Spanner: Google's Globally-distributed Database". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*. OSDI'12. Hollywood, CA, USA: USENIX Association, 2012, pp. 251–264 (cited on pages 29, 38).

[CFS+90] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin. *Simple Network Management Protocol (SNMP)*. United States, 1990 (cited on pages 24–25).

[Che96] Stuart Cheshire. *It's the Latency, Stupid*. `http://www.stuartcheshire.org/rants/latency.html`. [Online; accessed December 2018]. May 1996 (cited on page 132).

[Cisa] Cisco. *Catalyst Switched Port Analyzer (SPAN) Configuration Example*. `https://www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/10570-41.html`. [Online; accessed December 2018] (cited on page 25).

[Cisb] Cisco. *Cisco IOS IP SLAs Configuration Guide*. `http://www.cisco.com/c/en/us/td/docs/ios/12_4/ip_sla/configuration/guide/hsla_c/hsoverv.html`. [Online; accessed December 2018] (cited on pages 25, 45, 47).

[Cisc] NetFlow Cisco. *NetFlow*. `https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html`. [Online; accessed December 2018] (cited on pages 24–25).

[Cisd] Sampled NetFlow Cisco. *Sampled NetFlow*. `https://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/12s_sanf.html`. [Online; accessed December 2018] (cited on page 25).

[CJL+08] Ronnie Chaiken, Bob Jenkins, Per Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. "SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets". In: *Proc. VLDB Endow.* 1.2 (Aug. 2008), pp. 1265–1276 (cited on pages 47, 156).

[CK06] Mark Crovella and Balachander Krishnamurthy. *Internet Measurement: Infrastructure, Traffic and Applications*. New York, NY, USA: John Wiley & Sons, Inc., 2006 (cited on pages 23–25, 31).

[Clo53] C. Clos. "A study of non-blocking switching networks". In: *The Bell System Technical Journal* 32.2 (Mar. 1953), pp. 406–424 (cited on page 35).

[CR10] Marta Carbone and Luigi Rizzo. "Dummynet Revisited". In: *SIGCOMM Comput. Commun. Rev.* 40.2 (Apr. 2010), pp. 12–20 (cited on page 28).

[CS12] Mosharaf Chowdhury and Ion Stoica. "Coflow: A Networking Abstraction for Cluster Applications". In: *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*. HotNets-XI. Redmond, Washington: ACM, 2012, pp. 31–36 (cited on page 38).

[CS15] Mosharaf Chowdhury and Ion Stoica. "Efficient Coflow Scheduling Without Prior Knowledge". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 393–406 (cited on page 158).

[CZM+11] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I. Jordan, and Ion Stoica. "Managing Data Transfers in Computer Clusters with Orchestra". In: *Proceedings of the ACM SIGCOMM 2011 Conference*. SIGCOMM '11. Toronto, Ontario, Canada: ACM, 2011, pp. 98–109 (cited on page 40).

[CZS14] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. "Efficient Coflow Scheduling with Varys". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, pp. 443–454 (cited on pages 133, 158).

[Dat18] Databricks. *Spark-perf benchmark*. `https://github.com/databricks/spark-perf`. [Online; accessed December 2018]. 2018 (cited on page 117).

[DB13] Jeffrey Dean and Luiz André Barroso. "The Tail at Scale". In: *Commun. ACM* 56.2 (Feb. 2013), pp. 74–80 (cited on pages 71, 100, 103, 132).

[DDD+16] Pamela Delgado, Diego Didona, Florin Dinu, and Willy Zwaenepoel. "Job-aware Scheduling in Eagle: Divide and Stick to Your Probes". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. SoCC '16. Santa Clara, CA, USA: ACM, 2016, pp. 497–509 (cited on page 66).

[DDK+15] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. "Hawk: Hybrid Datacenter Scheduling". In: *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '15. Santa Clara, CA: USENIX Association, 2015, pp. 499–510 (cited on page 66).

[DDS+09] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009 (cited on page 40).

[Dee89] S. E. Deering. *Host Extensions for IP Multicasting*. United States, 1989 (cited on page 33).

[DG01] N. G. Duffield and Matthias Grossglauser. "Trajectory Sampling for Direct Traffic Observation". In: *IEEE/ACM Trans. Netw.* 9.3 (June 2001), pp. 280–292 (cited on page 25).

[DG04] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *OSDI'04: Sixth Symposium on Operating System Design and Implementation*. San Francisco, CA, 2004, pp. 137–150 (cited on pages 38, 125).

[DGG+99] N. G. Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, K. K. Ramakrishnan, and Jacobus E. van der Merive. "A Flexible Model for Resource Management in Virtual Private Networks". In: *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '99. Cambridge, Massachusetts, USA: ACM, 1999, pp. 95–108 (cited on page 68).

[DK13] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. Houston, Texas, USA: ACM, 2013, pp. 77–88 (cited on page 66).

[DK14] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-efficient and QoS-aware Cluster Management". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 127–144 (cited on page 66).

[DKB+14] Fahad R. Dogar, Thomas Karagiannis, Hitesh Ballani, and Antony Rowstron. "Decentralized Task-aware Scheduling for Data Center Networks". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, pp. 431–442 (cited on pages 133, 158).

[DSA+18] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, et al. "Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 373–387 (cited on pages 62, 83, 103, 110).

[EDC+09] Sameh Elnikety, Steven Dropsho, Emmanuel Cecchet, and Willy Zwaenepoel. "Predicting Replicated Database Scalability from Standalone Database Profiling". In: *Proceedings of the 4th ACM European Conference on Computer Systems*. EuroSys '09. Nuremberg, Germany: ACM, 2009, pp. 303–316 (cited on page 157).

[End] Endance. *Endance DAG 9.2SX2 Datasheet*. `https://www.endace.com/dag-9.2sx2-datasheet.pdf`. [Online; accessed December 2018] (cited on page 55).

[Ent16] Tolly Enterprises. *Mellanox Spectrum vs. Broadcom StrataXGS Tomahawk 25GbE & 100GbE Performance Evaluation - Evaluating Consistency & Predictability*. Technical report 216112. 2016 (cited on page 60).

[ERW+14] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. "A study of network stack latency for game servers". In: *13th Annual Workshop on Network and Systems Support for Games*. Dec. 2014, pp. 1–6 (cited on page 127).

