Enhancing Programmability for Adaptive Resource Management in Next Generation Data Centre Networks

Simon Jouet

Submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

School of Computing Science

College of Science and Engineering

University of Glasgow

October 2017

Abstract

Recently, Data Centre (DC) infrastructures have been growing rapidly to support a wide range of emerging services, and provide the underlying connectivity and compute resources that facilitate the "*-as-a-Service" model. This has led to the deployment of a multitude of services multiplexed over few, very large-scale centralised infrastructures. In order to cope with the ebb and flow of users, services and traffic, infrastructures have been provisioned for peak-demand resulting in the average utilisation of resources to be low. This overprovisionning has been further motivated by the complexity in predicting traffic demands over diverse timescales and the stringent economic impact of outages. At the same time, the emergence of Software Defined Networking (SDN), is offering new means to monitor and manage the network infrastructure to address this underutilisation.

This dissertation aims to show how measurement-based resource management can improve performance and resource utilisation by adaptively tuning the infrastructure to the changing operating conditions. To achieve this dynamicity, the infrastructure must be able to centrally monitor, notify and react based on the current operating state, from per-packet dynamics to longstanding traffic trends and topological changes. However, the management and orchestration abilities of current SDN realisations is too limiting and must evolve for next generation networks. The current focus has been on logically centralising the routing and forwarding decisions. However, in order to achieve the necessary fine-grained insight, the data plane of the individual device must be programmable to collect and disseminate the metrics of interest.

The results of this work demonstrates that a logically centralised controller can dynamically collect and measure network operating metrics to subsequently compute and disseminate fine-tuned environment-specific settings. They show how this approach can prevent TCP throughput incast collapse and improve TCP performance by an order of magnitude for partition-aggregate traffic patterns. Futhermore, the paradigm is generalised to show the benefits for other services widely used in DCs such as, e.g, routing, telemetry, and security.

Acknowledgements

Behind this thesis there has been many years of discussion, work and collaboration with incredible people, the following is an acknowledgement and thank you to those people.

First and foremost my thanks and profound appreciation go to my supervisor Dimitrios P. Pezaros who has provided continous feedback, support, motivation and much more. Over the years I have constantly learned from him both technically and personally in a manner and dedication that exceeded all expectations.

To my friends and officemates Kyle White and Richard Cziva for their friendship, input, collaboration and the occasional Friday beers.

I am grateful to the Scottish Informatics & Computer Science Alliance (SICSA) for funding the first couple of years of this research.

Finally, to my partner Ornela for the love, support and understanding during some tough and stressful times.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Outline

Internet Service Providers and Data Centres have been growing extremely rapidly since the early 2000s to support a wide new range of services that are now ubiquitous such as, VoIP, VoD, and modern online services and platforms. This growth in services has required the network infrastructure to quickly expand to provide much higher aggregate throughput, lower latencies, higher resilience, and increased compute capacity. At the same time, the competition between providers and operators has resulted in a drive to provide fast and reliable services as cost efficiently as possible. These user-driven business incentives have led network operators to look into better ways to manage and orchestrate their services in order to reduce operational expenditure, and improve the utilisation of existing resources to improve return on investment. However, achieving these goals is complex in large-scale environments while maintaining interoperability with unknown devices relying on legacy suboptimal protocols.

The management challenges of such large infrastructures are multifold and mainly relate to the extremely large number of devices that need to be continously monitored, managed, and orchestrated. In order to provide a competitive service to the end users, the cost of the infrastructure must be kept as low as possible by utilising the available resources as efficiently as possible. However, improving resource utilisation can be challenging when dealing with nu-

merous servers, changing applications, and complex network topologies while maintaining high availability and resilience. All of these challenges are difficult tasks on their own, making Data Centre (DC) management extremely complex, especially when failures can have substential economic impact. To tackle these problems, operators have quickly refined the design of the infrastructure, for example with Facebook now deploying their third generation of data centres in only 13 years of existence [1, 2].

Management of resources is critical in such large-scale infrastructures in order to continuously monitor the demand and dynamically adapt to maximise resource utilisation. However, current infrastructures provide little to no support for collecting the metrics required, resulting in existing systems having a very limited view of their current operating conditions across the different layers of the architecture. The problem is further amplified by the varying time-scale and frequency at which these metrics must be computed and collected, from sub-millisecond flow arrival times to yearly topological upgrades. This limited insight prevents providers from dynamically tuning and adapting applications and services to the current operating conditions of the infrastructure, resulting in the underutilisation of resources.

In order to facilitate this management and orchestration, and allow operators to better control the resources available in their infrastructure, both industry and academia have been heavily investigating new or improved systems since the early 1990s. The road to obtain what is now known as Software Defined Networking has been long, requiring about two decades of research to finally see large-scale deployments of partially programmable infrastructures. With the deployment of OpenFlow as the first realisation of SDN, a large new range of management functions have been available to the network operators, highlighting the benefits of a centralised management system with a complete overarching view over the infrastructure. This SDN architecture has been a strong shift away from the legacy networks designed to be fully autonomous and distributed. SDN has allowed network operators to manage the network infrastructure, opening up control on how packets and flows should be handled across multiple devices.

This dissertation investigates how this programmability and newly available insight into the

network can be leveraged in DC environments to tackle the underutilisation of resources and allow for the deployment of new services. Using OpenFlow, today's leading implementation of SDN, this work highlights the benefits of logically centralised network resource management schemes. With increased control over the network infrastructure, a centralised controller can maintain a global view of the network state, the current utilisation, the communication patterns, and therefore make globally informed decisions based on all available metrics. These decisions can provide a new understanding of the global dynamics of the architecture that are otherwise unavailable when observing the behaviour from independent endpoints. The benefits of the proposed approach are highlighted through Omniscient TCP (OTCP), a novel approach for tackling TCP incast thoughput collapse. TCP incast collapse stems from the partition-aggregate nature of DCs that results in gross underutilisation of the network resources and extremelly low application performance. Through OTCP, this work aims to show that significant improvement in resource utilisation can be obtained when globally-informed decisions are made. However, current SDN realisations are too limiting to improve resource utilisation and through OTCP this work demonstrates these limitations.

To further improve resource utilisation, the programmability of the current infrastructure must be further expanded to provide insight to the service and application providers. Through a programmable interface, the metrics of interest at the various layers of the infrastructure could be made available. Using these metrics, services can be dynamically adapted and modified to the changing operating environment. This approach would allow the centralised control plane to manage resources based on a wide range of information and metrics representing the current operting conditions. Through this approach, optimisation and improvement from the physical up to the application layer can be performed.

1.2 Thesis Statement

Hypothesis: *The deployment of a programmable data plane in data centre SDN infrastructures will enable operators to improve network utilisation by dynamically provisionning the network parameters based on temporal demand.*

1.3 Contributions

This work contributes to the abstraction, development, and paradigm shift necessary for next-generation networking architectures through:

- A comprehensive review of the past and current approaches of programmable networks and the motivation for increasing the programmability of the network infrastructure.
- An in-depth analysis of Data Centre (DC) network topologies with associated cost, complexity, deployability and resilience and the rationale for programmable networks in such environments.
- An investigation of the impact of existing and legacy resource management schemes on DCs network performance and maximum utilisation.
- Multiple experimental evaluations of the benefits of environment specific fine-tuning of TCP congestion control parameters on network performance.
- An implementation demonstrating the benefits and feasibility of a centralised SDN approach for dynamic management, orchestration, and tuning of the environment based on the full view of the network operating conditions.
- A discussion of the current limitations of today's SDN implementations and the resulting hurdles for dynamic resource allocation, management and orchestration.
- The design and implementation of a novel SDN framework addressing existing SDN limitations and promoting data plane programmability for fine-grained, high-performance packet processing and metrics collection.
- A platform, protocol, and language-independent packet processing pipeline leveraging the eBPF instruction set, and the introduction of a simple control plane protocol and controller software.

- A large collection of use cases and example implementations for data plane programmability, from packet forwarding, to telemetry, to debugging and middlebox-like functions such as, e.g., anomaly detection.

1.4 Organisation of the Thesis

The work presented in this thesis is structured as follows:

- **Chapter 2:** covers the motivation for the creation and development of Software Defined Networking, and the trade-offs that have been made to achieve current deployment. It then describes the current problems of very large-scale Data Centres and the incentive for Network Operators to leverage this newly available programmability.

- **Chapter 3:** describes the current resource management issues of modern Data Centres and in particular the issue of TCP incast throughput collapse that results in gross underutilisation of the network. It then introduces Omniscient TCP (OTCP), a logically centralised SDN approach to collect, measure, and compute environment specific operating parameters. Through extensive evaluation, OTCP demonstrates that centralised network resource management can improve performance and network utilisation by an order of magnitude.

- **Chapter 4:** highlights, through OTCP, the limitations of today's SDN implementations and the resulting barriers in resource management and orchestration. It associates these limitations to the current very limited programmability and control and introduces a novel SDN framework for data plane programmability. The framework design and implementation is presented in detail, and a wide range of use-cases that are unrealisable or impractical with current SDN implementations are demonstrated.

- **Chapter 5:** summarises the work and contributions of this dissertation and discusses future research directions for this work.

- **Appendix A:** provides source listing of example data plane functions and the associated controller logic for the proposed framework.

Chapter 2

Related Work

2.1 Outline

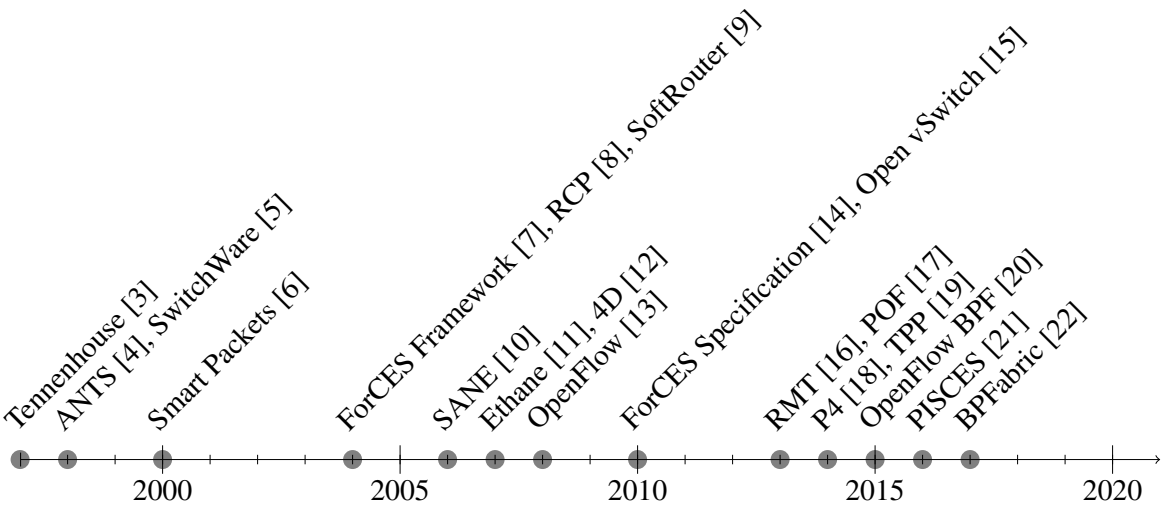


Figure 2.1: Selected key contributions to programmable networks, from Active Network to SDN.

User-driven business incentives has led the network operators to look into better ways to manage and orchestrate their services in order to reduce operational expenditure, and improve the utilisation of the existing resources to improve return on investment. However, achieving these goals is complex in an environment the size of the Internet with an extremely large number of endpoints and many providers with different incentives. This wide diversity, requires interoperability between unknown devices preventing rapid deployment of new technologies resulting in a legacy of suboptimal protocols.

In order to facilitate management and orchestration, and allow the operators to better control the resources available, both industry and academia have been heavily investigating new or improved systems since the early 1990s. The road to obtain what is now known as Software Defined Networking has been long, requiring about two decades of research to finally see large-scale deployments of partially programmable infrastructures. With the deployment of OpenFlow as the first realisation of SDN, a large new range of management functions have been available to the network operators, highlighting the benefits of a centralised management system with a complete overarching view over the infrastructure. This SDN architecture has been a strong shift away from the legacy networks designed to be fully autonomous and distributed. SDN has allowed network operators to manage the network infrastructure, opening up control on how packets and flows should be handled across multiple devices.

In this chapter, the research leading to SDN is presented, detailing the limitations of current implementations and the necessary steps to move forward. The operating environment and issues in DCs are then described, presenting legacy and modern approaches as well as clean-slate designs. Finally, the current limitations in management and orchestration are discussed, highlighting the problem space of this thesis. Figure 2.1 shows a timelime and evolution of the notable work in the area of programmable networks.

2.2 Towards a Centralised Software Defined Architecture

2.2.1 From legacy networks to SDN

The Internet has evolved from an experimental packet-switched network to the worlwide infrastructure we are familiar with today. This very rapid growth and reliability requirements associated with the need for all the endpoints to interact, resulted in many aspects of the infrastructure to be “set in stone”. This phenomenon has been retrospectively called the *ossification* of the Internet and mostly refers to the inability to transition away from the

Internet Protocol version 4 (IPv4) as the addressing architecture, and the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP) at the transport level [23, 24]. This ossification has been highlighted by the difficulties in deploying new technologies such as IP multicast at Internet-scale and has been mostly relegated to internal use by service providers to deliver IPTV or VoIP services over the infrastructure they maintain. Similarly, the IPv6 protocol has seen a very slow deployment rate, with less than 15% of the worldwide traffic despite being a standard for almost 20 years [25, 26].

At the transport layer, the ossification of the Internet has prevented the deployment of some more efficient protocols, such as TCP Vegas, a variant of TCP with a better congestion control algorithm which suffers unfairness when collocated with more greedy TCP variants. The deployment of newer variants of TCP has been possible under the condition that they operate at the same level of greediness as already deployed alternatives, such as TCP Cubic, TCP Westwood and very recently TCP BBR [27, 28, 29]. Newer transport protocols such as the Stream Control Transmission Protocol (SCTP) and the Datagram Congestion Control Protocol (DCCP) have been shown to provide improvements over TCP and UDP for some applications, but cannot be readily deployed over the existing infrastructure [30]. The infrastructure has been designed with TCP or UDP over IP, and most middleboxes and Network Address Translators (NATs) do not support newer protocols like SCTP resulting in the frames to be dropped or corrupted as they traverse the network [30, 31].

The ossification has also extended to the API available to the developers from the operating system. The POSIX socket API was a minor evolution of the Berkeley socket API and is now the *de facto* standard for application developers. However, the limited abstraction offered by the API, prevents the deployment and adoption of newer protocols without a significant development effort [32]. As a result, transport protocol designers have proposed the development of application-layer transport protocols using existing transport layers, like UDP, as a substrate. Some of the most notable realisations of application-layer transport protocols have been QUIC and SPDY for web traffic and to some extent HTTP/2 that performs custom flow control at the application layer [33, 34, 35]. These protocols are pragmatic

approaches to overcome ossification but they violate the protocol layering principle and require the transport protocol to be implemented for each application reducing consistency, maintainability and significantly increasing the developement effort [36].

Although the ossification of the network infrastructure has prevented worldwide adoption and deployment of new, more robust or efficient protocols, network operators have an incentive for deploying new services and pushing the capabilities of their infrastructure. With customers' demand for very high-throughput and low-latency required for today's cloud applications and services, video streaming, realtime communication and much more. To economically support this growth in demand, network operators are being pushed towards a better utilisation of the available resources. This economic incentive is two-fold: the first is to reduce the Capital Expenditure (CAPEX) of the operator by better using the infrastructure already in place; the second is to reduce Operational Expenditure (OPEX) to maintain the proper operation of the network. While reducing CAPEX and OPEX are economic objectives, they have been driving the evolution of very large networks over the years. To increase the network utilisation and therefore reduce CAPEX, the network must be designed to operate properly with suboptimal network protocols as well as provide a minimum guaranteed level of operation and service during peak demand. To reduce OPEX, the day-to-day management and control of the infrastructure and its resources must be simplified and streamlined with the ability to provide enough information for operators to quickly and successfully detect and remediate from faults or attacks.

To achieve this goal of higher utilisation and simplified network management, the network must evolve to move away from a simple autonomous forwarding plane to a more programmable system allowing user control and modification of the network processing and forwarding behaviour. In such scenario, it should be possible to steer traffic in a fine-grained manner to use any suitable routes and optimise the traffic allocation throughout a large-scale infrastructure. Through simplified routing mechanisms, it then becomes simpler to better leverage the fabric to deploy new functions and policies helping in reducing OPEX and CAPEX. Network devices providing caching, filtering, load balancing, anomaly detec-

tion and much more can then be deployed throughout the fabric and used on-demand by customizing the routing path for the traffic of interest. Using such approach, the network operator can better use underutilised links and hence reduce CAPEX while providing an improved service to the end-users. Another consequence of providing fine-grained configuration over the routing and forwarding of the infrastructure is the possible introduction of overlay networks. Overlay networks allow a physical network to be spliced into multiple smaller virtual networks that share some of the nodes and available resources. This splicing allows the network to be better utilised by allocating a specific subset of nodes and partial resources to a specific function, and allowing traffic isolation without requiring a seperation of the infrastructure. Virtualised overlay networks share the same objectives as virtual machines, the former being designed to share virtual network over a physical infrastructure, while the later provide virtual compute resources over physical machines.

However, this evolution from a standard simple autonomous forwarding plane to a more programmable system is a complex and difficult task. Computer networks are designed with a wide variety of equiment, mostly routers, switches and end-hosts but also a multitude of middleboxes. All of these devices are able to interoperate but must be configured individually by the network admninistrators using vendor-specific interfaces, often different across products from the same vendor. The multiple efforts in evolving the network have been focussing in removing this barrier in configuring the large amount of devices without being tied to a specific vendor, and to allow this programmability to operate at a higher level of abstraction than a specific network protocol, function or mechanism.

In the early 1990s, the first large-scale attempt at evolving the network was conducted by researchers employing *Active Networking* [3]. Active networking was a very different approach to network management and control from traditional networks in which network nodes are able to expose a set of resources such as processing, storage, packet queuing mechanisms through a *network API* [37]. Using this network API, individual network nodes could be dynamically altered to implement new network functions for the subset of packets passing through the node. Active networking research proposed two programming models,

the *capsule model* [4] and the *programmable switch model* [38, 5]. In the capsule model, the code to execute was carried in-band within data packets, while in the programmable switch model the code to execute was transmitted out-of-band. Through the evolution and popularisation of active networking, the capsule model became the most commonly associated programming model to active networks. Performance was not the main focus of the research community, but some efforts aimed to design high-performance active routers using multiple collocated network processors overcoming the limitations of traditional single core systems [39]. The security aspect was mostly overlooked, like many early research efforts of that time, although the Secure Active Network Environment Architecture was an exception aiming for type safety, signing and dynamic checks of the programs and authentication and authorisation of the users [40]. The widespread deployment and implementation of active networks was unsuccesful due to numerous factors. One of the most notable issues has been the rift between two sides of the network community in whether the network should be kept simple to facilitate its growth or "smarter" to evolve it past its current state. This rift was amplified by the unclear path to deployment for Active networks and a lack of immediate need for such programmability. These aspects as well as misconceptions about active networks, mostly regarding security and performance prevented a strong traction for large-scale deployments. Nonetheless, active networks offered many intellectual contributions to the community, such as the need for a unified architecture for device orchestration, the benefits of programmability within the network to facilitate innovation, and the capabilities of performing operator-defined actions based on packet headers.

In the early 2000s, with an Internet infrastructure mostly in place and a large increase in traffic volume, network operators shifted their focus away from deployment towards better performance, higher reliability and predictability of the infrastructure behaviour. These objectives led the operators to overprovision the network with aggregate bandwidth far superior to the peak demand. This gross overprovisioning of resources was a very simple, although costly, solution to the increasing demand. Subsequently, to leverage this large amount of bandwidth, providers focussed on optimizing network management functions to control the paths used to carry traffic, a concept nowadays called Traffic Engineering (TE). The lim-

ited routing protocols available at the time, coupled with the tight integration between the control and data planes of the network devices made network management tasks such as custom routing, debugging, remote monitoring (telemetry) extremely lengthy and difficult [41]. These limitations became a frustration to the network operators in the endehavour required to manage large-scale and continuously growing complex networks. At the same time, the quick improvements in processor speed and memory in general purpose computers resulted in recently deployed switches' to be greatly underpowered compared to commodity computers.