[ESn] Lawrence Berkeley National Laboratory ESnet. *iPerf*. `https://iperf.fr/`. [Online; accessed December 2018] (cited on page 25).

[Exa18a] ExaLINK. *ExaLINK 50 - ULTRA LOW LATENCY 50-PORT LAYER 1 MATRIX SWITCH*. `https://exablaze.com/downloads/pdf/ExaLINK-Datasheet_2014.pdf`. [Online; accessed December 2018]. 2018 (cited on page 49).

[Exa18b] ExaLINK. *ExaNIC X10 SUB-MICRO TCP HALF RTT DUAL-PORT 10GBE NETWORK INTERFACE CARD*. `https://exablaze.com/downloads/pdf/ExaNIC_X10_Datasheet_2018_1.1.pdf`. [Online; accessed December 2018]. 2018 (cited on page 49).

[FBK+12] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. "Jockey: Guaranteed Job Latency in Data Parallel Clusters". In: *Proceedings of the 7th ACM European Conference on Computer Systems*. EuroSys '12. Bern, Switzerland: ACM, 2012, pp. 99–112 (cited on page 66).

[FPM+18] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, et al. "Azure Accelerated Networking: SmartNICs in the Public Cloud". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 51–66 (cited on pages 62, 83, 91, 100, 103, 110).

[GAK+14] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. "Multi-resource Packing for Cluster Schedulers". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, pp. 455–466 (cited on page 66).

[GBQ+14] Ali Ghiasi, Rich Baca, Ghiasi Quantum, and LLC Commscope. "Overview of Largest Data Centers". In: *Proc. 802.3 bs Task Force Interim meeting*. May 2014 (cited on page 132).

[GC13] Nicolas Guilbaud and Ross Cartlidge. *Localizing packet loss in a large complex network*. `https://www.nanog.org/meetings/nanog57/presentations/Tuesday/tues.general.GuilbaudCartlidge.Topology.7.pdf`. [Online; accessed December 2018]. 2013 (cited on page 90).

[GCA+16] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. "Altruistic Scheduling in Multi-resource Clusters". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 65–80 (cited on page 66).

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles*. Bolton Landing, NY, 2003, pp. 20–43 (cited on page 38).

[GHC+18] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. "Sonata: Query-driven Streaming Network Telemetry". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: ACM, 2018, pp. 357–371 (cited on page 27).

[Gir] Apache Giraph. *Apache Giraph*. `http://giraph.apache.org/`. [Online; accessed December 2018] (cited on pages 38–39).

[GJK+09] Albert Greenberg, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, Dave Maltz, Parveen Patel, et al. "VL2: A Scalable and Flexible Data Center Network". In: *SIGCOMM*. Association for Computing Machinery, Inc., Aug. 2009 (cited on pages 36–37, 40, 42–43).

[GKP+18] Panagiotis Garefalakis, Konstantinos Karanasos, Peter Pietzuch, Arun Suresh, and Sriram Rao. "Medea: Scheduling of Long Running Applications in Shared Production Clusters". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: ACM, 2018, 4:1–4:13 (cited on page 67).

[GLL+09] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, et al. "BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers". In: *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*. SIGCOMM '09. Barcelona, Spain: ACM, 2009, pp. 63–74 (cited on page 37).

[GLW+10] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, et al. "SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees". In: *Proceedings of the 6th International COnference*. Co-NEXT '10. Philadelphia, Pennsylvania: ACM, 2010, 15:1–15:12 (cited on pages 69–71).

[GLY+18] Yilong Geng, Shiyu Liu, Zi Yin, Ashish Naik, Balaji Prabhakar, Mendel Rosenblum, and Amin Vahdat. "Exploiting a Natural Network Effect for Scalable, Fine-grained Clock Synchronization". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 81–94 (cited on pages 30, 34–35).

[GNK+15] Peter X. Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. "pHost: Distributed Near-optimal Datacenter Transport over Commodity Network Fabric". In: *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*. CoNEXT '15. Heidelberg, Germany: ACM, 2015, 1:1–1:12 (cited on pages 18, 65).

[GNK+16] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, et al. "Network Requirements for Resource Disaggregation". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 249–264 (cited on pages 37, 62, 67).

[Gog17] Ionel Corneliu Gog. "Flexible and efficient computation in large data centres". PhD thesis. University of Cambridge, 2017 (cited on pages 65–66, 137, 152).

[GSG+15] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. "Queues Don'T Matter when You Can JUMP Them!" In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. Oakland, CA: USENIX Association, 2015, pp. 1–14 (cited on pages 18, 65, 67, 70–71, 125, 133, 159).

[GSG+16] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. "Firmament: Fast, Centralized Cluster Scheduling at Scale". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 99–115 (cited on pages 19–20, 65, 67–68, 71, 133–134, 141).

[GSW15] Andrey Goder, Alexey Spiridonov, and Yin Wang. "Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems". In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 459–471 (cited on pages 65–66).

[GVM+11] Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex Snoeren, and Geoffrey M. Voelker. "DieCast: Testing Distributed Systems with an Accurate Scale Model". In: *ACM Trans. Comput. Syst.* 29.2 (May 2011), 4:1–4:48 (cited on page 28).

[GWT+08] Chuanxiong Guo, Haitao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. "Dcell: A Scalable and Fault-tolerant Network Structure for Data Centers". In: *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*. SIGCOMM '08. Seattle, WA, USA: ACM, 2008, pp. 75–86 (cited on page 37).

[GYX+15] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, et al. "Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 139–152 (cited on pages 29, 45–47, 84, 89–90, 96, 139, 156).

[HB11] Herodotos Herodotou and Shivnath Babu. "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs". In: *Proceedings of the VLDB Endowment*. VLDB'11. 2011 (cited on page 157).

[HBB+18] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, et al. "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Feb. 2018, pp. 620–629 (cited on page 39).

[HBB17] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running Dive into the Future of Infrastructure*. 1st. O'Reilly Media, Inc., 2017 (cited on page 67).

[HCG+18] Rob Harrison, Qizhe Cai, Arpit Gupta, and Jennifer Rexford. "Network-Wide Heavy Hitter Detection with Commodity Switches". In: *Proceedings of the Symposium on SDN Research*. SOSR '18. Los Angeles, CA, USA: ACM, 2018, 8:1–8:7 (cited on page 29).

[Hem] Stephen Hemminger. *NetEm - Network Emulator*. `http://man7.org/linux/man-pages/man8/tc-netem.8.html`. [Online; accessed December 2018] (cited on pages 28–29, 84).

[HF15] T. Hoßfeld and M. Fiedler. "The unexpected QoE killer: When the network emulator misshapes traffic and QoE". In: *2015 Seventh International Workshop on Quality of Multimedia Experience (QoMEX)*. May 2015, pp. 1–6 (cited on page 28).

[HHJ+12] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. "Reproducible Network Experiments Using Container-based Emulation". In: *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. CoNEXT '12. Nice, France: ACM, 2012, pp. 253–264 (cited on page 28).

[HKP+11] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. "Augmenting Data Center Networks with Multi-gigabit Wireless Links". In: *Proceedings of the ACM SIGCOMM 2011 Conference*. SIGCOMM '11. Toronto, Ontario, Canada: ACM, 2011, pp. 38–49 (cited on pages 40, 42, 44).

[HKZ+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, et al. "Mesos: A Platform for Fine-grained Resource Sharing in the Data Center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 295–308 (cited on page 66).

[HLB18] Qun Huang, Patrick P. C. Lee, and Yungang Bao. "Sketchlearn: Relieving User Burdens in Approximate Measurement with Automated Statistical Inference". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: ACM, 2018, pp. 576–590 (cited on page 27).

[HLL+18] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, et al. "SnailTrail: Generalizing Critical Paths for Online Analysis of Distributed Dataflows". In: *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, 2018, pp. 95–110 (cited on page 125).

[Hop00] C. Hopps. *Analysis of an Equal-Cost Multi-Path Algorithm*. Technical report. RFC 2992, 2000 (cited on page 36).

[HRA+17] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. "Re-architecting Datacenter Networks and Stacks for Low Latency and High Performance". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: ACM, 2017, pp. 29–42 (cited on pages 18, 38, 65).

[HRF+06] Thomas R. Henderson, Sumit Roy, Sally Floyd, and George F. Riley. "Ns-3 Project Goals". In: *Proceeding from the 2006 Workshop on Ns-2: The IP Network Simulator*. WNS2 '06. Pisa, Italy: ACM, 2006 (cited on page 27).

[HW16] Mukesh Hira and LJ Wobker. *Improving Network Monitoring and Management with Programmable Data Planes*. `http://p4.org/p4/inband-network-telemetry/`. [Online; accessed December 2018]. 2016 (cited on pages 27, 46–47).

[IEE08] IEEE. *IEEE 1588-2008 Precision Time Protocol*. `https://www.nist.gov/el/intelligent-systems-division-73500/introduction-ieee-1588`. [Online; accessed December 2018]. 2008 (cited on pages 25, 30, 32).

[IH08] Teerawat Issariyakul and Ekram Hossain. *Introduction to Network Simulator NS2*. 1st edition. Springer Publishing Company, Incorporated, 2008 (cited on page 27).

[Int18a] Intel. *Intel Disaggregated Servers Drive Data Center Efficiency and Innovation*. https://www.intel.co.uk/content/www/uk/en/it-management/intel-it-best-practices/disaggregated-server-architecture-drives-data-center-efficiency-paper.html. [Online; accessed December 2018]. 2018 (cited on page 37).

[Int18b] Intel. *Intel Rack Scale Design - Data Center Agility At Scale*. https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/rack-scale-design-overview.html. [Online; accessed December 2018]. 2018 (cited on page 62).

[Int18c] Intel. *Intel SSD D3-S4610 SERIES*. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/dc-d3-s4610-series/d3-s4610-3-84tb-2-5inch-3d2.html. [Online; accessed December 2018]. 2018 (cited on page 49).

[Int18d] Intel. *Intel SSD DC P4511 SERIES*. https://www.intel.co.uk/content/www/uk/en/products/memory-storage/solid-state-drives/data-center-ssds/dc-p4511-series/dc-p4511-1tb-m-2-110mm-3d2.html. [Online; accessed December 2018]. 2018 (cited on page 49).

[Int18e] Intel. *World's Most Responsive Data Center SSD*. https://www.intel.co.uk/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-p4800x-brief.pdf. [Online; accessed December 2018]. 2018 (cited on page 49).

[IPC+09] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. "Quincy: Fair Scheduling for Distributed Computing Clusters". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 261–276 (cited on pages 65–67, 71, 134, 136, 141).

[JAM+13] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. "EyeQ: Practical Network Performance Isolation at the Edge". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 297–312 (cited on pages 69–70).

[JBC+12] Virajith Jalaparti, Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. "Bridging the Tenant-provider Gap in Cloud Services". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: ACM, 2012, 10:1–10:14 (cited on page 157).

[JBM+15] Virajith Jalaparti, Peter Bodik, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. "Network-Aware Scheduling for Data-Parallel Jobs: Plan When You Can". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 407–420 (cited on pages 67, 158).

[JCM+16] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, et al. "Morpheus: Towards Automated SLOs for Enterprise Clusters". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Savannah, GA, USA: USENIX Association, 2016, pp. 117–134 (cited on page 66).

[JKW15] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. "Response Time Service Level Agreements for Cloud-hosted Web Applications". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC '15. Kohala Coast, Hawaii: ACM, 2015, pp. 315–328 (cited on page 157).

[JLH+11] Audrius Jurgelionis, Jukka-Pekka Laulajainen, Matti Hirvonen, and Alf Inge Wang. "An empirical study of netem network emulation functionalities". In: *Computer Communications and Networks (ICCCN), 2011 Proceedings of 20th International Conference on*. IEEE. 2011, pp. 1–6 (cited on page 28).

[Joh08] Robert Johnson. *Facebook's Scribe technology now open source*. `https://www.facebook.com/notes/facebook-engineering/facebooks-scribe-technology-now-open-source/32008268919/`. [Online; accessed December 2018]. Facebook, Oct. 2008 (cited on page 156).

[JQC09] Andrew Johnson, Juergen Quittek, and Benoît Claise. *Packet Sampling (PSAMP) Protocol Specifications*. RFC 5476. Mar. 2009 (cited on page 24).

[JSB+15] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. "Silo: Predictable Message Latency in the Cloud". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 435–448 (cited on pages 70–71).