These frustrations and the ability to implement new functions in the resources available in generic architectures resulted in two innovations: an *open interface* between the control and data plane and the motivation for a *logical centralised control plane*. As a result of these efforts, the ForCES (Forwarding and Control Element Separation) open interface was developed and standardised by the Internet Engineering Task Force (IETF) [7, 14]. To logically control networks, architectures such as the Routing Control Platform (RCP) and SoftRouter were proposed [42, 9, 8]. This work focussed on network management problems and how to allow network administrators to have network-wide visibility and programmability over the control plane. This was following the same line as active networks but was mostly designed to be quickly deployed by and for network operators instead of the end users or the research community. As a result, multiple logically centralised control planes were developed based on open-source routing software that simplified the development and deployment of prototype implementations. This first step in decoupling the control and data planes was a realistic and short-term solution for the problems network operators were facing but was a significant departure from the conventional tightly coupled and autonomous design of the Internet. As a result, many of the concepts developed at the time have been leveraged for subsequent SDN designs such as a *logically centralised control plane* communicating with the devices data plane over an *open interface*.

Network vendors had little incentive to adopt an open API like ForCES as it exposes internal aspects of the devices and allows for the competition to provide comparable products. It is

in the vendors' best interest to maintain a closed API with no direct comptabile competition, and create a vendor lock-in, making the switch to another vendor a costly operation. This business incentive was possible due to the lack of large deployments of programmable networks that could clearly show to the operators the benefits and therefore incentivise a paradigm shift by the network vendors. As a result, and due to the pragmatic approach in decoupling the control and data plane, the control plane communication protocol was relying on existing routing protocols [43], greatly limiting the range of applications that the logically centralised control plane could support and implement. The research community, without the same short-term incentive, built on this separation of planes and proposed new clean-slate architectures for a centralised control plane. The 4D project [12] envisaged a further breakdown of the planes with a 4 layer approach: the data plane being responsible for forwarding the packets, the discovery plane for collecting measurements and topology, the dissemination plane to install the packet processing rules, and the decision plane to centrally convert global rules into per device packet rules. The 4D approach with the many different layer, was able to provide for the first time a logical abstraction of the network model that allowed areas beyond routing to be explored.

The last step for this evolution from legacy networks to Software Defined Networks (SDN) has been to think of this separation of planes more broadly than solely in the context of packet forwarding. Up to this point and based on the main motivation of the providers to leverage more flexible routing, programmability was mostly limited to traffic engineering to better use the available infrastructure. Ethane and its predecessor SANE have been the stepping stone to SDN by proposing to use this separation of control plane and the centralised view of the network to implement different functions other than routing [11, 10]. They have proposed to use a logically centralised flow-level architecture to implement access control policies throughout the infrastructure. In the Ethane proposal, the switches are represented as a set of flow tables managed and populated by the controller based on a set of security policies, clearly separating the data plane behaviour and the control plane functions. Ethane became the foundation of OpenFlow with the first implementation of the Ethane switch design becoming the original OpenFlow API [13].

2.2.2 OpenFlow

Software Defined Networking emerged in mid-2000s as a means for researchers to experiment on medium to large-scale infrastructures, such as the Global Environment for Networking Innovation (GENI) [44], EU FIRE testbeds, and at smaller scale, within campus networks. The concept of SDN has been to provide programmability into the network to better operate and manage the infrastructure. However, the networking community has been split between a desire for highly programmable networks and pragmatism for achieving programmability on existing infratructures. OpenFlow was proposed by the Stanford Clean Slate Program [45] as a balanced compromise between programmability and pragmatism. The instrinsic switch design of OpenFlow, inherited from Ethane, leverages the packet processing hardware available in commodity switches while providing more programmability than earlier proposals. This design decision has been the main reason for OpenFlow to become the first widely deployed implementation of programmable networks by allowing vendors to provide programmability without the need to redesign the underlying hardware. However, this pragmatism in relying on existing hardware did limit the overall flexibility of the design as well as the capabilities of the overall proposal.

The principle of operation of OpenFlow is straightforward but it is necessary to understand the separation of concerns between the individual switches within the infrastructure and the central controller. The switches in OpenFlow are designed to be "dumb" or, more specifically, designed to delegate all decision making to the central controller in case they are unaware of what should be done with incoming packets. In contrast, a legacy switch would make a node-local decision, for instance, an ethernet learning switch will flood the packet to all other ports. In order to delegate the decision making, the switches' traditional control plane, consisting of the peering mechanisms and self-learning functions is replaced with an OpenFlow agent as shown in Figure 2.2. This OpenFlow agent is responsible for establishing a persistent TCP connection with the OpenFlow controller and exchanging messages following the OpenFlow API. Using the OpenFlow API, the agent in the switches is able to expose the internal state of each switch to the central controller and consequently the controller is

able to query and update each switch internal state remotely. The role of the controller is then to rely on the global view of the network generated from the aggregated switches' internal states to make informed decisions on how packets should be matched, processed, and forwarded. In order to perform this decision making, a specific controller application must be designed that is capable of understanding, replying and emitting OpenFlow API messages.

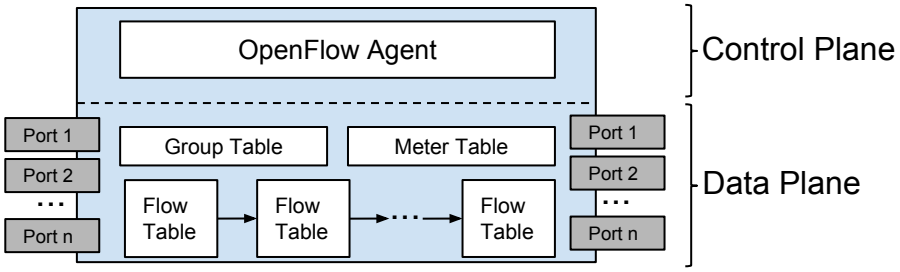


Figure 2.2: OpenFlow Switch architecture and data/control plane separation

In an OpenFlow switch, the device is composed of one or more flow tables each holding flow rules defining how packets are handled. Each flow rule defines a pattern on which packets are matched, a set of actions to be performed when a packet matches this pattern, a fixed set of statistics counters, and a priority to disambiguate between overlapping patterns. The pattern in each flow rule defines the subset of header fields in the packets that should be matched from the available fields defined in the OpenFlow specification. The initial production version of OpenFlow included 12 possible match fields, allowing flow rules to match on packet header fields from the physical layer (input port) to the transport layer (UDP, TCP). To cope with the demand for new functions to be implemented using OpenFlow, each new version of the protocol added support for new fields summing up to 44 supported fields in OpenFlow 1.5. For each packet matching the pattern of the flow, a set of actions is executed that allows header fields to be modified, routing headers to be altered and, most importantly, for an output action to be taken. Using this simple match-action approach, an OpenFlow network works as follows: a packet arrives at the ingress of an OpenFlow switch, and the highest priority flow rule is fetched from the flow table resulting in either a match or a table miss; in the former case, the associated flow rule actions are executed and the counters are incremented; in case of a table miss the corresponding packet is sent to the controller through the OpenFlow API *packet_in* message, the controller decides what should be done with this

packet and other subsequent similar packets, and inserts new flow rules in the switch.

In order to match the flow rules with the packet header, OpenFlow is able to perform two types of matching that are directly associated with the hardware available in typical networking hardware. The first matching type is a strict equality matching between the packet header and entries in memory. This matching is commonly implemented in hardware for very high speed lookup using Content Addressable Memory (CAM). In commodity switches, the CAM is traditionally used by the self-learning functions to maintain the mapping between the physical input port on which packets are received and the associated MAC address of the device sending the packet. The second type of matching uses Ternary Content Addressable Memory (TCAM) which follows the same principle as CAM but instead of relying only on exact bit values, it allows for a third "don't care" bit state which is matched regardless of its value in memory. Using TCAM, the matching can be much more fine-grained allowing entries in memory to partially match some of the packet headers while still providing very high speed lookups. TCAM is commonly used in commodity routers to perform layer 3 Longest Prefix Match (LPM) on IP entries and provide routing entries for different subnets. However, TCAM is expensive in terms of chip area requirement, power consumption and speed of operation, therefore it is often limited in size to a few thousands of entries [46]. OpenFlow leverages the CAM and TCAM available in off-the-shelf network devices to perform the flow rule matching, and can therefore be deployed purely in software relying on the existing available hardware. This wide availability of compatible network devices and the limited development cost of supporting existing and already deployed network devices open up the adoption of SDN in large-scale networks.

Using the SDN approach of delegating the decision making to the central controller and allowing the overall behaviour of the network to be defined in a single application written with a global view of the network infrastructure, many new research directions can be explored. These new research opportunities are supported by the spread of packet headers match field from layer 1 to layer 4, blurring the traditional separation between switches and routers. In an SDN environment, the devices are either switches or routers. Depending on the flow rules

installed and the behaviour of the controller either functionalitty can be implemented. By leveraging this programmability and the flexbility in centrally managing the behaviour of the network, a wide range of new network research has emerged over the last decade using OpenFlow as the foundation for large-scale experiments. OpenFlow was able to help research in domains such as routing, traffic engineering, quality of service, congestion control, protocol design, network function virtualization, virtual machine placement and much more. It followed on many principles from previous work on programmable networks, but made significant contributions to the networking community allowing for the first time large-scale deployments of programmable infrastructure in both academic and industrial networks. Possibly the main contribution has been to unify many different types of network devices by allowing actions to be performed from layer 1 to layer 4 instead of focussing only on a particular layer of the stack.

Early large-scale deployments and use-cases such as its initial deployment in the Stanford campus [13], its use in Google's private wide-area backbone for traffic management and optimisation [47], as well as Nicira's network Virtualization Platform [48] gathered significant industry attention. These commercial successes highlighted the strong potential of OpenFlow and more broadly SDN, resulting in significant efforts to study, evolve and standardise SDN. As a result, multiple large-scale initiatives by some of the largest information-technology companies including cloud providers, Internet Service Providers (ISPs), national research networks and equipment vendors joined SDN consortia such as the Open Networking Foundation (Microsoft, Google, Huawei, Telefonica, Brocade, etc.) or the Open Daylight initiative (IBM, Cisco, Arista Networks, etc.).

SDN and OpenFlow, have been criticised strongly by some members of the networking community sometimes due to legitimate considerations but often due to misconceptions. One of the major misconceptions has been regarding the centralisation of the controller and the safety and resilience concerns of doing so. SDN advocates for a *logically* centralised control plane and does not imply a physical centralisation of the infrastructure that would result in a single point of failure. In fact, many large-scale deployments of OpenFlow now rely on

a distributed system of controllers that can cope with multiple failures as demonstrated by systems like the Onix and ONOS controllers, and wide-area deployments of SDN such as Google's private backbone [49, 50]. A second misconception, is the need for the first packet of every flow to be sent to the controller and hence being only viable in low volume traffic environments. This misconception is mostly due to early implementation such as in Ethane or example proof-of-concept controller applications that rely on this approach. However, through the wide set of match fields, the granularity can be adapted to the traffic characteristics and flow rules can be configured to provide routing for large aggregates of traffic without involving the controller. The approach by which the controller deploys the flow rules is implementation-specific and can be done pro-actively when the topology changes, resulting in no packet sent to the controller during normal operation or reactively, when a new flow is established.

Although OpenFlow became the most successful implementation of SDN, it is critical to understand that OpenFlow and SDN are not identical concepts and that OpenFlow is merely one partial implementation of the broader SDN concept. Previous work of configuring the network infrastructure using active networks can be considered retrospectively as instantiations of SDN. More recently, competing approaches from other vendors, such as Cisco ONE [51], the JunOS SDK [52] and NetConf/YANG [53] represent other realisations of the SDN concept that are incompatible with OpenFlow. SDN conceptually offers a much broader programmability of the network than what OpenFlow or other realisations currently provide. In today's approaches, the network programmability, orchestration and management is very limited. OpenFlow offers some limited programmability in flow matching and packet processing but still considers the data plane of the devices as a blackbox. At a higher level, OpenFlow does not provide mechanisms to interface with systems such as end-hosts, hypervisors, virtual machines (VM) or applications, limiting cross-layer optimisation and resource management.

2.2.3 Beyond OpenFlow

OpenFlow has been critical in the evolution of SDN and the wide acceptance by research and industry of the benefits of exposing the network state through an open API. In the last 10 years, it has changed the way we think about network management and orchestration, solving problems such as network virtualization and slicing, and opening up new directions in many areas of network and systems research. However, OpenFlow, due to its pragmatic design choice to support existing hardware, has been lacking flexibility, requiring constant updates, and customisation, and rendering seemingly simple use-cases impractical or unfeasible [20]. To cope with the demand, OpenFlow added some support for new match-fields and actions as well as for new features such as port groups, counters and thresholds. However, these changes have been unable to address the wide range of scenarios required by the network operators and the features provided by the vendors, resulting in a very large number of bespoke OpenFlow extensions. Through these extensions, controller applications have been able to achieve particular behaviour on specific switch implementations but lost the vendor-agnostic nature of the controller applications returning to some partial vendor lock-in. The continuous changes to the OpenFlow specification as well as the effort by network vendors to support the extensions and the OpenFlow agent, have resulted in most of the networking community to implicitly agree on OpenFlow 1.3 as the long term supported (LTS) version of the protocol. As a result, even though OpenFlow 1.4 and 1.5 are over two years old, the hardware support for these versions is very limited and most implementations are not fully compliant with the specifications.

OpenFlow acted as a stepping-stone in showcasing the benefits of SDN by relying on existing hardware ASICs, and was a necessary step in the evolution of networks but, now that the benefits are evident, this pragmatic approach can be challenged. To support the next generation of networks and offer flexible programmability over the network to provide a wide new range of middlebox-like functions such as Deep Packet Inspection (DPI), packet reassembly, telemetry, protocol offloading, stateful routing and much more, it is necessary to explore different approaches. This transition away from tradditional networking ASICs is helped by the

recent deployment of new devices such as Network Processors (NPU), Field Programmable Gate Array (FPGA), and the increased use of software switches. To transition away from OpenFlow to the next generation of SDN architecture, two approaches have been trending: (i) a top-down view of the network where the focus is on the network operator requirements and how these requirements can be expressed; and (ii) a bottom-up view focussing on the switch design.

To address the match-field programmability issue in OpenFlow, switch architectures such as the reconfigurable match table (RMT) [16] model, and commercial chips such as Intel's FlexPipe [54] and Cavium's Xpliant [55] have been suggested. These three chips follow a match-action pipeline that can be reconfigured dynamically to match arbitrary packet headers while providing performance comparable to fixed-functions chips. The RMT architecture is designed to create an arbitrary pipeline based on the programmer's desired forwarding behaviour. This pipeline is built from a flexible parser able to match on arbitrary packet headers, and an arbitrary arrangement of match stages with associated SRAM and TCAM memory allocated based on the requirements of the function. To provide the arbitrary matching and packet editing, RMT relies on a very-long instruction word (VLIW) instruction set to parallelise the extraction, comparison, and modification of specific headers in each packet. Intel's FlexPipe approach is simpler and relies on a 32-stage pipeline with allocated TCAM to perform the packet matching and a very limited instruction set to perform actions on the matched packets. These designs have been able to show that modern hardware design can be used to perform flexible matching on network traffic without impacting network performance. The aggregate throughput achieved using RMT is close to 1Tb/s and Cavium's highest end ASIC is 3.2Tb/s.

To express this architectural flexibility and allow network operators to define the data plane function, domain specific languages (DSL), such as P4 and POF, have been suggested [18, 17]. P4's main design goals have been to provide platform independence, allowing the same P4 program to be deployed across multiple hardware or software switches and protocol independent to allow any new or custom network protocol to be supported. To achieve this, it

abstracts the switch model and provides syntax to define the set of protocol headers that are expected, the parser rules, the primitive actions, the tables to allocate and the control flow to define the order of the matching. Protocol-Oblivious Forwarding (POF) follows the same goal as P4 in providing a protocol and platform independent programming model for the data plane. However, POF does so with a very limited set of low-level programming instruction, akin to an assembly language. This approach allows the compilation, parsing and processing of the data plane functionality to be kept simple but increases the complexity of deployment.

To investigate data plane programmability in other architectures than the controller-centric one made popular by OpenFlow, new research has been suggested. Tiny Packet Program (TPP) [19], returns to the concept of capsule-based active networks and proposes to embed lightweight instructions into the packet content to query and manipulate the internal state of the network. Through this design, each packet can embed at most 5 instructions and each switch traversed will forward and execute the embedded program. The main concept of TPP is to separate the concern and responsibility, the switches are reponsible for high performance forwarding and limited execution while the end hosts perform arbitrary computation on the network state by introducing TPP programs. PISCES [21] has been another influential work in this domain, which moves away from the hardware infrastructure to focus only on the hypervisor-level software switches. PISCES justify this decision due to the very large number of software switches deployed in modern infrastructure, often one per host, resulting in more software than hardware switches in modern virtualised infrastructures.

The security aspects of SDN are an important part of the design as any attack on the control or data plane could disrupt the robustness and reliability of the network. However, current SDN deployments are mostly limited to privately owned infrastructures, such as data centres, with a separate and isolated management network carrying the management information. This isolated network prevent most of the attack vectors by physically preventing outside access to a critical part of the infrastructure. Existing deployments mostly rely on transport encryption and endpoints authorisation through TLS certificates to ensure that a listener on the network or a third-party is not able to capture or issue unauthorised requests

to the controller and network devices. A potential attack vector in a cloud environment is when an attacker manages a privilege escalation outside the virtual machine and gains access to the hypervisor and therefore the management network. However, the potential attack on the controller would be very limited as most communication is from the controller to the hypervisor's virtual switch, not the other way around and much more lucrative and faster attacks are possible at this point, such as accessing the other tenants' data and traffic.

With the significant popularity of SDN, and newer deployments focusing on service provider networks that do not provide the same level of isolation, research has been proposed to set the requirements for a secure SDN network [56, 57]. The security policies in SDNSEC [56] attempts to safeguard the SDN resources that may jeopardize the good behaviour of the network. In particular, it promotes the same security considerations for an SDN controller applications than any other software by assessing the behaviour of the software and detecting any exploit, vulnerability or reliability issues before deployment. The key aspect of the proposal is for authentication and authorisation to be used when the management of the network spans over multiple organisations at which point security by isolation is not possible anymore.

2.3 Data Centre Network Architecture

2.3.1 SDN in Data Centres

In this thesis the research and analysis is focused on Data Centre infrastructures as they have been suffering many of the same difficulties and limitations as ISP networks, due to their initial design decisions, but also offer new challenges and opportunities to address those problems. DC infrastructure have been growing much more quickly than ISP networks over the last 15 years, to provide end-users with "*-as-a-Service" platforms. This very rapid growth has resulted in modern Data Centres to house hundreds of thousands of nodes and links in very dense interconnects, with traffic volume and communication patterns orders of magnitude higher than modern ISP networks. These large-scale networks, with high vol-

umes of traffic, low latency and high throughput associated to new traffic patterns generated by DC-specific services, have resulted in a large number of novel issues. However, even though DCs are facing a myriad of difficulties, they are operated very differently from ISP networks opening up new ways to tackle these issues. DCs, in opposition to ISP networks, are managed and operated by a single authoritative entity that has complete control over the infrastructure. This single ownership opens new possibilities in management and orchestration as the complete infrastructure can be managed uniformly and paradigm shifts can be made at much shorter time scales than typically possible with ISP providers.

The network infrastructure of modern DCs have evolved beyond ISP networks in order to provide the resources and resilience required. In backbone Autonomous Systems (ASs), the connectivity is provided and managed in large traffic aggregates over link bundles providing high-capacity, but low redundancy in the case of failure. In contrast, modern DC topologies have evolved towards rich and redundant networks, with a large variety of links and paths between servers. This difference in traffic aggregates, has also changed the granularity at which ISP and DC operators are managing the traffic; the former deals in large aggregates of traffic between peering endpoints over a very limited set of high-volume, over-provisioned links, while the later micro-manages and schedules the individual flows to take advantage of the different paths. Through this fine grain scheduling of the network resources, DC operators are able to better use the resources available, allowing high performance over non over-provisioned networks. However, existing network protocols designed for the Internet are unable to use this rich path redundancy, as it was a non-goal based on the ISP network infrastructures. Therefore, in order to achieve this fine-grained management of the network resources, better insight into the network behaviour and control over per-flow forwarding is necessary. This requirement for control and programmability coupled with the ability for the DC operators to experiment with new design and technologies has led to the deployment of SDN infrastructures.

Alongside, the fine-grained management of the traffic, DC operators have had other incentives to deploy SDN in their infrastructures. With the advent of virtualization and its

prominance in DC networks to better use the compute resources, the separation between the network and the application is getting blurier. Modern servers are able to host dozens of VMs and containers have allowed hundreds of applications to be collocated in isolated environments. This collocation of VMs and containers results in most switches to be software switches running at the hypervisor to interconnect the virtualized environments and connect them to the physical network. In such an environment, even though a network is interconnecting a large number of virtual nodes, all interactions are performed in software by the Operating System and the hypervisor. By leveraging SDN, DC operators have been able to uniformly manage the physical or virtual network infrastructure.