[JYP+17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. Toronto, ON, Canada: ACM, 2017, pp. 1–12 (cited on page 67).

[KHL+16] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. "STRADS: A Distributed Framework for Scheduled Model Parallel Machine Learning". In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: ACM, 2016, 5:1–5:16 (cited on pages 19, 39, 113, 116, 118).

[KHL+18] Jin Kyu Kim, Qirong Ho, Seunghak Lee, Xun Zheng, Wei Dai, Garth A. Gibson, and Eric P. Xing. *STRADS*. `https://github.com/petuum/strads.git`. [Online; accessed December 2018]. 2018 (cited on page 116).

[KJN+15] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C. Stephen Gunn, et al. "BwE: Flexible, Hierarchical Bandwidth Allocation for WAN Distributed Computing". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 1–14 (cited on pages 72, 133).

[KPA+18] Jan Kucera, Diana Andreea Popescu, Gianni Antichi, Jan Korenek, and Andrew W. Moore. "Seek and Push: Detecting Large Traffic Aggregates in the Dataplane". In: *CoRR* abs/1805.05993 (2018). arXiv: `1805.05993` (cited on page 22).

[KPT+12] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. "Chronos: Predictable Low Latency for Data Center Applications". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: ACM, 2012, 9:1–9:14 (cited on pages 38–39, 71, 73).

[KRC+15] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, et al. "Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters". In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. Santa Clara, CA: USENIX Association, 2015, pp. 485–497 (cited on page 66).

[KSG+09] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. "The Nature of Data Center Traffic: Measurements & Analysis". In: *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference*. IMC '09. Chicago, Illinois, USA: ACM, 2009, pp. 202–208 (cited on pages 40–43).

[Lab17] Percona Lab. *TPCC MySQL benchmark*. `https://github.com/Percona-Lab/tpcc-mysql`. [Online; accessed December 2018]. 2017 (cited on pages 53, 117).

[Lab18] NLNet Labs. *NLnet Labs Name Server Daemon*. `https://www.nlnetlabs.nl/projects/nsd/`. [Online; accessed December 2018]. 2018 (cited on page 115).

[LAP+14] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, et al. "Scaling Distributed Machine Learning with the Parameter Server". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 583–598 (cited on pages 38–39).

[LC10] Yann LeCun and Corinna Cortes. *MNIST handwritten digit database*. `http://yann.lecun.com/exdb/mnist/`. [Online; accessed December 2018]. 2010 (cited on pages 40, 117).

[LDG+13] Katrina LaCurts, Shuo Deng, Ameesh Goyal, and Hari Balakrishnan. "Choreo: Network-aware Task Placement for Cloud Applications". In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC '13. Barcelona, Spain: ACM, 2013, pp. 191–204 (cited on page 70).

[Lev09] David Levinthal. *Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors*. Technical report. [Online; accessed December 2018]. 2009 (cited on page 49).

[Lev14] Jacob Leverich. *Mutilate: high-performance memcached load generator*. `https://github.com/leverich/mutilate`. [Online; accessed December 2018]. 2014 (cited on page 116).

[Lid08] James Liddle. *Amazon found every 100ms of latency cost them 1% in sales.* Aug. 2008 (cited on page 48).

[LK14] Jacob Leverich and Christos Kozyrakis. "Reconciling High Server Utilization and Sub-millisecond Quality-of-service". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 4:1–4:14 (cited on pages 39, 116, 119).

[LMB+14] Katrina LaCurts, Jeffrey C. Mogul, Hari Balakrishnan, and Yoshio Turner. "Cicada: Introducing Predictive Guarantees for Cloud Networks". In: *Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing*. HotCloud'14. Philadelphia, PA: USENIX Association, 2014, pp. 14–14 (cited on pages 69–70).

[LMK+16a] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. "FlowRadar: A Better NetFlow for Data Centers". In: *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*. NSDI'16. Santa Clara, CA: USENIX Association, 2016, pp. 311–324 (cited on pages 27, 157).

[LMK+16b] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. "LossRadar: Fast Detection of Lost Packets in Data Center Networks". In: *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '16. Irvine, California, USA: ACM, 2016, pp. 481–495 (cited on pages 27, 46–47, 157).

[LMV+16] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. "One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 101–114 (cited on page 27).

[LRP+12] Vinh The Lam, Sivasankar Radhakrishnan, Rong Pan, Amin Vahdat, and George Varghese. "Netshare and Stochastic Netshare: Predictable Bandwidth Allocation for Data Centers". In: *SIGCOMM Comput. Commun. Rev.* 42.3 (June 2012), pp. 5–11 (cited on page 70).

[LTL+14] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. "Application-driven Bandwidth Guarantees in Datacenters". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, pp. 467–478 (cited on pages 69–71).

[LWS+16] Ki Suh Lee, Han Wang, Vishal Shrivastav, and Hakim Weatherspoon. "Globally Synchronized Time via Datacenter Networks". In: *Proceedings of the 2016 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016 (cited on pages 30, 34).

[MAB+08] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *SIGCOMM Comput. Commun. Rev.* 38.2 (Mar. 2008), pp. 69–74 (cited on page 26).

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. SIGMOD '10. Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146 (cited on pages 38–39).

[MBK+10] Jim Martin, Jack Burbank, William Kasch, and Professor David L. Mills. *Network Time Protocol Version 4: Protocol and Algorithms Specification*. RFC 5905. June 2010 (cited on pages 25, 30).

[Mem18] Memcached. *Memcached*. `https://memcached.org/`. [Online; accessed December 2018]. 2018 (cited on pages 38, 53, 77, 113, 115, 118).

[Mil10] David L. Mills. *Computer Network Time Synchronization: The Network Time Protocol on Earth and in Space, Second Edition*. 2nd. Boca Raton, FL, USA: CRC Press, Inc., 2010 (cited on page 31).

[MK15] Jeffrey C. Mogul and Ramana Rao Kompella. "Inferring the Network Latency Requirements of Cloud Tenants". In: *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems*. HOTOS'15. Switzerland: USENIX Association, 2015, pp. 24–24 (cited on pages 18, 66, 89, 113, 159).

[MLD+15] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, et al. "TIMELY: RTT-based Congestion Control for the Datacenter". In: *Proceedings of the 2015 ACM Conference on Special*

Interest Group on Data Communication. SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 537–550 (cited on page 49).