2.3.2 Network Topologies

large-scale network operators have traditionally mostly been Internet Service Providers (ISP), but with the rise of the "*-as-a-Service" (*aaS) model, data centres (DCs) have grown to scale larger than ISPs. DCs have become the largest and fastest evolving network infrastructures, exchanging internally a volume of traffic orders of magnitude higher that ISPs. DCs have provided the underlying infrastructure for Cloud computing in which enterprises outsource the infrastructure based on a pay-as-you-use service model. This model relieves enterprises from significant capital expenditure (CAPEX) for purchasing and maintaining hardware and software assets, instead, they increase the operating expenses (OPEX) to pay for the infrastructure as required for the current demand [58]. This model has been widely beneficial to operators and users alike, by allowing users to allocate the resources needed for operation on-demand, and delegating the maintenance and support to a third-party while allowing large-scale network operators to lease extra resources to amortise the cost of operation.

DCs have been originally designed around the existing Internet technologies applying knowledge gathered by ISPs to provide large-scale high-performance infrastructures in which low cost of operation and maintenance is paramount. Following the same design process, DCs have inherited many of the existing flaws and limitations of ISP networks, relying on the ossified Internet stack designed for Wide Area Networks (WAN), relying on best-effort com-

munication, and being highly dependent on the demand of the network. DCs' unique environment with several thousand collocated devices, higher bandwidth and lower latency gave rise to a wide new range of issues. To extend these complex requirements, DC operators are required to provide very high reliability expressed as a Service Level Agreement (SLA) to the customers with yearly downtime tallying up to less than a few hours while keeping the cost of the infrastructure and maintenance at a minimum. In Table 2.1, the number of servers from the main DC operators is reported, highlighting the current scale providers are operating at [59].

Table 2.1: Number of physical servers owned by major operators (Source [59])

Company	Number of Servers	Date
Facebook	$> 180,000$	June 2013
Rackspace	$94,122$	March 2013
Amazon	$454,400$	March 2012
Microsoft	> 1 million	July 2013
Google	$>$ Microsoft	July 2013

DC networks encompass most of the problems currently in ISP networks as well as a wide range of new issues in design, management and orchestration of very large-scale networks. Although, DC networks are extremely complex, they operate in a very different environment than ISP networks and are not subject to the same limitations, allowing faster iterations in design and deployment. ISP networks have stringent requirements in protocol compatibility, peering agreements and policies enforcement that is a consequence of the Internet being a large distributed system with no central entity operating the network. However, in DCs a single operator controls the infrastructure and can therefore evolve more rapidly, modifying the design of the network to cope with the evolving demand over the infrastructure and deploying or experimenting with recent technologies that have the potential of reducing cost and improving efficiency. Likely the most notable infrastructure changes that have been performed over these evolution cycles have been the network topologies not only to reduce cost [60], but also to prevent resource contention and possible underutilisation impacting Return on Investment (RoI). The remainder of this section will cover the most important DC network topologies that have been deployed including their benefits, limitations and

particular considerations.

Multi-Tier topologies

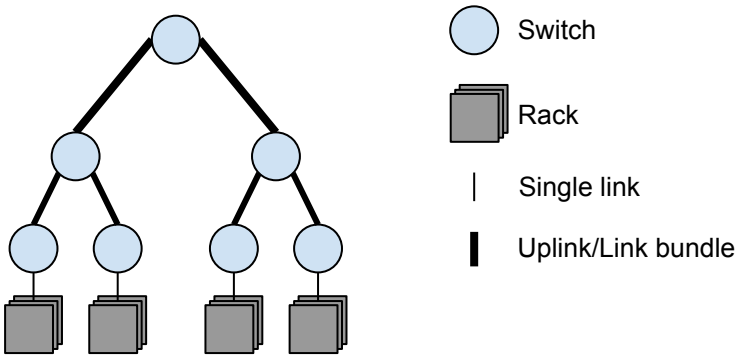


Figure 2.3: Canonical tree topology with three-tier network

The first generations of DC networks have generally been designed following a multi-tiered canonical tree topology as shown in Figure 2.3 [61]. This topology has the advantage of being very straightforward to implement, and requires a low number of devices and links in order to interconnect a large number of devices. In a three-tier topology, the bottom-most layer consists of racks of servers with each rack hosting a Top-of-Rack (ToR) switch. One layer higher in the tree, the aggregation switches connect multiple ToR switches together and provide the uplink to the core switches responsible for forwarding the traffic between the different branches. A typical deployment of this topology would likely use 1Gbps links between the hosts, ToR, and aggregation switches, and leverage the switches' uplink ports to provide 10Gbps to the core layer. This topology, although attractive due to its simplicity, suffers from a number of drawbacks:

- The links at the different layers of the topology can be highly oversubscribed, depending on the port density of the ToR switches, number of servers, and uplink capacity. A typical canonical tree topology can suffer from a 10:1 oversubscription at the aggregation layer and an oversubscription as high as 150:1 at the core [62].

- Traffic between servers in different racks must communicate through the aggregation or core layers, causing in the worst-case scenario east-west traffic across the data

centre, and resulting in significantly different traffic characteristics depending on the server's locality and application traffic patterns.

- The lack of path diversity in the infrastructure results in the loss of a large portion of servers if a single switch or link fails. In the worst case scenario, the core switch can fail preventing access to all servers.

- In order to increase the size of the network, the topology must be modified to either deploy higher density ToR switches to host more servers per rack or with higher density aggregation or core switches to increase the number of branches. In both cases, the expansion relies on replacing the existing devices with more expensive, higher-density ones, worsening the oversubscription as the number of devices increases.

While faster links are deployed in higher layers of the topology, these conventional architectures are heavily oversubscribed. The term *oversubscription* is defined as the ratio of the worst-case achievable aggregate bandwidth among the end hosts to the total bisection bandwidth of a particular communication topology. For instance, an oversubscription of 1:1 means that all hosts can communicate with arbitrary other hosts at full (local line-rate) bandwidth at any time and traffic load. An oversubscription ratio of 4:1 means that only the 25% of the host bandwidth is available for some communication patterns [63]. Traditional DC designs use oversubscription in order to reduce the cost of deployment but expose the network to congestion that results in increased latency, packet drops and reduced available throughput for the servers and virtual machines [62].

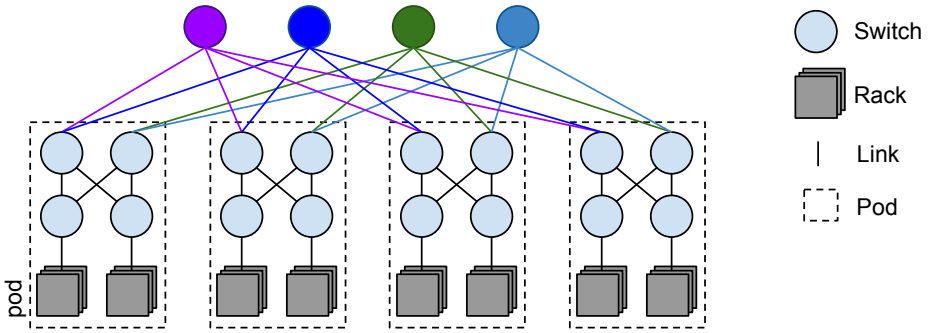


Figure 2.4: Fat-Tree topology (folded Clos) of radix 4

More recently, alternative topologies such as Clos-Tree [62] and Fat-Tree [63] have been proposed to address the oversubscription and path redundancy issues of canonical tree topologies. These architectures, as shown in Figure 2.4, promote horizontal rather than vertical expansion of the network through adding similar off-the-shelf commodity switches to the existing network instead of replacing with higher density devices. Dense interconnect in these new fabrics provides a large number of redundant paths between source and destination edge switches, resulting in better resilience in case of link or device failure and greatly reducing oversubscription. In a Fat-Tree topology, the size of the network is defined by the number of pods, with each pod connecting to the core switches. Each pod contains two layers of switches with the bottom-most layer connecting the servers to the aggregation switches. Clos/Fat-Tree architectures have seen an increasing popularity in modern data centres but scaling limitations in the number of links and switches possible often results in partial deployments:

- The limiting size for a Fat-Tree topology is the number of ports on the switches. Fat-Tree requires uniform devices to be used with k ports, resulting in k pods to be connected each containing k switches. Each ToR switch is connected to $k/2$ aggregation switches, resulting in the remaining $k/2$ ports to connect to servers, summing up $k^3/4$ hosts supported. This level of multipath network results in a very large number of links and devices to be deployed to support a limited number of servers.

- The redundant paths require the topology to be configured manually to prevent network loops. It also relies on load balancing mechanisms such as Equal Cost Multi Path (ECMP) [64] or Valiant Load Balancing (VLB) [65] to uniformly distribute the traffic between links which are unfair at balancing unequal-sized flows [66].

- Through the large number of redundant links, connectivity failures are less common. However, ToR failure can still result in the loss of connectivity to a rack and aggregation or core failure can significantly reduce the overall available bandwidth.

Until recently, two of the largest DC providers, Google and Facebook have been using the

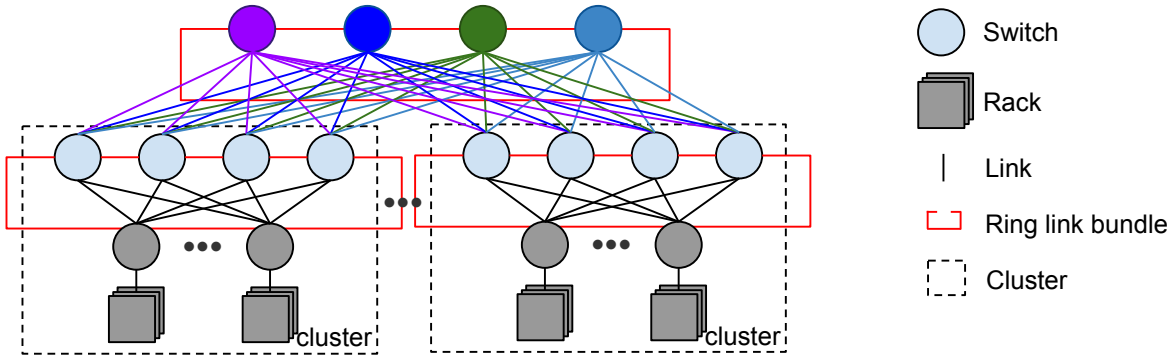


Figure 2.5: "Four posts" topology

"four-post" network architecture as shown in Figure 2.5 [1, 67]. In this architecture, the main building blocks are clusters which are much larger than the pods previously described, and each cluster involves hundreds of server cabinets each with a ToR switch. Each ToR switch is then connected to one of the 4 aggregation switches within the cluster and all the clusters are interconnected through the four core (spine) switches. To provide very high bandwidth, each ToR switch provides up to 44x10Gbps downlinks and 4x10Gbps uplink to the aggregation layer resulting in a 10:1 oversubscription. Each aggregation switch has 4x40Gbps uplinks for each core switch resulting in 4:1 oversubscription. To provide redundancy and safety from failure, the 4 aggregation switches are interconnected by a bundle of 8 links at 10Gbps (80Gbps) in a ring configuration, and the core is protected by a 16x10Gbps ring (160Gbps). This approach to topology design has been able to provide high throughput, high redundancy and resiliency, however the largest DC providers have been moving away from this design due to the following reasons:

- With the fixed design of the core and aggregation layers, the only way to scale up the number of devices within the infrastructure is to deploy higher density, higher throughput network devices that are expensive, produced only by few vendors and often very proprietary.

- The network is oversubscribed between clusters, mandating distributed applications to be confined to one cluster or allocating more uplinks at the cost of lower cluster density.

- Relying on a few but very high performance switches at the core aggregation, results in a 25% capacity reduction on intra-cluster capacity in case of aggregation switch failure while a core switch failure reduces inter-cluster capacity by 75%.

- The size of each cluster is based on the port density of the devices, resulting in only a few clusters each containing containing a large number of servers. These large building blocks make allocation of resources, management and orchestration more difficult.

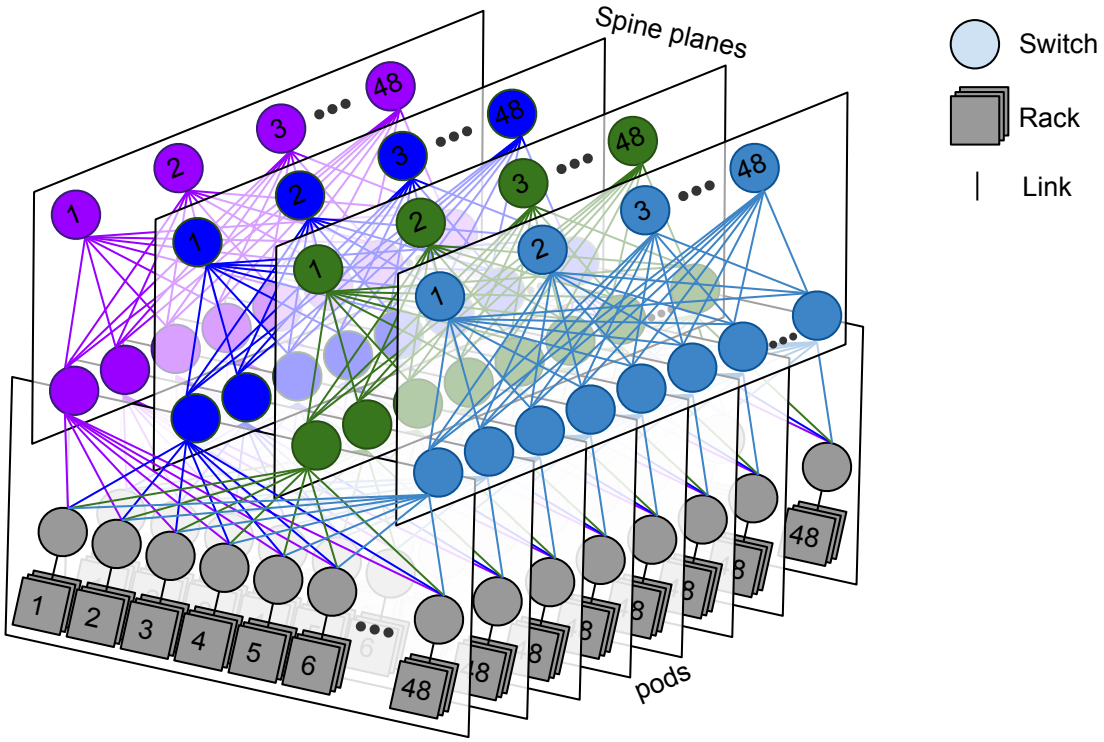


Figure 2.6: Facebook 3rd generation spine-leaf topology

The latest generation of DCs have focused on the design of very high-performance network instead of a hierarchically oversubscribed system of clusters as show in Figure 2.6 [2, 67]. In this model, the continuous evolution of the network and server infrastructure is paramount, allowing new servers and network links to be added without impacting the already deployed infrastructure. The main building blocks for this design are small-sized pods limited to 48 racks of servers to simplify the allocation of resources. To provide high bandwidth without oversubscription, each server is connected to the ToR switch with a 10Gbps link, and 4 uplinks at 40Gbps each are connected to the aggregation switches, providing a total of

160Gbps capacity for each rack of servers. To interconnect all the pods, independent planes of spine switches operating at 40Gbps are used. This design allows the deployment of new resources to be modular: If more compute capacity is required new pods can be added and if more inter-pod connectivity is required new spine planes are deployed. However, the path diversity results in some limitations similar to the FatTree topologies:

- To distribute the traffic across the different paths, ECMP flow-based hashing is used which results in an unequal distribution of traffic across the different links when the size of the flows is different.
- The high number of switches and links to provide a non-oversubscribed topology makes the deployment complex especially in existing infrastructures. To mitigate this issue, careful building design and placement of the different planes can be done to reduce the length and number of links [2].

Server-centric topologies

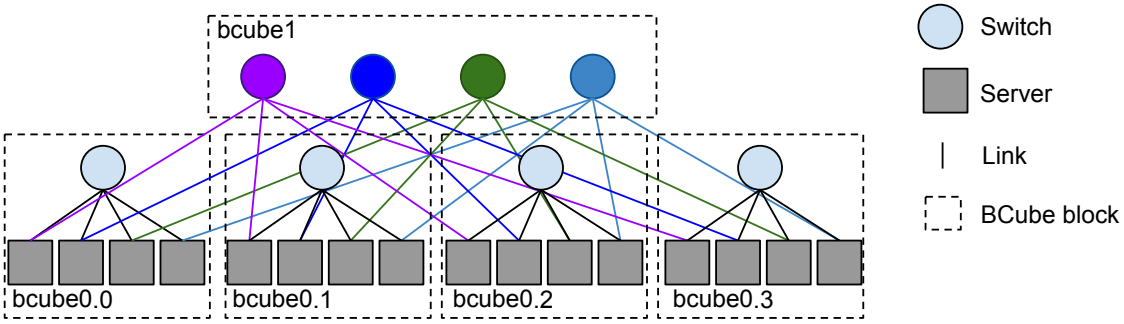


Figure 2.7: BCube topology

To address some of the issues directly related to tree topologies, new research has been looking at clean-slate designs diverging from the standard multi-tier architectures. BCube [68] and DCell [69] have been proposed as server-centric topologies, in which the servers also participate in forwarding packets. The goal of server-centric topologies is to provide high reliability of the network infrastructure at a low equipment cost. The design approach of both these topologies is similar and relies on a simple building block repeated recursively

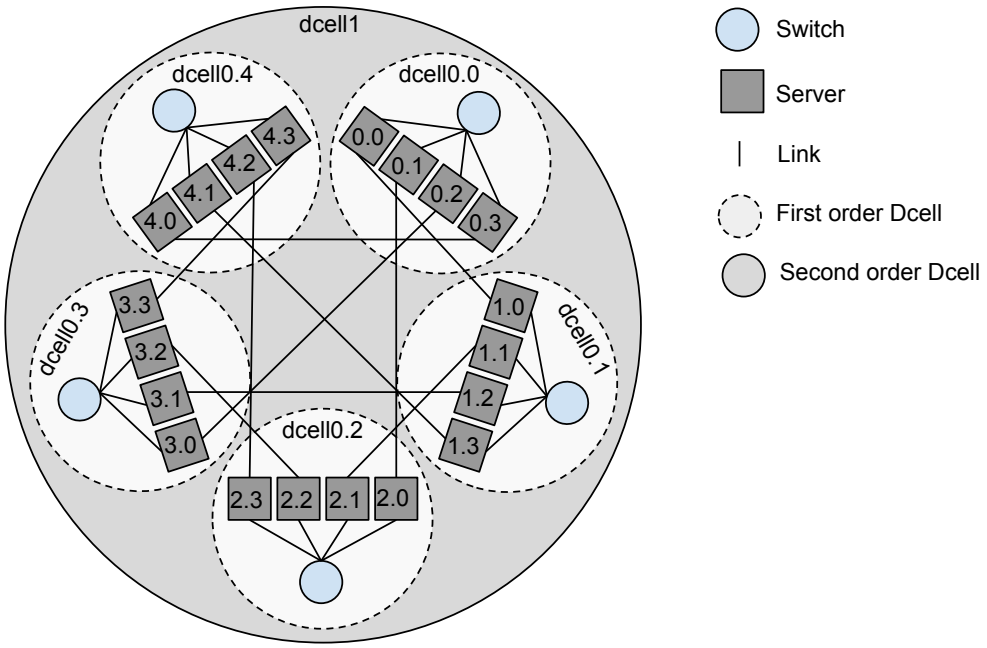


Figure 2.8: DCell topology

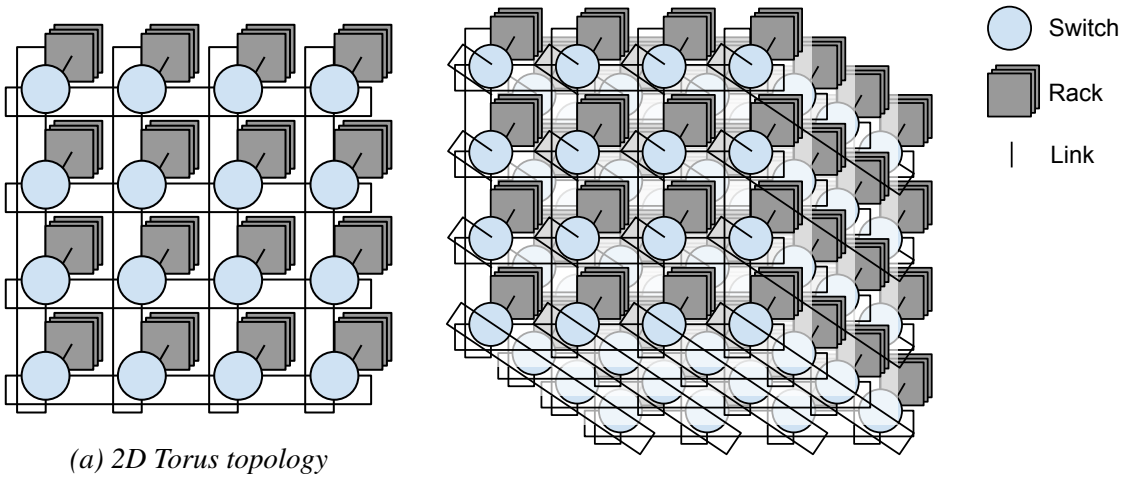
to create large network infrastructures [70]. In BCube, as shown in Figure 2.7, a block contains n servers connected to an n-port switch. A BCube_1 consists of n BCube_0's and n n-port switches [68]. In this approach, each BCube local low port density switch can provide high bandwidth connectivity amongst the servers in the same BCube and each server has an uplink to the higher level switches. The DCell design, shown in Figure 2.8, is very similar to BCube, a server is connected to a number of servers in other cells and a switch in its own cell, and large-scale infrastructures can be obtained by recursively creating new higher-order DCells. The main difference between the two is that in DCell, the inter-connection between cells is performed only through the servers and not through another set of switches. Server-centric architectures have shown that highly reliable networks can be designed without the multitude of high density switches and links in multi-tier topologies. However, they suffer a number of drawbacks preventing their adoption in mainstream infrastructures:

- Server-centric architectures are designed around low density switches but delegate a portion of the packet forwarding logic to the servers. This design choice, requires new routing mechanisms to be used to leverage the topological properties of the architecture preventing their easy deployment and compatibility with existing networks.

- The recursive design of the network makes the topology complex and hard to maintain at large-scale, requiring dedicated algorithms to generate a specific topology for a network of a certain size. This complexity in the topology makes the design, maintainability and programmability of the network harder as the network operators cannot rely on the inherent symmetry of the design like in multi-tier topologies.

- Relying on servers to provide packet forwarding has been viewed as an unreliable approach to packet forwarding at large-scale. Switches have been designed to provide network connectivity over very long periods of time without maintenance. However, servers have not been designed as a critical aspect of the network and therefore resulting in a loss of connectivity during maintenance, reboot and degraded performance when highly-utilised.

Mesh topologies



(a) 2D Torus topology

(b) 3D Torus topology

Figure 2.9: 2D and 3D Torus Topologies

Some specific environments such as High Performance Computing (HPC) have led to the development of dedicated topologies that are able to provide very high bandwidth with very low latencies between devices. The three most common topologies are mesh, torus and hypercubes that belong to the k-ary n-mesh family. A 2D torus and a 3D torus topologies are shown in Figures 2.9a and 2.9b respecitively. In a 2D k-ary n-mesh, each switch only requires

at-least 5 ports; four ports connect to the neighbouring nodes at the top, left, right and bottom, and the last port(s) connect to the servers for this switch. The difference between torus and mesh topologies is that, in a torus, the edge switches are interconnected both vertically and horizontally creating network loops at the edges, while mesh topologies are loop-free leaving a port disconnected on the edge devices. Hypercubes follow the same structure as toruses but create a 4 dimensional topology. These topologies have seen very little deployment in general data centres as the cost of operation and deployment can be very high and many of the design parameters are incompatible with today's data centre traffic:

- In k-ary, n-mesh topologies, the latency is directly related to the number of hops traversed which increases directly with the distance between nodes. This high hop count makes the performance of such topology directly related to the locality properties of the traffic pattern, resulting in high performance when communication is limited to neighbours but slower and oversubscribed when communicating with nodes further away.

- In DCs, the most common communication pattern is "many-to-one" where multiple servers aggregate data at a single node, for instance, caching clusters or databases. Because of such traffic patterns and the locality requirements of k-ary n-mesh topologies, the performance is often suboptimal for DC traffic.

- These topologies require a high number of low-density switches and a very large amount of links to create the interconnect. This large path diversity makes the deployment of such infrastructure very complex and costly over large areas to support a high number of servers.

2.3.3 Management & Orchestration

Management Network

The network infrastructure of DCs is generally separated, physically or virtually, into two distinct networks, one carrying the production traffic between servers and the outside world,

and a second private network used by the operator to manage and orchestrate the servers and the switches. The management software within the infrastructure is responsible for checking the state of the servers and maintaining the accurate view of the overall resources utilisation, such as processor, memory, storage, and network load. The management network is also used to communicate with the VM hypervisors on the servers to start, stop, or migrate the VMs between hosts at runtime. Moreover, with the rise of SDN and the centralisation of the control plane logic, these networks are becoming even more important. In order for the switches to notify the controller to make a routing decision and send this decision back to the switches, the management network is used. Through this separation, the control logic can remain private to the network operator and can be transmitted without being impacted by the production traffic.

While management networks can assume similar topologies to those described above (for the data-carrying network), these networks are typically designed for sparse and latency sensitive traffic, where maintaining high throughput is not as critical as for the data-carrying production network. The management network should therefore be designed to provide reliable and consistent performance, regardless of the load over the production traffic. Management networks have been generally one of the following three types:

1. In-band network: the simplest way is to use the same "in-band" network for the management network as the one carrying the production traffic of the tenants. In this scenario, there is no isolation between production and management traffic, therefore the management traffic is subject to congestion or low bandwidth caused by production traffic. As a result, in-band management network is not recommended for production DCs.

2. Logical out-of-band (OOB) network: in this approach, the management network is logically separated using, for instance, VLANs or dedicated flow rules in the switches. Extending the in-band solution, this approach allows QoS enforcement to prioritise management traffic over tenant's traffic (e.g., by assigning different queues in switches). However, as isolation can only happen in certain points of the network (at routers ca-

pable of QoS enforcement), logical OOB does not guarantee fully-fledged isolation of management and user traffic.

3. Physical OOB network: a physically separated network can be set up solely for management purposes. While this incurs significant investment in new switches and network interfaces for hosts, this solution is preferred for critical environments. In fact, a physically separated management network is being deployed at many production cloud DCs [71] and it is the recommended solution for the OpenStack open-source cloud software.

Network Operating System (NOS)

In conjunction with the deployment of SDN (specifically OpenFlow) and the creation of larger and more complex networks and topologies, the concept of Network Operating Systems (NOS) has been proposed to abstract the resources of a network. Alike Computer Operating Systems, NOS are designed to provide an abstraction over the low-level configuration of the individual components. In a traditional infrastructure, each component of the network such as switches, Access Control List (ACL), Firewall and much more are configured independently without any shared knowledge and abstraction between devices from different vendors. The goal of a NOS is to provide a uniform and centralised programmatic approach to the entire network components to allow simplified management and orchestration. By itself, it does not manage the network or infrastructure but provides the programmatic interface to do so, and allows applications implemented on top of the NOS to perform the actual management tasks. The programmability provided by a NOS should be broad enough to support a large set of management and orchestration applications, and over the years a large range of applications have been developped to manage resources throughout the networking stack including but not limited to the following:

- **Network Virtualization and Slicing** has been proposed to share the physical infrastructure between multiple tenants in the same infrastructure by virtualising a single physical network into multiple, lower capacity, virtual ones [72, 73].

- **Traffic Engineering** techniques have been proposed to better utilise the resources available in the infrastructure and particularly allow the network to be used more efficiently while alleviating existing problems such as network hotspots [74, 75].

- **Energy Aware Computing** is becoming more and more relevant as cost and governemental incentives are pushing operators to reduce the carbon emissions of the infrastructure while keeping the same performance and resilience provided by today's networks [76, 77].

- **Policy enforcement** has been a requirement for large networks since the very beginning but until recently, the management of such policies has been costly, risky and slow. With the deployment of a NOS these management tasks can be achieved by a single software application [10, 11, 78].

- **Virtual Machine migration** schemes have been proposed to use the globally available network information to relocate VMs at runtime to improve pairwise latency and throughput while reducing the utilisation over costly links within the infrastructure [79, 80, 81].

- **Network Function Virtualization** has been proposed more recently to replace the existing legacy middleboxes within the network to provide a cheap and reconfigurable infrastructure. Coupling NFV and SDN, the NOS application can dynamically reroute traffic to particular network functions throughout the network infrastructure [82, 83, 84].

- **Security and Resilience** aspects have also been addressed, in order to rapidly recover from failure and allow a very quick convergence of the routing tables over wide area network, to mitigate the impact on connectivity loss [85, 86].

The terms of "NOS" and "controller" have been used interchangeably by the networking community to express the centralised management and orchestration infrastructure. The term controller comes directly from the OpenFlow terminology in which a centralised application

called *controller* is responsible for taking the routing and forwarding decisions when the network devices are incapable of doing so. In a NOS two sets of APIs are exposed that can be defined as follows.

- **Southbound API:** A well-defined protocol designed for the communication between the centralised controller and the network devices (switches or routers). The southbound API defines the way the SDN controller should interact with the data plane to add or alter routing and forwarding entries, retrieve traffic characteristics and configure the network interfaces. The OpenFlow API is the most common southbound API and provide the abstraction necessary to add and alter the data plane flow tables. Other southbound APIs are now becoming well supported such as Netconf and OVSdb.

- **Northbound API:** An API exposed by applications running on the NOS for services and applications running over the network. The northbound API is used to expose the network state and management functions to applications and services. Services can query the current network state to dynamically adapt to the demand and can automatically provision the network to align with changing network requirements. A variety of different protocols have been deployed, but HTTP REST is commonly used due to its simplicity and wide support.

The controller communication is limited to the network devices to establish the routing and forwarding decisions through a southbound API such as OpenFlow. A NOS encompasses the function of a controller but also provides a wider abstraction on the networking elements, possibly supporting multiple southbound APIs as well as exposing the network state to third party applications through a northbound API. One of the main differences is that southbound APIs have been well defined through comprehensive and lengthy specifications to allow vendors to provide support in their switch implementations. On the contrary, the northbound interface is not defined and any NOS application provider can implement the northbound API in any way they choose. Over the last decade, software developers have mostly exposed the northbound API through REST HTTP interfaces as they are easy to implement and can

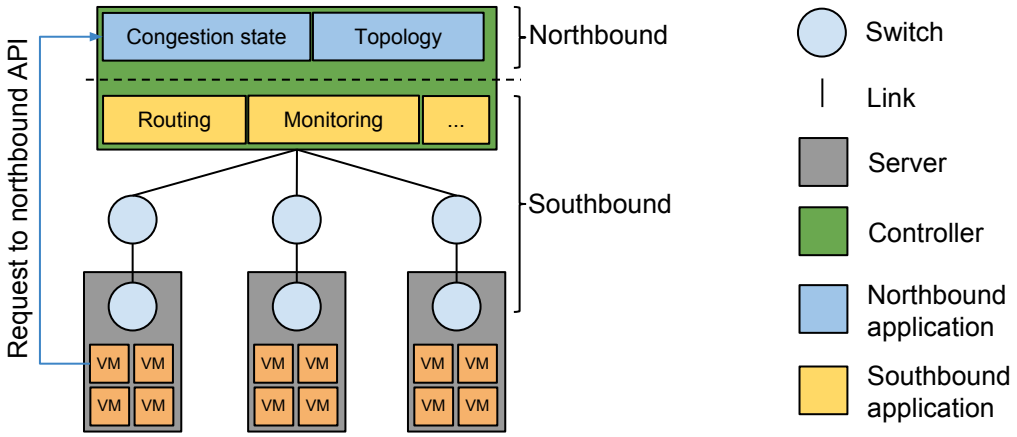


Figure 2.10: NOS Northbound and Southbound API separation

be easily supported in a large range of programming languages. Figure 2.10 shows the separation between the southbound interface managing the physical and virtual switches with the northbound interface exposing the network state to applications. Table 2.2 lists today's major SDN controllers/NOS with the supported southbound and northbound APIs.

Table 2.2: Major SDN Controllers/NOS

Software	Driving Organisation(s)	Language	Southbound API	Northbound API
NOX	Nicira Networks	C++	OpenFlow 1.0	Limited C++ API
POX	Nicira Networks	Python	OpenFlow 1.0	Python API
Ryu	OSRG & NTT	Python	OpenFlow 1.5, Netconf, OF-config	REST, WebSocket, Python API
OpenDaylight	Brocade, Cisco, Ericson ...	Java	OpenFlow 1.3, OVSdb, Netconf	REST, Java API
ONOS	ON.Lab	Java	OpenFlow 1.3, OVSsb, Netconf	REST, Java API
Floodlight	Big Switch Networks	Java	OpenFlow 1.3	REST, Java, Python, C API

2.4 Current Issues and Limitations

Cloud DCs have been designed following the inherited design patterns from ISP networks that have led to design decisions that underutilise the infrastructure and the resources available. In such environments, the resources are statically provisioned based on long procurement processes, leading to a fix set of resources that must support the changing demand. This static resource admission scheme has led to over provisioning in order to cope with peak demand, resulting in low resource utilisation during normal operation and therefore increased OPEX. To mitigate this, modern virtualisation technologies and containers have simplified multi-tenancy and the ability to transparently relocate VMs to consolidate the utilisation of

CPU and memory resources. However, this approach only considers a few of the available resources in a DC without considering the possible adverse impacts it might have on other aspects, such as network latency and throughput, policy enforcement, energy utilisation and much more.

This problem is further amplified by the different workload and traffic patterns that are observed in cloud DC infrastructures. At the scale of an ISP network, the aggregate demand over long periods of time remains constant, with peak periods of traffic that can be observed based on collected data over long term operation. In a DC environment, the resources utilisation and demand fluctuates much more rapidly based on the demand from the users and the services deployed. With each user being able to dynamically provision resources over short timescales, accurately predicting the demand for resources is unlikely. This is amplified by phenomena like flash-crowds that increase the peak demand extremely quickly for short periods of time, and the quick iterations in software design that constantly change the traffic patterns.

However, in DC environments, the infrastructre is owned and managed by a single entity and is not constrained by the multi-domain environment of the Internet. This single ownership of compute and network resources gives new opportunities to operators to better utilise the resources and adapt the environment to the infrastructure parameters. Such opportunities allow the operator to provision the infrastructure in an adaptive, load-sensitive manner, using information available at the different layers to tune its operation. The resulting optimisations allow the "capacity headroom" to be increased, allowing the utilisation of resources to be higher and consequently improving RoI.

In order to collect the information at the different layers, metrics must also be collected and updated continuously to reflect the changes in software, hardware, utilisation and communication patterns. However, due to the very large-scale of DC infrastructure with tens of thousand of devices, the frequency at which this information can be collected is limited with the traditional approaches. The Simple Network Management Protocol (SNMP) is likely the most widely-deployed telemetry system currently used but is becoming unsuitable for

large-scale infrastructures that need to cope with the high demand, frequency of updates and reported metrics. SNMP and the majority of the management and monitoring protocols have been designed around a "pull-model" in which the metrics are collected locally at the device and must be queried by a third party application continuously to keep a network-wide view of the metrics. This approach has been suitable for small-scale low frequency updates, but is unable to provide the granularity and accuracy required to control and manage very large-scale infrastructures in short timescales [87].

Additionally to the limited granularity and frequency of measurements, current management and orchestration systems are limited to only a few metrics due to the complexity and cost of large-scale fine-grained collection. Today's measurements are limited to per-host CPU, memory, and storage utilisation, while the network metrics are throughput and latency observed over a short sampling period by the end-hosts (10 seconds in the default configuration of Nagios, a popular monitoring software [88]). In order to better utilise the available resources and dynamically adapt the infrastructure to the changing demand, it is necessary for more metrics to be collected both at the end-hosts and from within the infrastructure to reflect both the observed behaviour and internal operation. An example of such divergence between the end-host observed behaviour and the internal operation is the misbehaviour of TCP congestion control: from the end-hosts, the network can look overloaded while measurement from the devices can highlight that the network is underutilised but the traffic pattern causes bursts of traffic that are perceived as a heavily utilised network by the end-hosts [89, 90].

To exarcerbate these issues, the different control loops and metrics in such large infrastructure operate at very different time scales. Individual packet dynamics change within a few microseconds, the admission of the flows in milliseconds, and the aggregate volume changes in larger time windows of multiple minutes. The VMs lifecycle and locations change over hours, days or months, and the physical infrastructure remains the same over few years. This large variety of metrics and associated time scale makes the collection process more complex as per-metric collection and analysis period must be defined. As a consequence, the correlation between the metrics is very limited and most of the information collected is used to

modify and adapt the layer that provided the metrics, without providing additional contextual information to the lower or higher layers that could be beneficial.

Table 2.3: DC Control Loops Timescales

Control Loop	Timescale	Description
Congestion Control	microseconds	Dynamic of packets within the network that is impacted by the instantaneous demand on the infrastructure.
Flow Admission	milliseconds	Changing number of flows within the network based on the user and application demand.
Aggregate volume load	minutes – months	Cummulative view of the current utilisation of the network, showing peak demand during particular hours of the day (lunch, evenings), day of the weeks (week-days, weekends), months (peak demand before holiday seasons).
Policy Enforcement	hourly	Changes in access control lists, firewalls or network modules that are dependent on the currently authorised users, services and applications.
Software Iteration	daily – weekly	Continuous iterations of deployed software especially for user oriented applications that are updated regularly.
VM lifecycle	minutes – years	Dynamic changes in the state of the virtual machines, especially in cloud environments in which tenants have direct access to start, stop or deploy new VMs.
Link/Device Failure	milliseconds – hours	Time for equipment failure, transient error can last tens of milliseconds while misconfiguration or critical failures can last up to few hours.
Topology Changes	months – years	The node and links in the DC infrastructure evolve as the demand changes and new hardware becomes available.

This lack of information exchange results in the resources within the infrastructure to be underutilised. In an ideal environment, the information gathered at all layers should be globally available to the different control loops and at different granularities to cope with the individual operational time scales. To show the different operating time scale and the different operating layers, a summary of some of the most important control loops is listed in

Table 2.3. However, this information exchange is complex as the requirements for each layer are different, and collecting this large volume of data continuously at a fine granularity with no direct motivation on which layer is going to be improved is unattractive to the operator. The cost of collecting information at such large-scale is multi-fold, the storage cost is high in order to cope with the volume and long term historical data, the impact on the management infrastructure to continuously report metrics from all the nodes in the network and the complexity to identify relevant anomalies or deviation from normal in the large aggregate of data.

A further complication of this approach is the requirement for safety and confidentially of the collected data, it might be useful for a tenant of the infrastructure to be able to know the current utilisation and contention of some of the resources on the infrastructure to better optimise its own operation but also exposes internal and potentially sensitive information.

The optimisations and tuning should be envisaged throughout the operation stack of the infrastructure, from the bare-metal server and the interconnecting switches and network links, to the top-most application and service layers. To fully support this, the design of the infrastructure and inter layer communication should be approached in a converged manner, both from an application and service perspective, while considering the realistic requirements and limitations of the hardware and the infrastructure. Today's designs fail to consider such approach by either providing a high-level design that cannot be deployed easily or efficiently at large-scale, or high-performance low-level designs that fails to provide the abstractions necessary for large-scale adoption. These design issues are directly related to the gap between the user's requirements, the operator's cost, utilisation and privacy objectives, and the hardware vendor's incentives.

In an environment designed for full stack tuning and optimisation, a set of APIs should be available to the application developers. Through this APIs and the provided abstractions, application developers should be able to collect the metrics and historical data necessary to optimise the control loop and operation. The very large-scale DC customers like Netflix or Airbnb could fully leverage the available programmability to better utilise the very

large number of machines they have allocated and optimise the delivery of content to the end-users [91]. More traditional users operating at smaller scale with few servers, might rely on some libraries or services provided by the DC operators leveraging such API without the need to have a deep understanding of the problem space and complete access to sensitive information (physical DC topology, other users, etc.). Examples are some features already implemented by DC operators, such as the autoscaling provided by the Google Cloud Platform that relies on end-hosts CPU and network utilisation as well as HTTP metrics to dynamically provision and load-balance virtual machines. In either case, for the customers and the operators to adapt to the operating environment a novel set of low level APIs should be available.

Using the data available through these APIs, the applications can be designed considering the real operating environment without the need to rely on conservative values in order to cope with the unknown operating condition. Once these requirements defined by the application, the infrastructure should be dynamically programmed to collect and report the metrics of interest to the application. This proposed approach relies directly on the infrastructure to be programmable and dynamically reconfigured depending on the current operating status and the applications deployed. The ability to deploy on-demand monitoring modules can be easily imagined for server side resources at either the physical layer on the hypervisor or virtualisation layer. However, monitoring hardware resources is much more complicated. This complexity comes from the trade-off of most hardware architecture in which flexibility and programmability often compromises the maximum achievable performances.

2.5 Summary

large-scale infrastructure operators such as ISPs and DC operators have been heavily competitive while the constantly growing user demand has pushed the operators to deploy extremely large-scale infrastructures. The main complexity with this approach has been to maintain the end-to-end compatibility between operators and services. Today's protocols are a legacy of the network and control loops designed in the early days of the Internet that perform

suboptimally in modern environments. The providers have therefore been challenged with maintaining the compatibility requirements of these legacy systems while trying to improve the design and management of the modern infrastructures.