[Moc87] P. Mockapetris. *RFC 1035 Domain Names - Implementation and Specification*. `http://tools.ietf.org/html/rfc1035`. [Online; accessed December 2018]. Internet Engineering Task Force, Nov. 1987 (cited on pages 113, 118).

[MPZ10] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. "Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement". In: *Proceedings of the 29th Conference on Information Communications*. INFOCOM'10. San Diego, California, USA: IEEE Press, 2010, pp. 1154–1162 (cited on pages 40, 42, 44, 71).

[MWH14] Ilias Marinos, Robert N.M. Watson, and Mark Handley. "Network Stack Specialization for Performance". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. ACM, 2014, pp. 175–186 (cited on page 113).

[MYG+14] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. "DREAM: Dynamic Resource Allocation for Software-defined Measurement". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, pp. 419–430 (cited on pages 26–27, 29).

[MYG+16] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. "Trumpet: Timely and Precise Triggers in Data Centers". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 129–143 (cited on page 29).

[MZ05] Andrew W. Moore and Denis Zuev. "Internet Traffic Classification Using Bayesian Analysis Techniques". In: *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '05. Banff, Alberta, Canada: ACM, 2005, pp. 50–60 (cited on page 157).

[NAZ+18] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. "Understanding PCIe Performance for End Host Networking". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: ACM, 2018, pp. 327–341 (cited on pages 49, 91).

[Neta] Arista Networks. *Arista 7124FX Application Switch*. `https://www.arista.com/assets/data/pdf/7124FX/7124FX_Data_Sheet.pdf`. [Online; accessed December 2018] (cited on page 49).

[Netb] Arista Networks. *Arista 7500R Series Data Center Switch Router*. `https://www.arista.com/assets/data/pdf/Datasheets/7500RDataSheet.pdf`. [Online; accessed December 2018] (cited on pages 49, 60).

[Nom18] Nominium. *DNSPerf*. `https://www.akamai.com/us/en/products/network-operator/measurement-tools.jsp`. [Online; accessed December 2018]. 2018 (cited on page 115).

[NPF+09] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, et al. "PortLand: A Scalable Fault-tolerant Layer 2 Data Center Network Fabric". In: *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*. SIGCOMM '09. Barcelona, Spain: ACM, 2009, pp. 39–50 (cited on page 37).

[NR09] Lucas Nussbaum and Olivier Richard. "A Comparative Study of Network Link Emulators". In: *Proceedings of the 2009 Spring Simulation Multiconference*. SpringSim '09. San Diego, California: Society for Computer Simulation International, 2009, 85:1–85:8 (cited on page 28).

[NSN+17] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, et al. "Language-Directed Hardware Design for Network Performance Monitoring". In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '17. Los Angeles, CA, USA: ACM, 2017, pp. 85–98 (cited on pages 46–47, 157).

[NZM17] Christos Nikolaou, Noa Zilberman, and Andrew W Moore. "Characterization of Network Tools for Traffic Generation and Traffic Capture". In: *Proceedings of ACM Internet Measurement Conference (IMC)*. ACM. 2017 (cited on pages 20, 28).

[OAB15] L. Ortiz, V. de Almeida, and M. Balazinska. "Changing the Face of Database Cloud Services with Personalized Service Level Agreements". In: *Proceedings of the International Conference on Innovative Data Systems Research (CIDR)*. CIDR 2015. 2015 (cited on page 157).

[OLS08] Patrick Ohly, David N. Lombard, and Kevin B. Stanton. "Hardware Assisted Precision Time Protocol. Design and case study". In: *Proc. of the 9th LCI International Conference on High-Performance Clustered Computing. Intel GmbH*. 2008 (cited on pages 33, 84).

[Opt] Net Optics. *Net Optics Network Taps*. `http://www.nextgigsystems.com/net_optics/10_100_1000BaseT_Tap.html`. [Online; accessed December 2018] (cited on page 55).

[Ora] Oracle. *Oracle VM VirtualBox*. [Online; accessed December 2018] (cited on page 57).

[Ora18] Oracle. *MySQL*. `https://www.mysql.com/`. [Online; accessed December 2018]. 2018 (cited on page 53).

[ORR+15] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. "Making Sense of Performance in Data Analytics Frameworks". In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. NSDI'15. Oakland, CA: USENIX Association, 2015, pp. 293–307 (cited on pages 123, 125).

[OWZ+13] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. "Sparrow: Distributed, Low Latency Scheduling". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: ACM, 2013, pp. 69–84 (cited on pages 66, 68).

[PAM17] Diana Andreea Popescu, Gianni Antichi, and Andrew W. Moore. "Enabling Fast Hierarchical Heavy Hitter Detection Using Programmable Data Planes". In: *Proceedings of the Symposium on SDN Research*. SOSR '17. Santa Clara, CA, USA: ACM, 2017, pp. 191–192 (cited on pages 22, 27, 157).

[Pao10] G. Paoloni. *How to benchmark code execution times on Intel IA-32 and IA-64 instruction set architectures*. Technical Report 324264-001. Intel, 2010 (cited on pages 49, 56).

[Pax06] Vern Paxson. "End-to-end Routing Behavior in the Internet". In: *SIGCOMM Comput. Commun. Rev.* 36.5 (Oct. 2006), pp. 41–56 (cited on page 73).

[PBC+18] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. "Optimus: An Efficient Dynamic Resource Scheduler for Deep Learning Clusters". In: *Proceedings of the Thirteenth EuroSys Conference*. EuroSys '18. Porto, Portugal: ACM, 2018, 3:1–3:14 (cited on page 66).

[PCV+13] Cristel Pelsser, Luca Cittadini, Stefano Vissicchio, and Randy Bush. "From Paris to Tokyo: On the Suitability of Ping to Measure Latency". In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC '13. Barcelona, Spain: ACM, 2013, pp. 427–432 (cited on page 77).

[PEA+96] Bradford W. Parkinson, Per Enge, Penina Axelrad, and James J. Spilker Jr. *Global Positioning System: Theory and Applications, Volume II*. American Institute of Aeronautics and Astronautics, 1996 (cited on page 34).

[PG16] Diana Andreea Popescu and Rogelio Tomas Garcia. "Multivariate Polynomial Multiplication on GPU". In: *Procedia Computer Science* 80 (2016). International Conference on Computational Science 2016, ICCS 2016, 6-8 June 2016, San Diego, California, USA, pp. 154–165 (cited on page 22).

[PH83] J. Postel and K. Harrenstien. *Time Protocol*. RFC 868. May 1983 (cited on page 30).

[PKC+12] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion 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*. SIGCOMM ’12. Helsinki, Finland: ACM, 2012, pp. 187–198 (cited on page 70).

[Pla] Google Cloud Platform. *Virtual Private Cloud (VPC) Network Overview*. `https://cloud.google.com/vpc/docs/vpc`. [Online; accessed December 2018] (cited on page 91).

[Plu82] D. C. Plummer. *Ethernet Address Resolution Protocol: Or Converting Network Protocol Addresses to 48.Bit Ethernet Address for Transmission on Ethernet Hardware*. United States, 1982 (cited on page 37).

[PM15] Diana Andreea Popescu and Andrew W. Moore. “Omniscient: Towards realizing near real-time data center network traffic maps”. In: *Proceedings of the Symposium on SDN Research*. CoNEXT Student Workshop ’15. Heidelberg, Germany: ACM, 2015 (cited on pages 22, 46).

[PM16] Diana Andreea Popescu and Andrew W. Moore. “Reproducing Network Experiments in a Time-controlled Emulation Environment”. In: *Proceedings of The 8th International Workshop on Traffic Monitoring and Analysis (TMA 2016)*. TMA ’16. Louvain-La-Neuve, Belgium: IFIP, 2016 (cited on page 22).

[PM17] Diana Andreea Popescu and Andrew W. Moore. “PTPmesh: Data Center Network Latency Measurements Using PTP”. In: *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Sept. 2017, pp. 73–79 (cited on pages 21, 47, 139).

[PM18] Diana Andreea Popescu and Andrew W. Moore. “A First Look at Data Center Network Condition Through The Eyes of PTPmesh”. In: *2018 Network Traffic Measurement and Analysis Conference (TMA)*. June 2018, pp. 1–8 (cited on page 21).

[PMB+15a] V. Persico, P. Marchetta, A. Botta, and A. Pescape. “On Network Throughput Variability in Microsoft Azure Cloud”. In: *2015 IEEE Global Communications Conference (GLOBECOM)*. Dec. 2015, pp. 1–6 (cited on page 68).

[PMB+15b] Valerio Persico, Pietro Marchetta, Alessio Botta, and Antonio Pescapè. “Measuring Network Throughput in the Cloud”. In: *Comput. Netw.* 93.P3 (Dec. 2015), pp. 408–422 (cited on page 68).

[POB+14] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. “Fastpass: A Centralized "Zero-queue" Datacenter Network”. In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM ’14. Chicago, Illinois, USA: ACM, 2014, pp. 307–318 (cited on pages 18, 38, 65, 133, 158).

[Pos81] J. Postel. *Internet Protocol*. `https://rfc-editor.org/rfc/rfc791.txt`. Sept. 1981 (cited on page 23).

[PRM14] Dimosthenis Pediaditakis, Charalampos Rotsos, and Andrew William Moore. "Faithful Reproduction of Network Experiments". In: *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ANCS '14. Los Angeles, California, USA: ACM, 2014, pp. 41–52 (cited on page 28).

[Proa] Apache HTTP Server Project. *ab - Apache HTTP server benchmarking tool*. `https://httpd.apache.org/docs/2.4/programs/ab.html`. [Online; accessed December 2018] (cited on pages 53, 117).

[Prob] libMemcached Project. *memaslap - Load testing and benchmarking a server*. `http://docs.libmemcached.org/bin/memaslap.html`. [Online; accessed December 2018] (cited on pages 53, 58, 77, 79).

[Pro18] Apache HTTP Server Project. *Apache HTTP Server Project*. `https://www.mysql.com/`. [Online; accessed December 2018]. 2018 (cited on pages 38, 53).

[PTP18] PTPd. *PTP daemon*. `https://github.com/ptpd/ptpd`. [Online; accessed December 2018]. 2018 (cited on pages 18–19, 34, 38, 73, 89–90, 95).

[PYB+13] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C. Mogul, Yoshio Turner, and Jose Renato Santos. "ElasticSwitch: Practical Work-conserving Bandwidth Guarantees for Cloud Computing". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. SIGCOMM '13. Hong Kong, China: ACM, 2013, pp. 351–362 (cited on pages 69–70).

[PYW+17] Yanghua Peng, Ji Yang, Chuan Wu, Chuanxiong Guo, Chengchen Hu, and Zongpeng Li. "deTector: a Topology-aware Monitoring System for Data Center Networks". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 55–68 (cited on pages 46–47).

[PZM17] Diana Andreea Popescu, Noa Zilberman, and Andrew W. Moore. *Characterizing the impact of network latency on cloud-based applications' performance*. Technical report UCAM-CL-TR-914. University of Cambridge, Computer Laboratory, Nov. 2017 (cited on page 22).

[RBB+18] Arjun Roy, Deepak Bansal, David Brumley, Harish Kumar Chandrappa, Parag Sharma, Rishabh Tewari, Behnaz Arzani, et al. "Cloud Datacenter SDN Monitoring: Experiences and Challenges". In: *Proceedings of the Internet Measurement Conference 2018*. IMC '18. Boston, MA, USA: ACM, 2018, pp. 464–470 (cited on pages 18, 45–47, 100, 110, 144, 157).

[RL95] Y. Rekhter and T. Li. *A Border Gateway Protocol 4 (BGP-4)*. United States, 1995 (cited on page 37).

[ROS+11] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. "It's Time for Low Latency". In: *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*. HotOS'13. Napa, California: USENIX Association, 2011, pp. 11–11 (cited on page 132).

[RSD+14] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, et al. "Planck: Millisecond-scale Monitoring and Control for Commodity Networks". In: *Proceedings of the 2014 ACM Conference on SIGCOMM*. SIGCOMM '14. Chicago, Illinois, USA: ACM, 2014, pp. 407–418 (cited on page 27).

[RST+11] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. "Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks". In: *Proceedings of the 3rd Conference on I/O Virtualization*. WIOV'11. Portland, OR: USENIX Association, 2011, pp. 6–6 (cited on pages 69–70).

[RTG+12] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: ACM, 2012, 7:1–7:13 (cited on pages 21, 63, 144).