In order to provide a better orchestration, management and programmability of the infrastructure, multiple programmability models have been proposed over the years allowing operators and end-users to directly interface with the infrastructure. The concept of Software Defined Networking, has been the first approach to be widely accepted and deployed by both industry and the research community due to its pragmatic design allowing limited programmability on existing devices. This newly available programmability has showed the usefulness and benefits that some level of programmability can provide in a wide range of domains, from the hardware layer up to the application layer. However, OpenFlow, due to its very pragmatic design has limited flexibility, and therefore the SDN paradigm must evolve beyond its current state to provide the next generation of management and orchestration.

In this thesis, the research will address the performance and utilisation benefits, that cross layer information can bring when applied to the transport layer. Secondly, this work presents a novel framework allowing the measurement, collection, and reporting of network metrics necessary for large-scale infrastructure's management and orchestration.

Chapter 3

Logically Centralised Network Resource Management

3.1 Outline

Numerous warehouse-scale data centres have been deployed in the last decade to support the emerging needs of Internet applications and services, ranging from web search to storage and general large-scale computation clusters such as Hadoop [92]. In order to support existing applications, data centres have been built using legacy Internet technologies. As such, the time-tested TCP protocol is used as transport for most applications. However, TCP was originally designed to operate over the wide area where the network characteristics are unknown and the latency, throughput, packet loss and flow characteristics are widely different from those in a data centre environment [93]. This mismatch between the TCP congestion control parameters and the operating environment of a data centre are responsible for a phenomenon called TCP throughput incast collapse, resulting in underutilising the network and delivering extremely poor performance for applications with a partition-aggregate traffic pattern [94, 60].

Since data centres are typically managed by a single authoritative entity and centralised management protocols such as OpenFlow are becoming increasingly popular, the unknowns

upon which TCP congestion control parameters are set should be revisited. In contrast to the wide-area Internet, the topology, throughput, latency, and device properties are either known or can be discovered, which can subsequently be used instead of the conservative and inadequate default congestion control values. Therefore, in such an environment, it comes to question whether each host should still estimate its own congestion control parameters based on locally-observable, and hence limited network behaviour, or environment-specific parameters should be computed globally using the complete view of the network and distributed to the hosts. As a consequence of managing the congestion control parameters centrally, typical misbehaviour created by the static pair-wise congestion control configuration, including long latencies, low throughput and unfairness can be alleviated.

In this chapter, the main resource of interest for centralised management is network utilisation as networks are critical of DC infrastructures and improving network connectivity directly benefits application and services. The following sections describe the misbehaviour of TCP in a DC environment as well as the conditions for this misbehaviour to occur. Subsequently, Omniscient TCP (OTCP) is presented as an approach to improve the overall network performance and utilisation of modern infrastructures through the information provided centrally in an SDN environment. OTCP is able to achieve this by centrally collecting the network infrastructure properties and subsequently computing and distributing congestion control parameters suitable for the operating environment. Through OpenFlow [13], the leading realisation of SDN, OTCP can be deployed side-by-side with a controller to collect operating network metrics including topology, latency, throughput and switches' buffer sizes. Once those metrics are collected, OTCP calculates suitable TCP retransmission timers, RTOmin and RTOinit, as well as the initial and maximum congestion window size that matches the Bandwidth Delay Product (BDP) between end-hosts.

3.2 TCP in Data Centres

3.2.1 DC Network Characteristics

TCP has been designed to operate efficiently over WAN networks. However, in DCs the network characteristics are different. The reported values from large-scale providers highlight that TCP accounts for 99.9% of DC traffic [95]. Nearly 75% of the traffic stays within a rack, and most of it is kept within the data centre [93]. The network can be highly oversubscribed between the different layers of the fabric depending on the network topology and the number of servers. At the rack level, each machine has an associated 1 or 10 Gigabit Ethernet (GbE) link to the top of rack (ToR) switch. The aggregation layer (Agg) responsible for interconnecting ToRs together has typically a 5 — 20 oversubscription ratio, while the core of the network is typically oversubscribed by a factor of 80 — 240 [62]. With latencies varying from hundreds of microseconds within the same rack to few milliseconds inside the same DC, a single-value-fits-all for TCP congestion control in DCs is not suitable without being overly aggressive or conservative. The traffic characteristics are different to the Internet, more than 50% of the time a machine has 10 concurrent flows and at least 5% of the time it has over 80 [62]. 99% of the flows are mice flows, small in size and delay-sensitive. The remaining flows are large, representing 90% of the overall bytes transferred and are throughput instead of delay sensitive called elephant flows [95, 62, 93].

- **Mice flows:** Short TCP flows, with a small number of packets that are associated with bursty, latency-sensitive applications. They are the most numerous flows within data centres but represent a small fraction of the total bytes carried. The size of a mouse flow is not well defined, varying in the literature between 10kB to 1MB payloads [96, 97].

- **Elephant flows:** A small number of long-lasting TCP flows carrying the majority of the data. Elephant flows are sensitive to throughput, to quickly tranfer large amount of data, but are not latency sensitive.

DCs are managed by a single authoritative entity, therefore infrastructure-wide characteristics are available. They can be used to better tune the TCP stack for a specific environment and manage the allocation of network resources. Contrary to the Internet, the topology is known or can be discovered, there are a limited number of hardware configurations available, and they have known properties such as buffer occupancy, switching speed and induced delay, dropping mechanisms, and supported active queue management (AQM). In addition, the bandwidth and latency between nodes in the network is known or discoverable, allowing a maximum bound on the congestion window to be set based on the actual BDP of the path.

Over the last decade, SDN has become increasingly prominent in DC infrastructures since the OpenFlow protocol defined a framework for network management and configuration in a centralised manner. Through SDN, a centralised controller is able to maintain, manage and control an up-to-date view of the networking infrastructure. Using available information, such as the network topology and latency, routing mechanisms, queue length and throughput, globally-informed decisions can be made that would not have been possible in a legacy network where the control plane is fully distributed amongst the forwarding elements [13].

3.2.2 TCP Incast Collapse

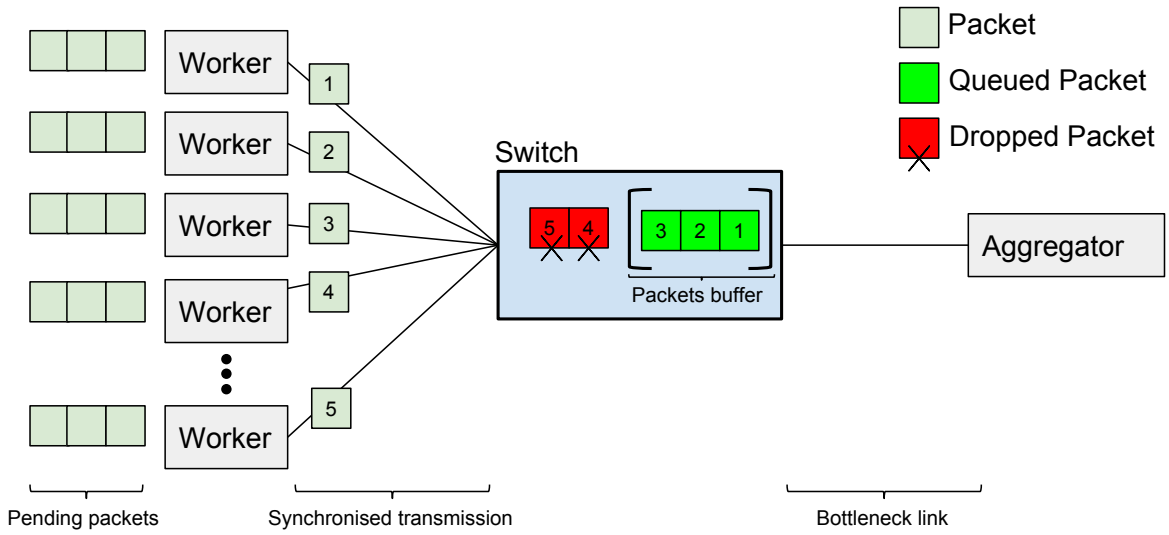


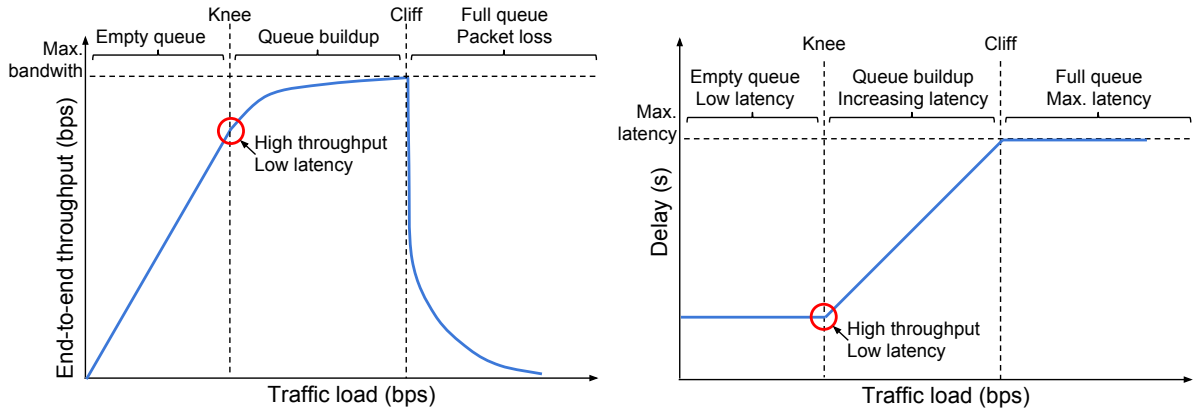
Figure 3.1: Simplified representation of partition-aggregate traffic resulting in TCP incast throughput collapse

A known problem in many DCs relying on the legacy TCP protocol is TCP incast throughput collapse. This is caused by the typical partition-aggregate workload where a single host issues a single query to a large number of workers and waits for the response [98, 99]. The response of the multiple servers is highly synchronised, generating a large burst of traffic from the workers to the aggregator, overwhelming the buffer of the bottleneck switches and causing significant packet loss. Figure 3.1 shows a simplified diagram of TCP incast throughput collapse in an environment with 5 workers simultaneously replying to the aggregator, resulting in the switch dropping some of the packets. To cope with the loss of packets, TCP relies on mechanisms such as TCP fast retransmit (F-RTO) to resend lost packets when duplicate (usually three) acknowledgement (ACK) numbers are received [100]. If the number of lost packets is high enough to prevent the receiving host to generate the duplicate ACKs required for F-RTO, TCP relies on the retransmission timeout (RTO) to resend the packet. During communication the RTO is configured by TCP to match the round-trip delay between sending a packet and receiving its acknowledgement, commonly known as Round Trip Time (RTT) with a minimum value (RTOmin) of 1 second by RFC2988 and no less than 200ms by commodity operating systems [101]. This 2–3 orders magnitude difference between the network RTT and the timeout results in bursty retransmissions phases underutilising the network, and resulting in low throughput and long Flow Completion Time (FCT) for delay sensitive flows [95].

Previous literature on TCP incast collapse has attributed the misbehaviour of TCP and the extreme burstiness of the transmission under partition-aggregate workloads, to the mismatch of the retransmission timeout and network characteristics [94, 102, 99]. The long retransmission timeouts create long idle periods between short transmission periods with high throughput and high buffer occupancy. This work contributes to this discussion of mismatch between TCP congestion control parameters and the network characteristics. It highlights that the retransmission timers are important but the currently neglected congestion window control parameters should also be considered while solving TCP incast. At the onset of congestion collapse, RTO is used as a recovery mechanism. However, by also configuring the congestion window to match the BDP of the network, the rapid buffer build-up in the bottleneck

switches can be mitigated, consequently preventing numerous packets dropped and resulting in throughput collapse.

The initial window (IW), increased to 10 Maximum Segment Size (MSS) in RFC6928 from 2–4 MSS in RFC3390 is not suitable for high-throughput, low-latency networks [103, 104]. With an IW of 10 segments, a single flow can send up to 15kB of unacknowledged data after the three-way handshake. In partition-aggregate workloads, hundreds of short-lived flows can be initiated simultaneously sharing the same bottleneck path, resulting in a burst of few megabytes that will quickly overflow the bottleneck switch. The value of 10 MSS might be suitable for WAN environments with long fat pipes carrying few long-lived large flows, but it is unsuitable for a DC environment with numerous concurrent mice flows competing for high throughput over extremely low latency links. The initial window should be configured to match the BDP of the network as it represents the amount of unacknowledged data on the path between two endpoints. However, because of the large buffers on the switches along the path, the delay can greatly vary from an idle network to a network with all buffers full.



(a) Impact of traffic load on end-to-end throughput as queue occupancy increases.

(b) Impact of traffic load on latency as delay increases with queue occupancy.

Figure 3.2: Impact of traffic load on end-to-end throughput and latency.

Figure 3.2a schematically demonstrates the relationship between the traffic load and the end-to-end throughput. When the traffic load is low, the packets can be forwarded as fast as they are received. Once the traffic load reaches a certain point, the traversed switch must start queuing packets. This point is referred to as the “knee”, where the load reaches the network capacity. After the “knee”, the throughput increases very slowly, but the queue

starts to build up. As the traffic load increases further, it reaches the "cliff" at which point the queue is fully occupied and incoming packets are dropped. As the packets are retransmitted to recover from losses, the total load increases, worsening the congesion and resulting in congestion collapse with very low end-to-end throughput. As the traffic load reaches the "knee" and the queue starts to build up, the end-to-end latency increases as each packet is delayed until every packet received beforehand is transmitted. This relationship is shown in Figure 3.2b. Up to the "knee", the traffic load does not impact the latency and the delay is a consequence of the link length and the packet processing time necessary at the end-hosts and traversed switches. After the "knee", the latency increases steadily as the queue occupancy grows until the "cliff" at which point the maximum delay is reached. In order to achieve the maximum throughput and the minimum latency TCP must operate at the "knee", in which the end-to-end throughput is high, but the traffic load low enough to keep low delays and low end-to-end latency [105].

Therefore, high throughput and low latency are achievable in DC network with partition-aggregate workloads but require fine tuning of multiple parameters based on the infrastructure operating characteristics. The size of switch buffers is not strictly a congestion control parameter, but significantly affects the algorithm's behaviour by hiding or delaying the congestion events. Using shallow-buffered switches, the latency can be kept low, but TCP's large, default IW quickly overflows the buffers, resulting in low throughput. This fine tuning of the parameters requires the IW to be reduced in order to limit the initial burst of packets and resulting queue overflows in traversed switches. RTOmin must also be decreased to prevent long "off" periods between transmissions. By following these two steps to mitigate and recover from TCP incast throughput collapse, the overall FCT of synchronised flows can be greatly improved.

3.2.3 Research efforts

Significant work has been done on TCP to optimise its performance in specific environments. Variants such as the Rate Control Protocol (RCP) and the Variable-Structure conges-

tion Control Protocol (VCP) estimate link congestion to avoid queue build-up but require custom support on the switch and end-hosts [106]. Fast TCP and XCP have been proposed for environments opposite to DC using long-distance, high-latency links [107, 108]. These protocols have been optimised to achieve high throughput for long-lived flows over long fat pipes (LFP), opposite to what OTCP aims to achieve. TCP NewReno and TCP SACK modify the congestion mechanisms to slightly increase the throughput, but do not prevent TCP Incast collapse. TCP Vegas aims to reduce buffer occupancy by introducing a delay-based algorithm. However the low latency and high variability of DC networks is too fine-grained for delay-based congestion control. TCP pacing has been proposed to tackle low throughput, but it does not prevent high latencies due to queue build-up when the number of concurrent flows is high, and it is not effective in low-latency networks [109, 110]. TCP pacing relies on the estimated RTT to "pace" the sending rate of the packets, but in incast collapse scenario, packets of the initial window of a flow can be dropped preventing the RTT to be measured and preventing the pacing to happen before congestion occurs.

In order to recover from TCP incast collapse and prevent the ripple effect of synchronous retransmission after backoff, adding jitter to the backoff timeout has shown an improvement on the throughput, but a degradation on the mean FCT. Facebook had the more radical approach to drop TCP and implement a congestion algorithm on top of UDP with a transmission window size based on the number of concurrent flows in the system, supposedly halving the overall request time of their memcache cluster [94]. Zhang et al. propose to retransmit the last packet of the congestion window multiple times to forcefully generate ACK for F-RTO [99]. F-RTO requires duplicate ACKs to be received, but during congestion events most of the packets in the window can be lost. Therefore, by sending multiple copies of the last frame in the window the chances to receive duplicate ACKs is higher and hence recover quicker. However, this approach adds unnecesary traffic to the communication and increases the size of the congestion window which is already too big for DC environments. Another significant contribution to this problem has been the evaluation of fine-grained retransmission timeouts in a data centre environment [94]. Allowing the minimum retransmission timeout to match the round trip time of the environment instead of using an overly conservative value

as well as providing high resolution timers for the TCP stack shows that the throughput can be significantly improved, and long idle times between retransmissions can be prevented.

DCTCP has been a significant contribution to mitigating TCP Incast Collapse in data centres [95]. DCTCP requires Explicit Congestion Notification (ECN) support from the network and instead of treating each ECN-marked packet as a congestion event, it uses the fraction of marked packets to pace the sending rate. In doing so, the presence and extent of congestion can be estimated. The main concept of this approach is to keep the buffer occupancy of each switch as low as possible to prevent new packets from being delayed. DCTCP requires support from the kernel of both the sending and receiving hosts and, similarly to ECN, it reacts to Incast collapse but does not prevent it. In addition, it has known problems to keep the queue at a stable occupancy and some work such as double-threshold DCTCP (DT-DCTCP) has been proposed to solve this issue at the cost of a more complex configuration [111]. PCC and Remy use machine learning techniques to discover suitable congestion control parameters for the network [112, 113]. PCC relies on online micro-experiments performed by the end-hosts to measure the throughput and loss rate to empirically estimate the best operating parameters. Remy generates congestion control algorithms based on a priori knowledge or assumptions about the network and desired objectives.

Fastpass has been the first proposal to depart significantly from the traditional congestion control algorithms and use global state information to achieve zero-queuing transmissions. Fastpass proposes to disable congestion control in the end-host altogether and delegate individual packet scheduling decisions to a centralised controller [114] with a timeslot allocated by a controller for every packet. This approach allows buffers to be kept at low utilisation and limit TCP retransmissions, at the cost of increasing the mean time of each packet by the RTT between the host and controller. Fastpass requires the controller to centrally manage a very large amount of flows and packets and it comes to question whether such approach is tractable with a large number of hosts in the infrastructure.

3.3 TCP Congestion Control Parameters Analysis

In this section, the TCP incast throughput collapse impact on traffic is experimentally evaluated through measurements. In order to accurately estimate the impact of the different parameters, and avoid the possible inaccuracies or artefacts of simulated environments, the following experiments have been conducted over a hardware-accelerated NetFPGA 1GbE switch with event capture and rate limiting modules. The NetFPGA is a highly specialised Network Interface Card (NIC) that contains a high performance Field Programmable Gate Array (FPGA), allowing experiments and measurements to be performed directly in hardware. By performing the analysis directly within the NIC, the performance bottlenecks of the PCI bus and potential inaccuracies of the Operating System can be alleviated. Using a NetFPGA operating as a simple switch, the queue occupancy for every *enqueue*, *dequeue* and *drop* operations can be monitored at a granularity of 8ns from the internal 125MHz clock, as well as the traversal delay of each packet based on the instantaneous queue occupancy.

Figure 3.3 shows the experimental setup used by the following experiments in this section. Connected to the NetFPGA are two hosts, one acting as the worker node and a second one as the aggregator, emulating a partition-aggregate scenario. Due to the low port density of the NetFPGA platform, providing only 4×1Gbps ports, and the requirement for a dedicated monitoring port, it is necessary for these experiments to reproduce the required oversubscription ratio without a large number of nodes. In a real deployment, the oversubscription would come from many worker nodes connected to one aggregator through similar speed links, for instance 1Gbps. However, in the following experiments the oversubscription must be obtained with different bandwidth between the switch, worker and aggregator. Considering the hardware limitations of the rate limiter module and a realistic oversubscription ratio of 8:1 (see §2.3.2), the uplink bandwidth, between the switch and the aggregator has been set to 126Mbps.