[RZB+15] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C. Snoeren. "Inside the Social Network's (Datacenter) Network". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. London, United Kingdom: ACM, 2015, pp. 123–137 (cited on pages 40–42, 44).

[RZB+17] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, and Alex C. Snoeren. "Passive Realtime Datacenter Fault Detection and Localization". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 595–612 (cited on pages 48, 90).

[SA99] Tom Shanley and Don Anderson. *PCI System Architecture (4th Ed.)* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999 (cited on page 57).

[SCG+14] Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. "The Internet at the Speed of Light". In: *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. HotNets-XIII. Los Angeles, CA, USA: ACM, 2014, 1:1–1:7 (cited on page 48).

[SCH+11] Bikash Sharma, Victor Chudnovsky, Joseph L. Hellerstein, Rasekh Rifaat, and Chita R. Das. "Modeling and Synthesizing Task Placement Constraints in Google Compute Clusters". In: *Proceedings of the 2Nd ACM Symposium on Cloud Computing*. SOCC '11. Cascais, Portugal: ACM, 2011, 3:1–3:14 (cited on page 67).

[Sch16] Malte Schwarzkopf. "Operating system support for warehouse-scale computing". PhD thesis. University of Cambridge, 2016 (cited on pages 65, 136).

[SCK+09] A. J. Su, D. R. Choffnes, A. Kuzmanovic, and F. E. Bustamante. "Drafting Behind Akamai: Inferring Network Conditions Based on CDN Redirections". In: *IEEE/ACM Transactions on Networking* 17.6 (Dec. 2009), pp. 1752–1765 (cited on page 48).

[sFl] Consortium sFlow. *sFlow*. `https://sflow.org/`. [Online; accessed December 2018] (cited on pages 24–25).

[SHC+18] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 69–87 (cited on pages 37, 67).

[SHP+12] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. "Jellyfish: Networking Data Centers Randomly". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. NSDI'12. San Jose, CA: USENIX Association, 2012, pp. 17–17 (cited on page 37).

[SKA+13] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: Flexible, Scalable Schedulers for Large Compute Clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. Prague, Czech Republic: ACM, 2013, pp. 351–364 (cited on page 66).

[SKD+14] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. "OpenSample: A Low-Latency, Sampling-Based Measurement Platform for Commodity SDN". In: *2014 IEEE 34th International Conference on Distributed Computing Systems (ICDCS)*. June 2014, pp. 228–237 (cited on page 27).

[SKG+11] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. "Sharing the Data Center Network". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*. NSDI'11. Boston, MA: USENIX Association, 2011, pp. 309–322 (cited on page 70).

[SMA+10] Junaid Shaikh, Tahir Nawaz Minhas, Patrik Arlos, and Markus Fiedler. "Evaluation of delay performance of traffic shapers". In: *Security and Communication Networks (IWSCN), 2010 2nd International Workshop on*. IEEE. 2010, pp. 1–8 (cited on page 28).

[SNR+17] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S. Muthukrishnan, and Jennifer Rexford. "Heavy-Hitter Detection Entirely in the Data Plane". In: *Proceedings of the Symposium on SDN Research*. SOSR '17. Santa Clara, CA, USA: ACM, 2017, pp. 164–176 (cited on page 27).

[SOA+15] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, et al. "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network". In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 183–197 (cited on pages 36–37, 62, 67–68).

[Sola] Solarflare. *OpenOnload*. `https://support.solarflare.com/index.php/component/cognidox/?view=categories&id=361`. [Online; accessed December 2018] (cited on page 60).

[Solb] Solarflare. *Solarflare PTP Adapters*. `http://www.solarflare.com/ptp-adapters`. [Online; accessed December 2018] (cited on pages 33, 74, 85).

[Spa] Apache Spark. *Apache Spark MLLib*. `https://spark.apache.org/`. [Online; accessed December 2018] (cited on pages 19, 113, 117–118).

[Sto] Apache Storm. *Apache Storm*. `http://storm.apache.org/`. [Online; accessed December 2018] (cited on page 38).

[Str13] Stephen D. Strowes. *Passively Measuring TCP Round-trip Times*. `https://queue.acm.org/detail.cfm?id=2539132`. 2013 (cited on page 45).

[SWH06] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. "Open Versus Closed: A Cautionary Tale". In: *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*. NSDI'06. San Jose, CA: USENIX Association, 2006, pp. 18–18 (cited on page 116).

[TAL15] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. "CherryPick: Tracing Packet Trajectory in Software-defined Datacenter Networks". In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. SOSR '15. Santa Clara, California: ACM, 2015, 23:1–23:7 (cited on pages 46, 90).

[TAL16] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. "Simplifying Datacenter Network Debugging with PathDump". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 233–248 (cited on pages 47–48).

[TGG10] Amin Tootoonchian, Monia Ghobadi, and Yashar Ganjali. "OpenTM: Traffic Matrix Estimator for OpenFlow Networks". In: *Proceedings of the 11th International Conference on Passive and Active Measurement*. PAM'10. Zurich, Switzerland: Springer-Verlag, 2010, pp. 201–210 (cited on page 26).

[Tib11] Robert Tibshirani. "Regression shrinkage and selection via the lasso: a retrospective". In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 73.3 (2011), pp. 273–282 (cited on page 116).

[Tim] Timekeeper. *TimeKeeper*. `http://www.fsmlabs.com/timekeeper`. [Online; accessed December 2018] (cited on page 34).

[TZP+16] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. "TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters". In: *Proceedings of the Eleventh European Conference on Computer Systems*. EuroSys '16. London, United Kingdom: ACM, 2016, 35:1–35:16 (cited on pages 65–67).

[VCC11] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. "ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments". In: *Proceedings of the 8th ACM International Conference on Autonomic Computing*. ICAC '11. Karlsruhe, Germany: ACM, 2011, pp. 235–244 (cited on page 157).

[VH08] András Varga and Rudolf Hornig. "An Overview of the OMNeT++ Simulation Environment". In: *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*. Simutools '08. Marseille, France: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, 60:1–60:10 (cited on page 27).

[VHV12] Balajee Vamanan, Jahangir Hasan, and T.N. Vijaykumar. "Deadline-aware Datacenter TCP (D2TCP)". In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '12. Helsinki, Finland: ACM, 2012, pp. 115–126 (cited on page 37).