In order to generate a TCP incast throughput collapse between the workers and the aggregator the traffic must contain a large number of flows and be highly synchronised. To emulate the traffic pattern of workloads of common data centre partition-aggregate applications, a burst

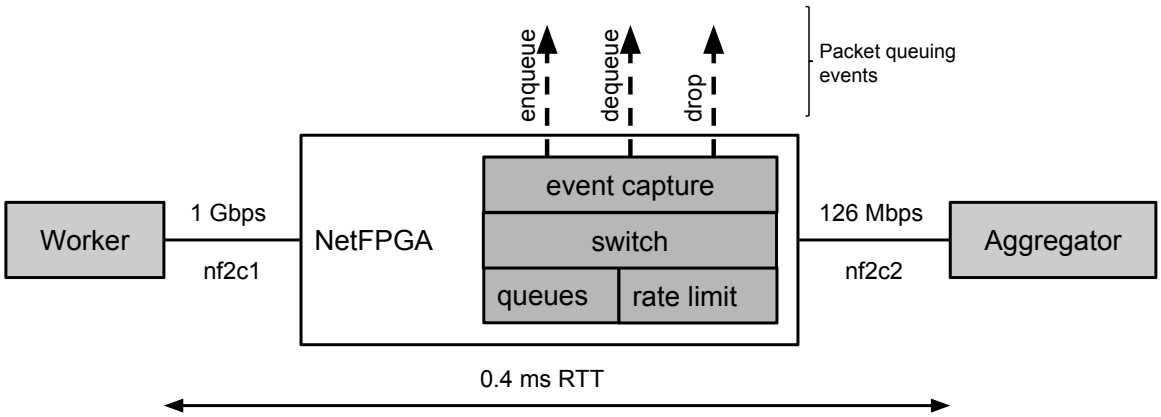


Figure 3.3: NetFPGA experimental setup for buffer occupancy monitoring

of 100 synchronised TCP flows is started from the worker towards the aggregator. This traffic has been defined to match the reported number of flows per host (>10, 50% of the time) for 10 hosts replying using short-lived, delay-sensitive mice flows to a partition-aggregate query [62]. In the following experiments, each flow is 20kB as it matches Cisco's definition of mice flows [115], is representative of published DC network traffic characteristics [93, 62] and is used extensively in DCTCP's mice flows experiments [95]. In the following experiments, the worker and aggregator nodes are both running Linux with the kernel version 3.19 and the default TCP congestion control parameters with TCP Cubic for congestion control. The following subsections evaluate experimentally the impact of buffering, congestion window, and retransmission timeouts on the performance of TCP, and their relationship to TCP throughput incast collapse.

3.3.1 Deep and Shallow Buffering

Traditionally, TCP incast collapse and the resulting low throughput has been tackled by using deep-buffered switches, reducing packet drop during congestion events, and ensuring that the link is always utilised. This approach has been used as it was an easy way to hide the issue. With more expensive switches with larger buffers, bigger bursts of traffic can be handled mitigating the performance degradation of TCP throughput incast collapse. However, as a side effect, these deeper buffers induce large delays, retransmission synchronization,

unpredictable end-to-end RTT, and prevent the congestion control algorithms to react in a timely fashion [116]. By hiding the congestion event, large buffers prevent the end hosts from adapting their sending rate to recover from congestion and artificially increase the BDP of the network. The original "rule of thumb" has been to size each buffer to be equal to the BDP of the network. Appenzeller et al. have suggested that the buffer should be a fraction of the BDP depending on the square root of the number of active flows [117]. Although, this proposal stem from the unique traffic characteristics of backbone networks, in which the traffic can be considered independent and unsynchronized. More recently, it was suggested to size the buffers depending on the capacity ratio between the input and output links and the congestion control algorithm [118]. Altogether, recent researches suggest that buffers are too large for both short bursts and long lasting flows, and that smaller buffers should be used.

In this section, the trade-off between latency and throughput is highlighted when using a deep-buffered switch of 512kB and shallow-buffered switch of 85kB while using TCP default congestion control parameter settings. In the following experiments, 100 flows are started at the same time to generate a short burst of multiple mice flows typical of partition-aggregate traffic. Each experiment was repeated 5 times, and the figure in each experiment represents a typical run. Table 3.1 summarises the mean and standard deviation for the different experimental runs.

Table 3.1: Run statistics of 5 rounds of NetFPGA experiments

Experiment	Goodput (Mbps)	Completion time (ms)	Drops	On-time (ms)	Off-time (ms)
IW=10 minRTO=200 Buf=512kB	56.844 (0.899)	274.95 (4.360)	529.667 (80.384)	136.836 (1.915)	138.110 (6.263)
IW=10 minRTO=200 Buf=85kB	24.99 (0.118)	32.64 (3.010)	1075 (17.114)	137.855 (0.085)	494.783 (3.093)
IW=1 minRTO=200 Buf=85kB	36.857 (0.207)	423.94 (2.372)	82.456 (4.541)	135.744 (2.046)	288.349 (4.568)
IW=1 minRTO=1 Buf=85kB	94.506 (0.967)	165.349 (1.682)	221.323 (19.293)	110.867 (2.022)	54.484 (3.704)

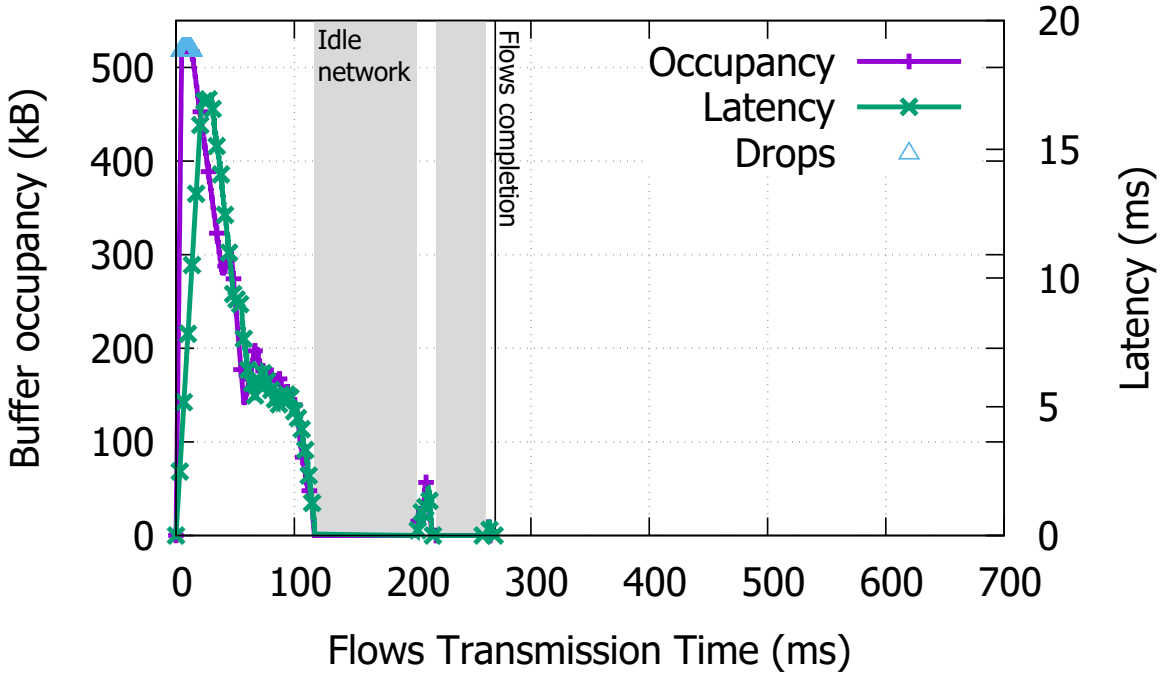


Figure 3.4: Impact of deep-buffers on FCT. (512kB per port), IW=10, minRTO=200

Figures 3.4 and 3.5 show the queue size growth and the packet traversal latency of the 100 flows for a deep and shallow buffered switch, respectively. In those figures, the 100 flows are all started at time $t = 0$ and the slowest flow from this burst completes when the last packet is received, marked as *Flows completion* on the figures. Figure 3.4, shows that using the default Linux TCP congestion control parameters with a deep buffered switch, the buffer occupancy grows very rapidly at the beginning of the burst, resulting in numerous packets dropped and an idle time waiting for the senders to retransmits the lost packets. As a consequence of this idle period, some flows are delayed resulting in the last flow to complete after 270ms. Additionally, as the buffer occupancy grows, the per-packet latency spikes at the beginning of the transmission, with the latency increasing to 16.9ms when the buffer occupancy reaches its maximum size of 512kB. The idle latency of the experimental network is around 400μs, therefore this spike results in a 37$\times$ increase in latency from the idle RTT. The different TCP variants alter the slow-start phase or congestion-avoidance phase of the transmission but rely on the same value for the initial window, resulting in the initial burst at $t = 0$ to be the same regardless of the TCP variant used. Similarly, without enough packets to generate a fast retransmit, the idle delay between retransmissions will be bound by the minimum

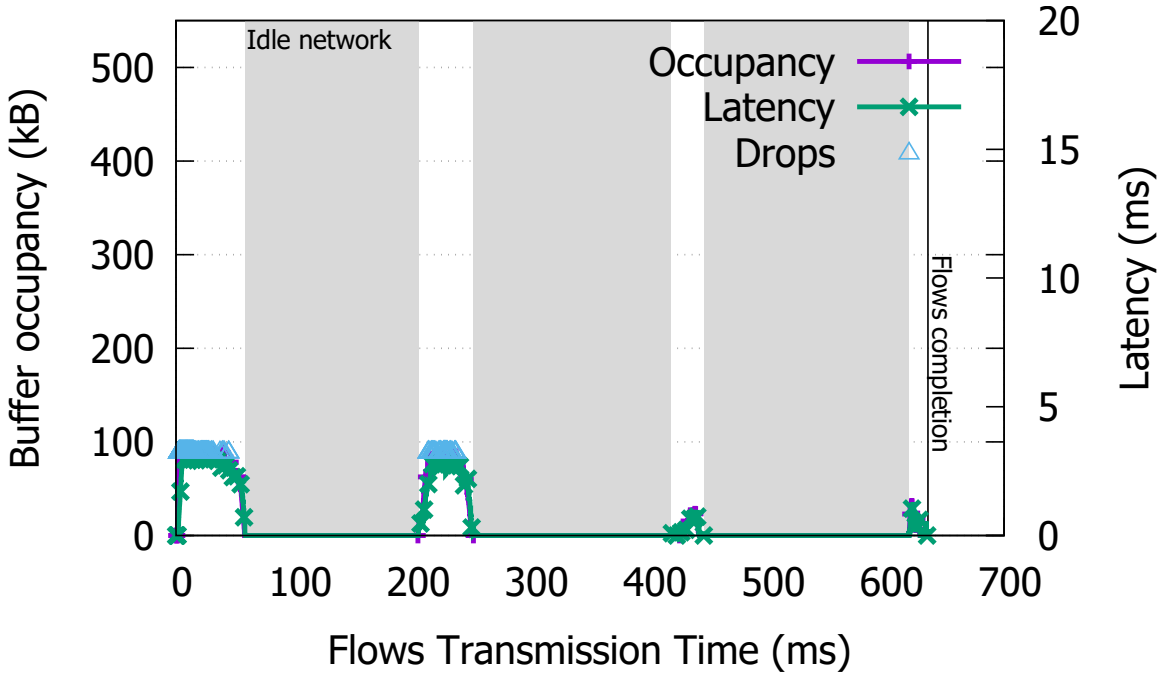


Figure 3.5: Impact of shallow-buffers on FCT. (85kB per port), IW=10, minRTO=200

retransmission timeout.

In Figure 3.5 the same experiment is repeated for a shallow-buffered switch, the 100 flows are started at time $t = 0$ and the last flow completes at $t = 636ms$. The maximum latency observed is 5ms that results in a $12\times$ increase from the idle RTT. From a FCT perspective the deep-buffered switch completes flow transmission $2\times$ faster than the shallow-buffered switch. However from a packet delay aspect, shallow-buffers result in an almost $8\times$ reduction in worst-case delay. Due to the smaller buffer and the large congestion window size, numerous packets are dropped at each synchronised retransmission event, resulting in three minRTO-long idle periods. Therefore, depending on the requirements of the applications, large buffers can provide higher throughput and FCT but at a cost of high latency, while shallow buffers can provide low latency at the cost of low perceived bandwidth and long FCT. In either case, the network is underutilised, visible in the figures by the gaps with zero buffer occupancy when the application awaits for the retransmission timer (RTOmin) to elapse and resend the dropped packets.

The throughput achievable by a flow at the switch is an important metric as it defines the pace at which the data is sent. However, in protocols such as TCP where packets are retransmitted

Table 3.2: Performance impact of buffering

Buffer size	Goodput (Mbps) (net. util.)	Completion time (ms)	Packet drop	Delay avg/-max (stddev)	On-Off ratio
512KB	56.94 (45%)	274.42	585	6.0/16.9 (5.1)	1.003
85KB	24.58 (20%)	635.65	1058	1.7/3.0 (1.2)	0.277
85KB (tuned)	94.11 (75%)	166.03	213	1.8/3.0 (1.1)	1.961

on failure, the throughput does not reflect the number of **useful** bits delivered. A more representative metric is the end-to-end goodput, which defines the amount of data transferred excluding retransmissions and protocol overhead. Table 3.2 summarizes the transmission performance in terms of goodput and overall completion time for the burst of 100 synchronised TCP flows, the number of packets dropped at the switch due to overflow, and the occupied-to-empty buffer ratio. In both the shallow and deep buffered switch experiments, the network utilisation remains low. Even for a deep-buffered switch, the maximum network utilisation is 45%. The ratio of empty-to-occupied buffer is shown as the On-Off ratio describing the transmission ripples in Figures 3.4 and 3.5. The *On* and *Off* periods are defined by the transmission time with non-empty and empty buffers respectively. For the deep-buffered switch, half of the overall transmission time is idle and for a shallow-buffered switch, over 70% of the transmission time is idle.

These experiments highlight the misbehaviour of TCP in the very high-thoughput low-latency environment with a partition-aggregate traffic pattern. Deep buffered switches can partially mitigate throughput incast collapse, but this comes at the cost of high and variable latency. On the contrary, shallow buffers lead to low throughput and completion time, but with a much smaller impact on latency. Hence, the traditional approach of using deep-buffers only partially solves the problem, and introduces high and variable latency for soft-realtime flows. Therefore shallow-buffers are necessary to achieve low-latency transmissions and the congestion control must be adequately configured in the end-hosts and not masked by the network infrastructure.

3.3.2 Initial Congestion Window

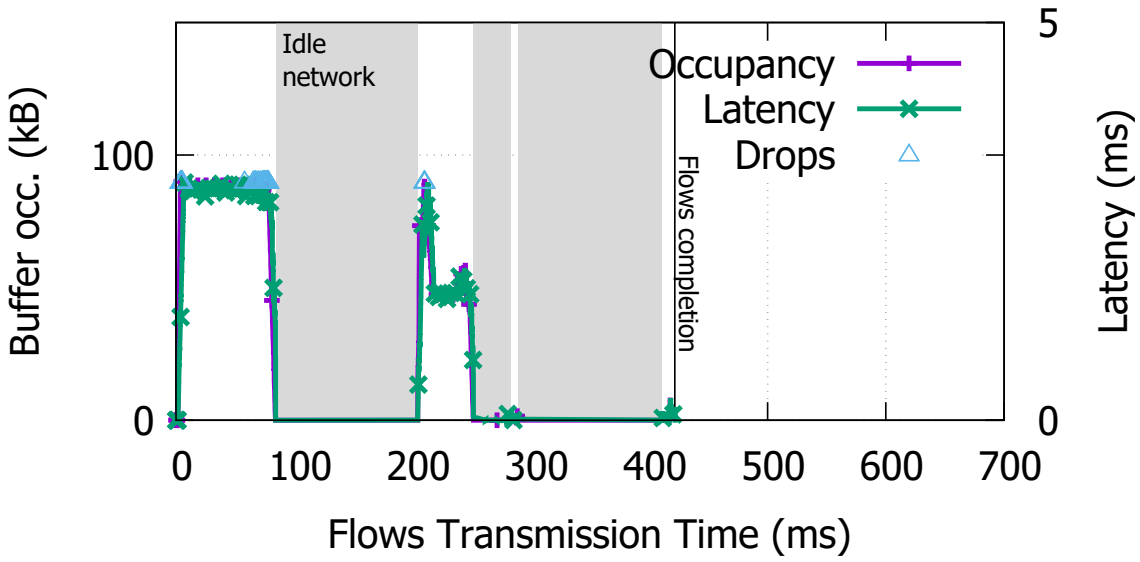


Figure 3.6: Configuring the IW close to BDP, IW=1, minRTO=200

In this experiment, the impact of the congestion window on TCP incast collapse, and particularly the behaviour of the IW, is evaluated. In Figure 3.6, the shallow-buffered switch experiment performed in Figure 3.5 is repeated as OTCP focusses on limiting the maximum latency and FCT. However, the IW is decreased to 1 MSS (the closest value to the BDP) considering 100 synchronised TCP flows. As a result, the completion time of the last flow is 422ms, a reduction of over 200ms in FCT is observed compared to the default congestion control parameters. This 30% reduction in FCT, highlights that the default IW of TCP is too aggressive for low-latency high-throughput environments causing overflow of the buffers, and resulting in significant packet loss.

This experiment highlights that the IW, increased to 10 Maximum Segment Size (MSS) in RFC 6928 from 2–4 MSS in RFC 3390, can significantly impact the performance of high-throughput, low-latency networks. With an IW of 10 segments, a single flow can send up to 15kB of unacknowledged data after the three-way handshake is complete. When 100 synchronised flows are transmitted, each flow with an IW of 10 packets and each packet of maximum size (1500 bytes) the initial burst of traffic sums up to 1.5MB, resulting in numerous dropped packets. In this experimental topology, the buffer of the switch is 85kB and the

output link is oversubscribed by a 8:1 ratio resulting in a very quick overflow of the buffer as the initial burst of traffic is sent. To prevent the large packet loss with the default TCP congestion control parameters, the buffer size should be increased proportionally to the oversubscription ratio as discussed in [118] but would result in the issues described previously of long and unpredictable delays, synchronisation of the flows and hiding the congestion of the network from the end-hosts.

The value of 10 MSS might be suitable for WAN environments with long fat pipes carrying few long-lived large flows, but it is unsuitable for a DC environment with numerous concurrent mice flows competing for high throughput over extremely low latency links. The IW and other initial TCP congestion control parameters should be conservative for a new flow to achieve a fair share of the network resources once the network properties are measured. However, the value of 10 MSS for the IW is over-aggressive, and a new flow can operate far above its fair share when starting and must rely on packet loss to trigger congestion avoidance mechanisms and reduce the congestion window.

IW should be configured with respect to the BDP of the network and the number of concurrent flows as it represents the amount of unacknowledged data on the path between two endpoints shared fairly amongst competing flows. However, because of the switches' buffers, the BDP is artificially increased as the delay changes from sub-millisecond when the network is idle and the buffers are empty to tens of milliseconds when the buffers are highly occupied. Therefore IW should be configured based on the BDP of the network when all the buffers are empty, to represent the minimum fabric latency. In this case, IW matches the minimum number of bits that can be sent unacknowledged to achieve line-rate without requiring any queuing in the intermediary switches. This would result in the maximum throughput and minimum latency achieved by the fabric between the two endpoints. Practically, as the packet header size is constant, to maximize goodput, the overhead of transmission must be as low as possible. Therefore, the minimum congestion window should be 1 MSS, which might overall be higher than the BDP if the number of concurrent flows is high.

3.3.3 Retransmission Timeout

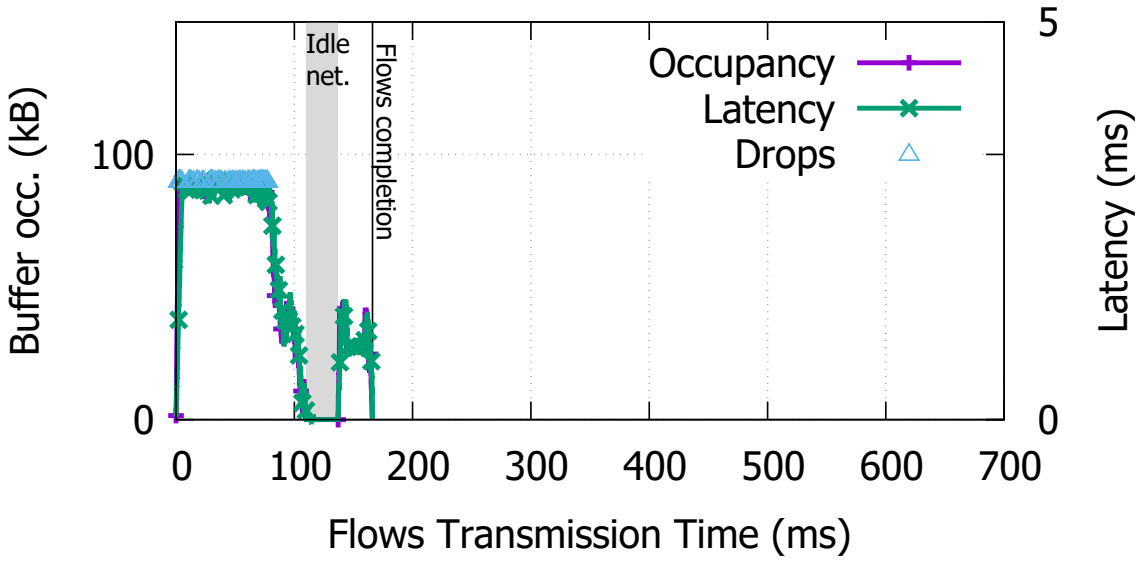


Figure 3.7: Configuring IW and minRTO, IW=1, minRTO=1

In the experiments presented previously, the on/off transmission caused by TCP incast traffic can be observed. For a short period of time the throughput is high and the buffers are fully occupied. This is followed by an idle period with no traffic and empty buffers waiting for the retransmission timeout to retransmit the dropped packets. Previous work on TCP incast collapse has shown that reducing the minimum bound on the RTO can prevent the long off periods between transmissions resulting in a shorter FCT [94]. In this experiment, the performances of configuring both the IW and RTO are evaluated.

The retransmission timeout is the default mechanism of TCP to recover from packet loss, and is set based on Van Jacobson's RTO estimator, which relies on the measured RTT of packets known to have been transferred without retransmissions, as well as the variation of the RTT. This value is bound to a minimum of 1 second set by RFC6298, but it is not honoured by the operating system implementations that vary their RTO between 200 and 400ms [119]. The minimum bound on the retransmission timeout is used to avoid spurious retransmissions based on the link latency and possibly worsening congestion and reducing goodput. The use of the RTT variation to calculate RTO highlights the requirements for a stable latency in the network in order to minimise idle time. It is important to understand that setting RTOmin

does not mean that all packets will be retransmitted after this time, rather the RTO estimator is still operational and the measured RTT will still be used as well as its variation to calculate a suitable RTO with a minimum value of RTOmin.

In Figure 3.7, the congestion window is close to the BDP to limit initial bursts of traffic and RTOmin has been reduced to 1ms. In this experiment with the IW and minRTO reduced, the the last flow completes at $t = 166ms$ and the maximum latency is 3ms. With the lowered value of minRTO, the retransmission timeout calculated can match the network characteristics, resulting in an idle period of less than 25ms when waiting for the end-hosts to retransmit the lost packets. As shown, with a small IW and RTOmin, the FCT is 1.5× faster than with the default congestion control parameters. The packets dropped 2.7× lower as the IW has been reduced to match the BDP of the network and the number of concurrent flows. Finally, with the improved network utilisation and reduced packet loss, the goodput is 1.7× higher, while latency is low and stable as shown in the last row of Table 3.2.

3.3.4 Discussion

The previous three evaluations highlighted that high throughput and low latency are achievable in DC networks with partition-aggregate workloads but require fine tuning of multiple parameters dependent on the infrastructure. The size of switch buffers is not strictly a congestion control parameter, but significantly affects the algorithm's behaviour by hiding or delaying the congestion events. Using shallow-buffered switches, the latency can be kept low, but TCP's large default IW quickly overflows the buffers, resulting in low throughput. Adequately tuning the IW to reduce the initial burst of packets and subsequent overflow in the intermediate switches, and also tuning RTOmin to prevent long off periods between transmissions, the overall FCT of synchronised flows can be greatly improved.

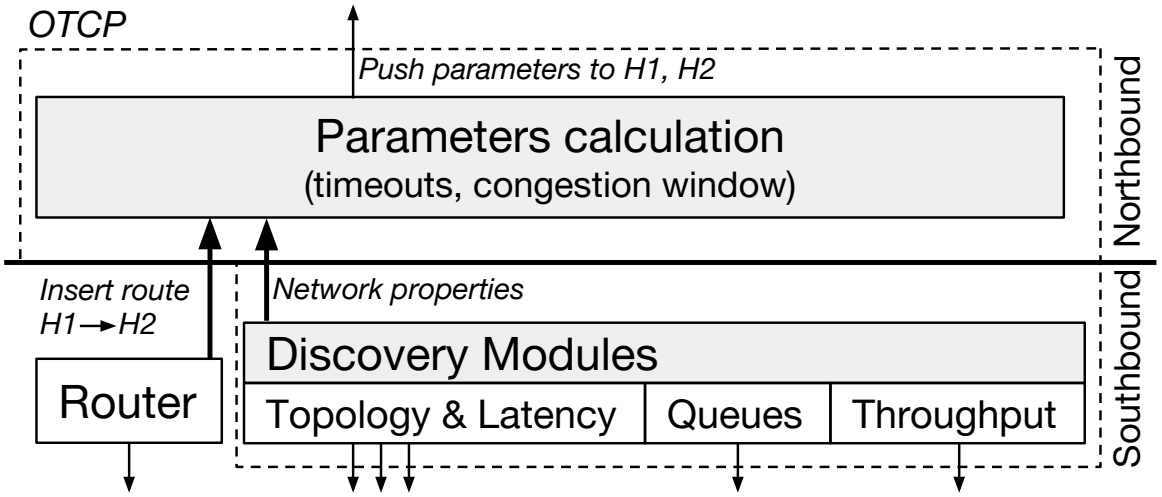


Figure 3.8: Overview of OTCP architecture.

3.4 Omniscient TCP

Omniscient TCP (OTCP) is presented as a tuning approach for TCP based on network-wide information available from the controller in SDN-based DCs. OTCP aims to demonstrate that traditional control loops can be altered in DCs based on metrics and information available from the different layers of the infrastructure. OTCP addresses the impairments of TCP under partition-aggregate workloads to achieve high throughput and low and stable latency, with commodity shallow-buffered switches. It achieves this by configuring TCP congestion parameters on a per-route basis based on the end-to-end network characteristics including topology, latency, throughput, and buffering. The congestion window is configured to match the distinct BDP of each route, and the initial burst of traffic can be reduced, limiting the number of packet drops, increasing goodput and fairness for the flows. Using OpenFlow, static network characteristics of large-scale networks are collected when the controller-to-switch connection is established.

Since the switching latency and buffer size of switches are static, this approach only requires incremental updates to the measurements on topological alterations and therefore is not necessary to be performed continuously to mitigate TCP incast collapse. An overview of the OTCP architecture is shown in Figure 3.8, showing the separation between the Southbound interface responsible for collecting the metrics of interest in the network, and the Northbound

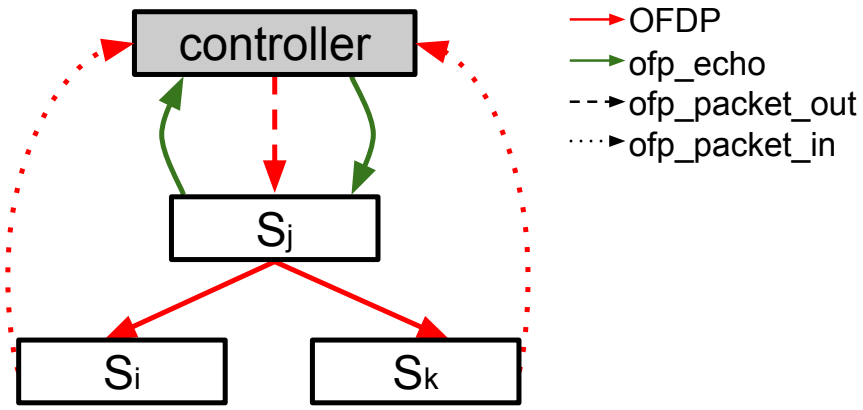


Figure 3.9: Latency gathering at the controller using OpenFlow.

interface interacting with the end-hosts to adjust the congestion control parameters. In the following discussions a single logical controller is responsible for collecting and gathering the network metrics. However, for this approach to be scalable over a larger number of hosts and network devices, the controller should be physically distributed over multiple nodes for a large provider network, as provided by controllers such as OpenDaylight or ONOS. In a multiple node configuration, the measurements should be performed the same way as with the single controller, with each controller managing its own cluster and sharing the metrics with the other controllers.

3.4.1 OpenFlow network information gathering

To calculate suitable congestion control parameter settings for the congestion window and the retransmission timeout, readily available information including latency, buffer sizes, and link rate must be collected by the controller.

End-to-end latency

To calculate the minimum bound on the RTO as well as the BDP, the idle latency of the switching fabric must be measured. OpenFlow (OF) does not provide any direct functionality to measure latency, but the protocol can be fitted to allow accurate measurements. A common approach for topology discovery in an OF network is to use the OpenFlow Dis-

covery Protocol (OFDP), a variant to the Link Layer Discovery Protocol (LLDP), which allows custom Type-Length-Value (TLV) structures to be inserted in the packet payload. By storing the current timestamp in the TLV payload, OFDP can be used for both topology discovery and latency measurement. The controller generates an OFDP packet, issues a **ofp_packet_out** command to every switch to flood the packet to its neighbours, upon reception the neighbours forward the OFDP packet back to the controller using **ofp_packet_in**. As shown in Figure 3.9 this measurement gives us the 3-hop latency from controller to S_j, S_j to S_i and from S_i back to the controller. This latency is a combination of the management and data network latency and includes the time taken by the control plane to encapsulate and decapsulate the OFDP message which can be significantly longer than data plane latencies as discussed in [120].

The controller-to-switch latency is measured using OpenFlow **ofpt_echo_request** and **ofpt_echo_reply** usually used by the switch as a keep-alive signal. Using this approach to measure the latency allows the control plane processing time to be included as part of the measurements. Therefore, using this measurement as well as the 3-hop latency measured during topology discovery the latency from S_i to S_j can be inferred. Consequently, the latency between two arbitrary switches (S_i and S_k) through the route R is the sum of the latency of every link traversed (equation 3.1).

$$L_{S_i \to S_k} = \sum_{R_{i \to k}} L_{S_i \to S_{i+1}} \tag{3.1}$$

These measurements provide accurate latency estimate of the fabric, but do not include the latency between the ToR switches and the hosts. The controller generates an ARP Probe packet and sends it from the ToR switch to the host using a **ofp_packet_out** message [121]. The host's ARP response is then intercepted at the ToR and sent to the controller. Hence, the RTT between the switch and the host can be calculated based on the controller-to-switch delay from this 4 hops latency. This approach is especially suitable in environments where the controller is used as a central directory service for the ARPs such as in VL2 [62]. Once the switch-to-switch and host-to-switch latencies are measured, the minimum retransmis-

sion timer bound (RTOmin) can be set to the round-trip delay between the two hosts. It is worth noting than the previous discussion assumes that the forward and return paths are the same, hence the forward link latency and the return latency are equal. However if the links are asymmetric the calculation can be easily modified to take into account each latency independently.

Buffer Size

Buffering in the switches impacts the latency of transmission of packets, as a packet can be delayed by the time it takes to transmit all other queued packets. A maximum bound on the latency can be calculated using the maximum buffer occupancy of each switch and the egress link rate. If queues have been assigned to switch ports, the OpenFlow controller can send a **ofp_queue_get_config** packet to retrieve the queue characteristics including the length in bytes as well as the minimum and maximum data-rate (since OpenFlow 1.2). Therefore, the maximum bound on the retransmission timer can be characterised by the idle latency (RTOmin) plus the switches' queue delay. For any switch s along the route (R) from H_1 to H_2, the delay is the buffer size (Q) divided by the egress link rate (T) (equation 3.2).

$$RTO_{max}(H_1 \rightarrow H_2) = RTO_{min}(H_1 \rightarrow H_2) + \sum_{s \in R} \frac{Q_s}{T_s} \tag{3.2}$$

The initial retransmission timeout (RTOinit) can also be set to the same value as RTOmax, but some additional delay can be caused at the end hosts, for instance if connections have been backlogged or the machine is highly utilised. To evaluate the impact of a highly utilised machine on TCP transmission latency and variation, the testbed machines were artificially heavily loaded. By stressing source and destination CPU cores to 100% utilisation, high number of IO operations and numerous memory operations, the maximum increase in latency observed in the experiments was 131μs. In TCP, RTOinit is one order of magnitude larger than RTOmin. In OTCP, and in order to account for the possible increase in latency on highly loaded hosts, RTOinit is set to twice RTOmax.

Link Rate

To associate a rate to each link of the topology, the controller listens for **ofp_port_status** asynchronous messages containing the port operating mode (10Mb, 100Mb, 1Gb, 10Gb) as well as the medium used by the link. If topology discovery is used, this event will also be used to keep track of topological changes such as link failure, addition and removal. Knowing the latency (L) between H_1 and H_2, and the maximum sending rate between those two points (T) characterised by the lowest link rate along the route (R), the maximum value of the congestion window ($CWND_{max}$) can be calculated as the BDP (equations 3.3, 3.4).

$$BDP_{H_1 \to H_2} = RTT_{H_1 \to H_2} \times T_R \tag{3.3}$$

$$CWND_{max}(H_1 \to H_2) = BDP_{H_1 \to H_2} \tag{3.4}$$

By integrating simple measurements to the topology discovery and querying the switches configuration parameters, an SDN controller is able to collect and manage the metrics required to calculate finely-tuned congestion control parameters for the operating environment. As described previously, these measurements are only required when the operating parameters are modified, such as the replacement or addition of links and switches and, are not continuously performed.

3.4.2 OTCP parameter propagation

To distribute the congestion control parameters, the controller exposes a JSON/REST northbound API to which a user-space daemon in the end-hosts connects. When the controller updates the route characteristics, the daemon is notified with the updated values, and the route entry associated is updated. In the Linux kernel, since kernel version 2.6.23, most TCP congestion control parameters can be configured from user-space for specific destination IP addresses or subnets using *netlink* to configure the kernel routing table. However, the initial

retransmission timeout cannot be configured without a simple kernel modification (16 lines of code in this implementation). These parameters are used by newly established flows instead of the default conservative ones. Such approach requires each end-host to execute a small custom daemon (<50 lines of code) connecting to the controller. However, with the increasing importance of network virtualization and virtual machines, end-hosts are already heavily customized, hosting hypervisors, monitoring software, and other daemons.

On port or switch failure, the controller is notified either with a **ofp_port_status** packet sent by the switches when a port changes state, or of device failure when the TCP connection is torn down. When such event is received, as typical in an OF environment, the new route is computed by the controller and the associated congestion control parameters are also recomputed and sent to the relevant end-hosts. As these events are triggered by link removal, it is not required to perform new measurements as the new path will be along links and switches with already known characteristics.

Using the kernel routing table to configure congestion control parameters, allows the daemon on the end-host to be extremely lightweight. However to improve network performance, routing strategies such as Equal-Cost Multi-Path (ECMP) can be used to load-balance the traffic amongst multiple paths. In such a scenario, the destination-based entries in the routing table might not be sufficient, and therefore the daemon should be designed to configure individual sockets when they are created. As far as the controller is concerned the logic is similar, simply requiring parameters to be calculated for the different available routes.

3.4.3 Initial Congestion Window

The size of the initial congestion window is dependent on the maximum congestion window as well as the number of active flows in the network during the slow-start phase. Setting the congestion window too high with respect to the BDP will lead to queue build-up on the traversed switches, and consequently to higher latencies and unfairness in flow throughput. Setting the congestion window too low will require more RTTs to reach the bottleneck capacity of the link, lengthening the slow-start phase and therefore increasing FCT. In OTCP,

CWNDinit (IW) is set to a fraction of the *CWNDmax*, since *CWNDmax* represents the maximum congestion window assuming a single flow along the path. When the path is shared between multiple flows, the maximum congestion window of each flow should be *CWNDmax* divided by the number of active flows (α) in the link (equation 3.5). In low latency, high-throughput environments, *CWNDmax* is small, hence, as number of flows α increases, the *CWNDinit* will quickly reach its minimum value of 1 MSS. In the following experiments, α is set to 100 to allow a congestion window for 100 flows going through the same bottleneck link.

$$CWND_{init} = min(1, \frac{CWND_{max}}{\alpha}) \tag{3.5}$$

In the current implementation, α can be configured either manually to a value matching the maximum number of flows that will fan-in from the workers, or updated at run-time by polling the flow statistics of switches using **ofp_flow_stats_request** similarly to the approach used by Hedera [122]. However, Hedera's approach requires every single TCP flow to have a matching entry in the switches' flow table. This requirement is impractical in very large-scale networks in which the number of concurrent TCP flows is larger than the maximum flow entries a switch can support (few thousands in commodity switches). In static deployments, the value of α could be calculated ad-hoc by measuring the number of concurrent flows for the running applications, for instance using NetFlow, sFlow, or a traffic capture at the bottleneck switch or alternatively at the end-host, if the topology is known. Further discussion on the measurement and calculation of α is covered in sections 4.1 and 4.5.4.

These improvements solve TCP Incast collapse by allowing multiple mice flows to efficiently (goodput) use the network fabric without suffering from long latencies which are problematic for soft-realtime traffic.

3.5 Evaluation

This section shows how OTCP can successfully mitigate TCP incast collapse in DC environments by improving **latency**, **throughput**, **goodput**, and **fairness** based on the metrics available in DCs. The performance evaluation is performed on a Mininet HiFi 2.2 emulation network for large-scale experimentation [123]. Through the following evaluation, the ability for a central controller to accurately collect network properties and distribute finely tuned metrics to the end-hosts is demonstrated.

Latency: Through the use of (a) shallow buffers to avoid large variations in packet delivery, (b) a small congestion window to avoid queue overflow on bursts, and (c) a lock on the maximum congestion window matching the route BDP, Omniscient TCP prevents buffer occupancy in the switches from growing exceedingly large during incast events.

Throughput & Goodput: Configuring RTOmin and RTOinit to match the minimum network latency allows higher throughput by preventing long idle periods. A small congestion window leads to better goodput by requiring less packet retransmissions.

Fairness: With an initial window size of 10 segments, the first flows sending their initial window will complete quickly but overflow the queues, therefore subsequent flows will experience high packet loss and long delays. With a smaller congestion window, many more flows will be able to send packets creating a much more interleaved packet distribution, therefore increasing fairness.

3.5.1 Experimental Setup

Mininet was selected for the experiments as it allows large-scale network topologies to be created easily and relies on the underlying OS for the networking stack and applications. Relying on the kernel allows the experiments to demonstrate how existing end-hosts would behave over such infrastructure without introducing artefacts from the simulation system's TCP implementation. Mininet has also been shown to be highly reliable for the reproducibility of network experiments [124]. Early work on OTCP was performed in simulation using

the popular NS-3 network simulator, but limitations made the simulation results widely different from real hardware such as the NetFPGA platform. One of the limitation is the way ARP requests are resolved, relying on a global cache that is limited to 3 concurrent requests and therefore preventing the burst of concurrent TCP flows expected in a TCP incast scenario. Another limitation was the integration of the Linux TCP stack. NS-3 supports through the Network Simulation Cradle (nsc) the Linux kernel stack to be used, but only supports very old version of the kernel (2.6.26) that are not representative of today's implementations.

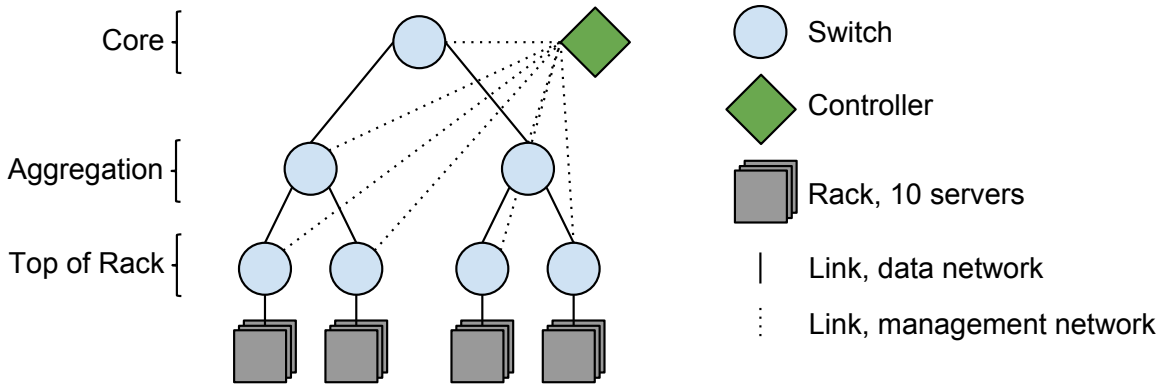


Figure 3.10: OTCP Emulation Network Topology

The experimental topology for the evaluation of OTCP, illustrated in Figure 3.10, has been set up as a typical three-layer tree with an increasing over-subscription ratio and latency at the higher layers of the data network (ref. to 2.3.2). This setup has been used to simulate the bandwidth, latency and oversubscription expected in a production grade DC [62, 95]. To achieve a 10:1 oversubscription at gigabit rate, each rack contains 10 hosts connected with an Ethernet gigabit link to the ToR and an egress link from ToR to Agg also at 1Gbps. To match with the latencies varying from hundreds of microseconds within the same rack, to few milliseconds for traffic traversing the aggregation and core layers, the latency of the link from host-to-Tor is set to 100μs. This value results in a host-to-host RTT of 0.4ms, matching published values [94]. To achieve millisecond latency at the Agg layer, ToR-to-Agg latency has been set to 0.2ms resulting in a 1.2ms latency and finally Agg-to-Core to 1ms to get cross DC latency of 5.2ms. The network is designed to follow the IEEE 802.3 Ethernet standard with a Maximum Transmission Unit (MTU) of 1500 bytes and therefore not supporting Jumbo or mini-Jumbo frames [125].