[VMD+13] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, et al. "Apache Hadoop YARN: Yet Another Resource Negotiator". In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC '13. Santa Clara, California: ACM, 2013, 5:1–5:16 (cited on page 66).

[VPK+15] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale Cluster Management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: ACM, 2015, 18:1–18:17 (cited on pages 38, 65–66).

[VSD+16] Asaf Valadarsky, Gal Shahaf, Michael Dinitz, and Michael Schapira. "Xpander: Towards Optimal-Performance Datacenters". In: *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '16. Irvine, California, USA: ACM, 2016, pp. 205–219 (cited on page 37).

[VYF+16] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. "Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics". In: *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*. NSDI'16. Santa Clara, CA: USENIX Association, 2016, pp. 363–378 (cited on pages 66, 157).

[WN10] Guohui Wang and T. S. Eugene Ng. "The Impact of Virtualization on Network Performance of Amazon EC2 Data Center". In: *Proceedings of the 29th Conference on Information Communications*. INFOCOM'10. San Diego, California, USA: IEEE Press, 2010, pp. 1163–1171 (cited on pages 77, 113).

[XDH+12] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Kompella. "The Only Constant is Change: Incorporating Time-varying Network Reservations in Data Centers". In: *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '12. Helsinki, Finland: ACM, 2012, pp. 199–210 (cited on pages 69–71).

[XHD+15] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, et al. "Petuum: A New Platform for Distributed Machine Learning on Big Data". In: *IEEE Transactions on Big Data* 1.2 (June 2015), pp. 49–67 (cited on pages 39, 125).

[XHG18] Shihan Xiao, Dongdong He, and Zhibo Gong. "Deep-Q: Traffic-driven QoS Inference Using Deep Generative Network". In: *Proceedings of the 2018 Workshop on Network Meets AI & ML*. NetAI'18. Budapest, Hungary: ACM, 2018, pp. 67–73 (cited on page 157).

[XLJ+14] Fei Xu, Fangming Liu, Hai Jin, and Athanasios V Vasilakos. "Managing performance overhead of virtual machines in cloud computing: A survey, state of the art, and future directions". In: *Proceedings of the IEEE* 102.1 (2014), pp. 11–31 (cited on page 113).

[XMN+13] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. "Bobtail: Avoiding Long Tails in the Cloud". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 329–342 (cited on pages 71, 113).

[YJL+18] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, et al. "Elastic Sketch: Adaptive and Fast Network-wide Measurements". In: *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. SIGCOMM '18. Budapest, Hungary: ACM, 2018, pp. 561–575 (cited on page 27).

[YJM13] Minlan Yu, Lavanya Jose, and Rui Miao. "Software Defined Traffic Measurement with OpenSketch". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi'13. Lombard, IL: USENIX Association, 2013, pp. 29–42 (cited on page 26).

[YLS+15] Curtis Yu, Cristian Lumezanu, Abhishek Sharma, Qiang Xu, Guofei Jiang, and Harsha V. Madhyastha. "Software-Defined Latency Monitoring in Data Center Networks". In: *Passive and Active Measurement: 16th International Conference, PAM 2015, New York, NY, USA, March 19-20, 2015, Proceedings*. Edited by Jelena Mirkovic and Yong Liu. Cham: Springer International Publishing, 2015, pp. 360–372 (cited on pages 46–47).

[YLZ+13] Curtis Yu, Cristian Lumezanu, Yueping Zhang, Vishal Singh, Guofei Jiang, and Harsha V. Madhyastha. "FlowSense: Monitoring Network Utilization with Zero Measurement Cost". In: *Proceedings of the 14th International Conference on Passive and Active Measurement*. PAM'13. Hong Kong, China: Springer-Verlag, 2013, pp. 31–41 (cited on page 26).

[ZAC+14] Noa Zilberman, Yury Audzevich, G.Adam Covington, and Andrew W. Moore. "NetFPGA SUME: Toward 100 Gbps as Research Commodity". In: *IEEE Micro* 34.5 (Sept. 2014), pp. 32–41 (cited on pages 29, 57, 123).

[ZBH16] Timothy Zhu, Daniel S. Berger, and Mor Harchol-Balter. "SNC-Meister: Admitting More Tenants with Tail Latency SLOs". In: *Proceedings of the Seventh ACM Symposium on Cloud Computing*. SoCC '16. Santa Clara, CA, USA: ACM, 2016, pp. 374–387 (cited on pages 70–71).

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28 (cited on pages 38–39).

[ZDM+12] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. "DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks". In: *SIGCOMM Comput. Commun. Rev.* 42.4 (Aug. 2012), pp. 139–150 (cited on page 71).

[ZGP+17] Noa Zilberman, Matthew Grosvenor, Diana Andreea Popescu, Neelakandan Manihatty-Bojan, Gianni Antichi, Marcin Wojcik, and Andrew W. Moore. "Where Has My Time Gone?" In: *Proceedings of the 18th International Conference on Passive and Active Measurement*. PAM 2017. Sydney, Australia, 2017 (cited on pages 21, 50, 55–56, 58, 61, 76–77, 121, 127).

[ZKC+15] Yibo Zhu, Nanxi Kang, Jiaxin Cao, Albert Greenberg, Guohan Lu, Ratul Mahajan, Dave Maltz, et al. “Packet-Level Telemetry in Large Datacenter Networks”. In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM ’15. London, United Kingdom: ACM, 2015, pp. 479–491 (cited on pages 45, 47).

[ZLZ+17] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. “High-resolution Measurement of Data Center Microbursts”. In: *Proceedings of the 2017 Internet Measurement Conference*. IMC ’17. London, United Kingdom: ACM, 2017, pp. 78–85 (cited on page 48).

[ZTH+13] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. “CPI2: CPU Performance Isolation for Shared Compute Clusters”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys ’13. ACM, 2013, pp. 379–391 (cited on page 113).

[ZTZ+14] Junlan Zhou, Malveeka Tewari, Min Zhu, Abdul Kabbani, Leon Poutievski, Arjun Singh, and Amin Vahdat. “WCMP: Weighted Cost Multipathing for Improved Fairness in Data Centers”. In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys ’14. Amsterdam, The Netherlands: ACM, 2014, 5:1–5:14 (cited on page 73).