To match the shallow buffers of commodity switches, all switches have egress queues of 60 packets regardless of the layer of the topology, with a drop-tail mechanism and without any AQM for early congestion notification or drop. Finally, the Linux kernel for the Mininet host is 3.19.4 with the stock TCP parameters using TCP CUBIC, a 10 MSS IW, a RTOmin of 200ms, and a RTOinit of 1s (decreased from 3s since version 3.2 of the kernel). Both the northbound (REST) and southbound (OpenFlow 1.3) interfaces have been implemented in a single Go controller designed for the purpose of OTCP managing OpenvSwitch (OvS) software switches in Mininet. To prevent the management traffic from impacting the production traffic and vice-versa, the traffic to the controller is on a separate out-of-band network. The centralised controller exposes an OpenFlow 1.3 southbound interface responsible for the routing, forwarding, measurement of path latency and link rate as described in section 3.4.1.

Addressing of the hosts follows the convention 10.*AggId.ToRId.HostId*, therefore the /24 subnet mask maps to all the hosts under the same ToR, the /16 mask to all the hosts under the same Agg and a /8 mask for any flow that must cross the Core of the network. Providing location information in the addressing as well as the symmetrical nature of the topology allows OTCP to insert only 3 entries in the routing table of the end-hosts based on the destination subnet (ToR, Agg and Core routes). This approach allows the routing table to be kept small for extremely large data centers, but assumes symmetry in the topology. In case it is necessary for individual route metrics to be stored, the propagation scheme should be modified to only propagate the routes that are relevant to the current host. In a large data centre, the routing table could grow very rapidly if an entry is stored per destination, resulting in slower lookup times and potentially longer flow establishment times. However, each server only communicates with a fraction of the other nodes and therefore it is unnecessary to keep an exhaustive list of routing entries if the inherent symmetry of the network cannot be used.

3.5.2 OTCP measurements

First, the ability for the controller to collect the measurements defined in section 3.4.1 from the network infrastructure using OpenFlow is evaluated. These measurements should be

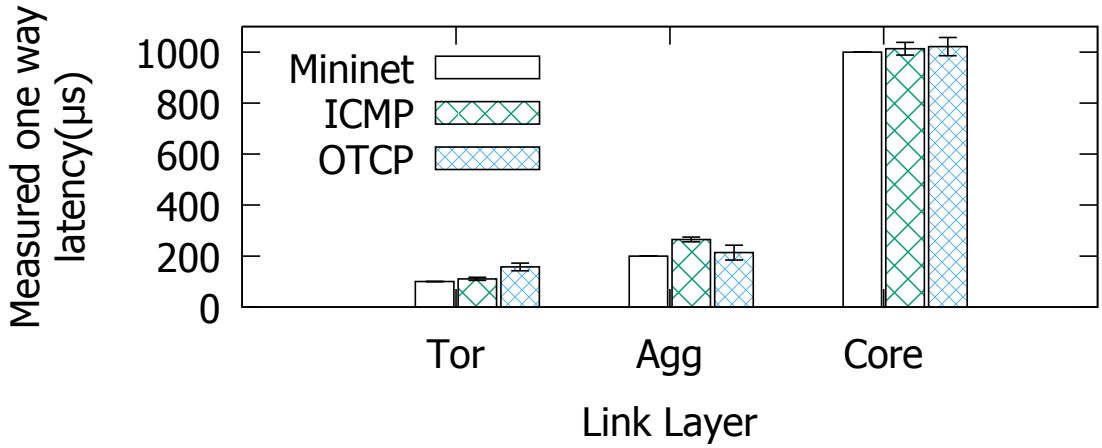


Figure 3.11: Switch-to-switch and host-to-switch latency measurement comparing the topological settings, ICMP and OTCP.

conducted when the network is idle, when the buffers are empty, as the objective is to bind RTOmin to the fabric latency, and setting the congestion window to a value achieving maximum throughput without requiring buffering in the intermediary switches. Figure 3.11 presents the mean latency of 100 independent measurements comparing the latency configured in the topology (Mininet, Figure 3.10), ICMP ping, and OTCP latency measurements. From these measurements, it can be seen that the latency of each link can be measured accurately enough even in low latency environments with OTCP and ICMP returning similar metrics close to the baseline.

Table 3.3: OTCP calculated route parameters

	RTT (μs)	RTOmin (ms)	RTOmax (ms)	RTOinit (ms)	CWNDmax (MSS)	IW (MSS)
ToR	629	1	2.069	4	49	1
Agg	1485	2	5.805	12	127	2
Core	5571	6	12.771	25	476	5

Table 3.3 shows the route parameters, grouped by layer, calculated by OTCP based on the latencies shown in Figure 3.11, and the buffer size as well as throughput the of the network. Due to the time granularity of the Linux kernel, RTOmin has a minimum integer value of 1ms, therefore the RTT has been rounded to the highest integer. RTOmax represents the maximum delay the network fabric can have considering the egress queue of each switch traversed is fully occupied. In the topology, the maximum queue length (Qs) regardless of

the switch layer is 60 packets to represent a shallow buffered switch [95]. In the worst case scenario each packet is 1500 bytes, a full Ethernet Maximum Transmission Unit (MTU). Therefore, a fully occupied queue can buffer up to 90000 bytes ($Qs \times MTU$) waiting to be transmitted on the egress port. The egress link has a throughput of 1Gbps (Ts) and therefore the maximum delay waiting for the 90000 bytes to be transferred is 720μs (($Qs \times MTU$) $\div$ Ts) per queue traversed. As stated in section 3.4.1, in OTCP RTOinit is set to twice the RTOmax and must be expressed in milliseconds similarly to RTOmin. CWNDmax is set to match the BDP of the network, therefore it is the RTT times the bottleneck throughput divided by MSS. Dividing the BDP by MSS is required as the Linux kernel expresses the congestion window as a multiple of the MSS. Finally, IW is set to match the number of expected 100 synchronised flows (α).

3.5.3 Flow Completion Time

TCP, OTCP and DCTCP perfomance under TCP incast are evaluated with respect to the mean overall FCT to show overall behaviour, and at the 95th percentile to highlight unfairness between competing flows. DCTCP is a variant of TCP designed to reduce buffer build-up through Explicit Congestion Notification (ECN). Instead of the common approach of treating ECN markings as a packet loss and react by halving the congestion window, DCTCP uses multiple ECN marks to adapt the sending rate [95]. This approach requires support from both source and destination hosts as well as ECN support in intermediary switches, but can reduce buffer utilisation by 90% resulting in lower latencies. Additionally this evaluation introduces ODCTCP a combination of OTCP and DCTCP. ODCTCP performs the same fine tuning to the congestion control parameters but instead of operating with a classic TCP stack, DCTCP is used. In ODCTCP, the OTCP portion finely tune the congestion window and retransmission timeouts while the DCTCP portion relies on ECN marked packets to prevent the queue build-up in the intermediary switches. OTCP and ODCTP experiments use the parameters shown in Table 3.3.

The different variants of TCP are compared with respect to the FCT of flows experiencing

incast collapse. A single host is set as the aggregator and the remaining nodes initiate a short-lived burst of traffic to the aggregator simulating a partition-aggregate scenario. In this scenario, 1 host is used as the aggregator and the remaining 9 hosts as the workers leading to a 9:1 oversubscription. For each round of the experiment, the worker nodes each initiate 10 flows to the aggregator with a size varying from 1 to 100 MSS-sized packets, totalling to 90 mice flows received at the aggregator. For each congestion control algorithm and flow size the experiment has been run 10 times for consistency.

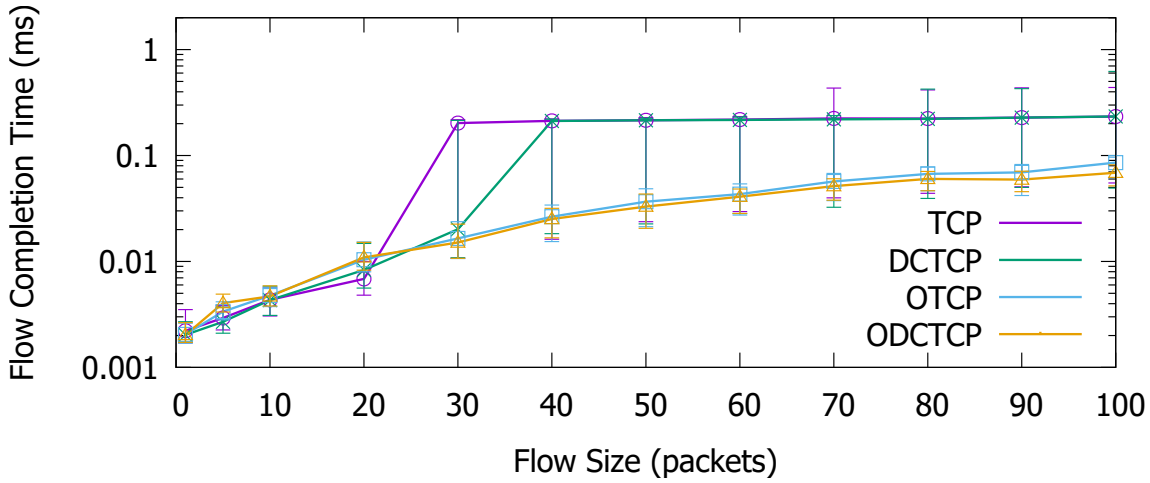


Figure 3.12: Mean FCT

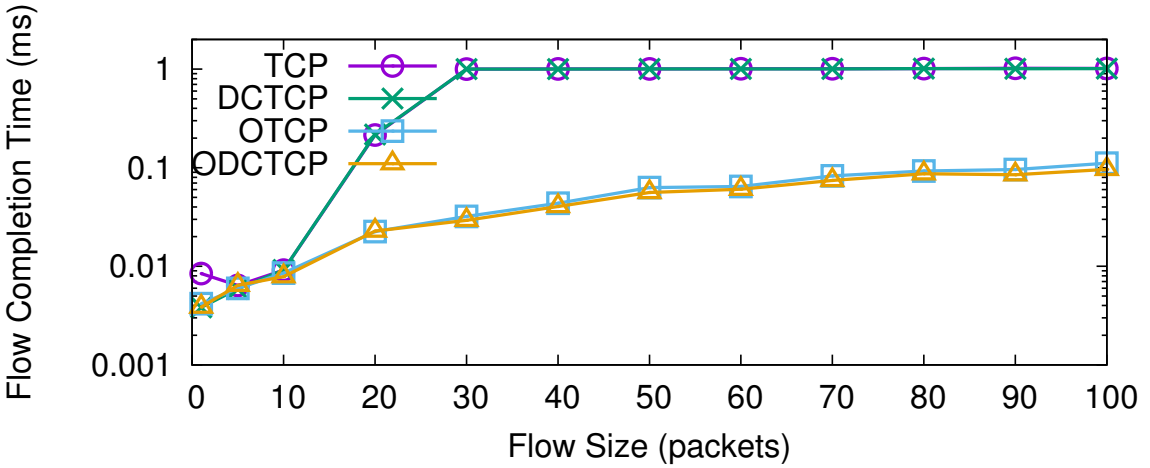


Figure 3.13: 95th percentile FCT

In Figure 3.12 it can be observed that the mean FCT for both TCP and DCTCP is dominated by RTOmin when the flow size is 20 and 40 packets respectively. This is explained by IW being too large, quickly filling the buffers dropping more packets than required for F-RTO

to be triggered and therefore relying on RTOmin to retransmit lost packets. DCTCP does not solve the problem of incast collapse on short-lived flows, because DCTCP relies on ECN markings to adapt the sending rate. This approach requires 1 RTT to be effective due to ECN, once the buffer has already been overflowed by each flow sending their initial window. Therefore it reacts to congestion only after the congestion event has passed. OTCP and subsequently ODCTCP, using finely tuned parameters avoid overflowing the buffers from the initial burst and recover quickly on packet drops, resulting in an improvement of up to 12× in mean FCT.

Figure 3.13 shows the FCT at the 95th percentile over the different runs. Similar pattern can be observed as for the mean with TCP and DCTCP performing similarly and the FCT being dominated by RTOinit instead of RTOmin. In this case, due to the large IW, the SYN packet of some flows is dropped by the switch requiring RTOinit to elapse before the packet is resent. These long delays at the 95th percentile result in long tails in the overall FCT. Using OTCP, the FCT of the long tail flows can be improved by 31× in these experiments. Overall, at the mean and 95th percentile, OTCP performs similarly ensuring fairness amongst flows and no long delays caused by a minority of the flows.

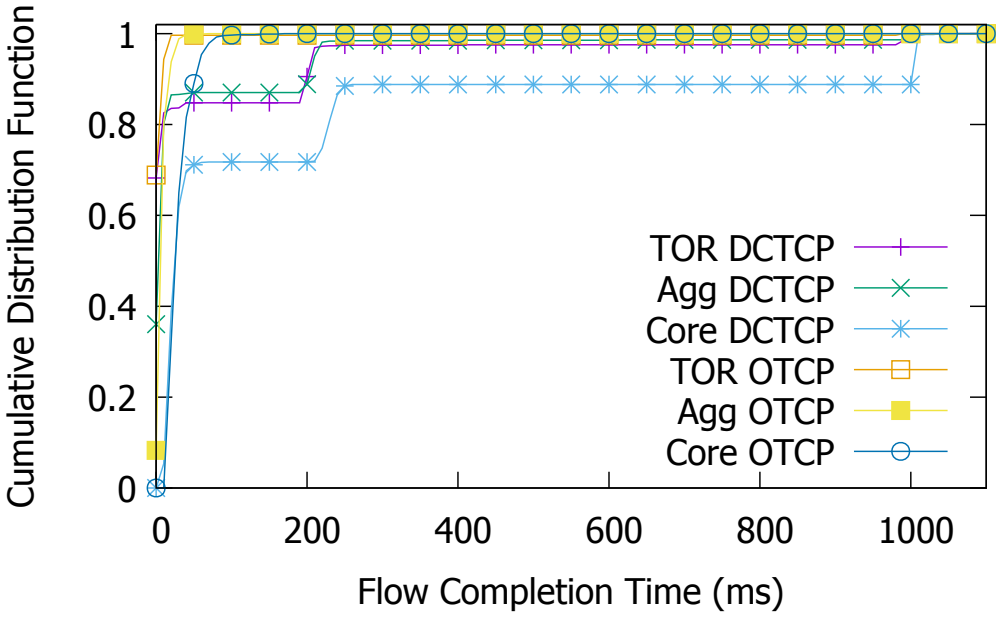


Figure 3.14: CDF of Flow Completion Time (FCT) comparing OTCP with default TCP in a three-layer topology

Finally, in Figure 3.14, the Cumulative Distribution Function (CDF) of the FCT for OTCP and DCTCP across the three layers of the topology is shown. For clarity, TCP and ODCTCP have been omitted from this experiment. In all three layers, OTCP outperforms DCTCP. RTOmin and RTOinit for DCTCP are clearly visible on the FCT. At the three layers, a plateau of 200ms can be observed as well as a long tail at 1s generated by the loss of the initial SYN packet for 9% of the flows. When traffic is confined in the same rack, OTCP is able to complete all of the flows in less than 30ms compared to DCTCP completing 82% of the flow in the same interval, 97% after 250ms, and every flow after 1 second. Through the *Agg* layer, using OTCP, all the flows are transmitted in 40ms while DCTCP completes 86% in the same time and 98% after 220ms. Finally, at the *Core*, OTCP completes transmission after 80ms while DCTCP completes 72% in the same interval and 89% after 250ms.

3.5.4 Goodput

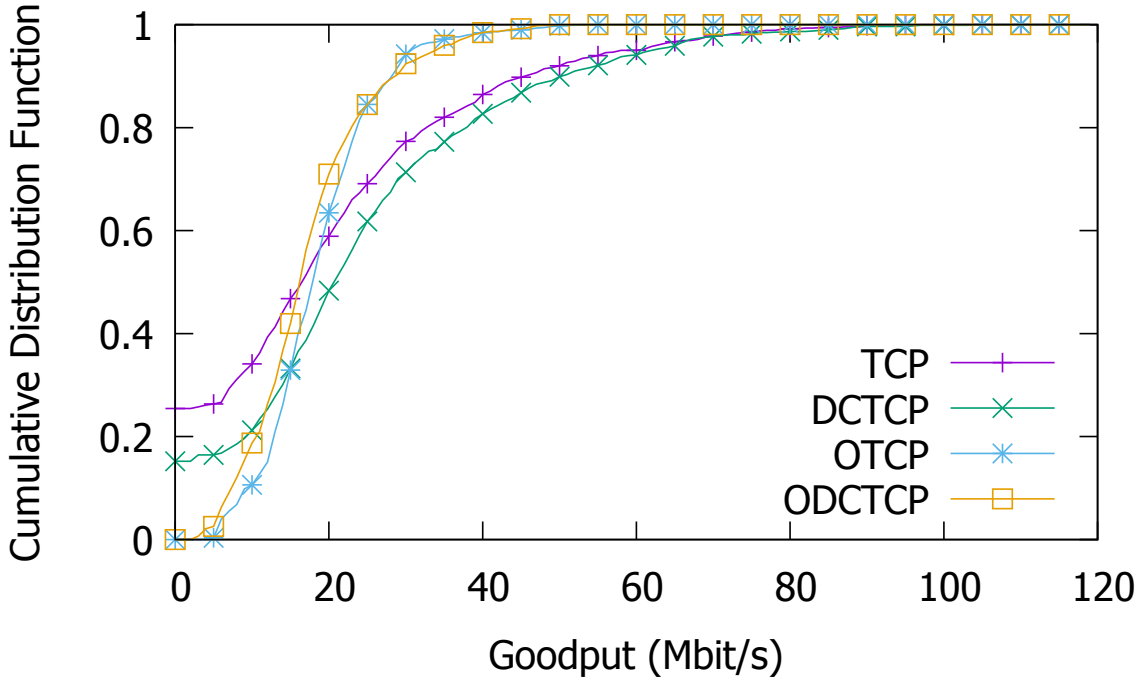


Figure 3.15: CDF of flow goodput experiencing incast collapse.

In order to show that the link is fairly shared amongst the competing flows, this experiment compares the goodput distribution of the different congestion control algorithms. Figure

3.15 highlights, that the goodput distribution for TCP and DCTCP is unfair with few flows reaching extremely high throughput and most flows performing poorly. Consistent with the FCT evaluation for TCP and DCTCP, a large number of flows achieve a goodput of less than 1Mbit/s due to long off periods and few lucky flows are able to achieve high goodput. This long tail in goodput represents the few flows that were able to send the full IW of 10 MSS without retransmissions, and therefore completed their transmission early.

O(DC)TCP does not suffer from this unfairness in goodput, with every flow transmitting from 10 to 30 Mbit/s while for TCP and DCTCP it ranges from lower than 1Mbit/s to 80Mbit/s. This fairness improvement is achieved by reducing the IW and therefore better interleaving the packets when the synchronised flows send their IW and resulting in each flow observing a similar level of network congestion. These observations highlight that in OTCP the bandwidth is fairly divided amongst the competing flows, reducing the impact of incast collapse and improving the end-to-end goodput necessary for partition-aggregate workloads to operate properly.

3.5.5 Goodput with Active Queue Management

To evaluate the impact of the tuned parameters on achievable flow throughput as well as the fairness of link allocation, this experiment compares a tuned TCP stack against Explicit Congestion Notification (ECN) and Random Early Detection (RED). ECN allows notification of network congestion without the requirement for packets to be dropped, ECN-aware routers can set a flag in the packets header to notify the transmitter of impending congestion. RED prevents the queue build-up in the switches and routers by dropping random packets at a frequency based on the buffer occupation and hence resulting in the transmitter to adapt its sending rate. The topology and traffic characteristics are similar to the previous experiments, 100 mice flows from the workers to the aggregator on a 10:1 oversubscribed link with 85kB of buffering that would cause TCP Incast Collapse in an unmodified TCP stack. The RED and ECN queuing disciplines rely on Linux's Traffic Control packet scheduler. Both disciplines use RED as the underlying scheduler, but RED will drop the packet if it is marked,

while ECN will set the ECN flag in the packet header. Both have been configured to start marking when the queue occupancy is 30kB, approximately a third of the maximum queue size, as recommended in the IProute documentation [126]. The marking probability has been set to 1 for the maximum queue size of 85kB. Following the Linux's RED implementation, the chance of marking packets increases linearly from the minimum value to the maximum value where it reaches 1.

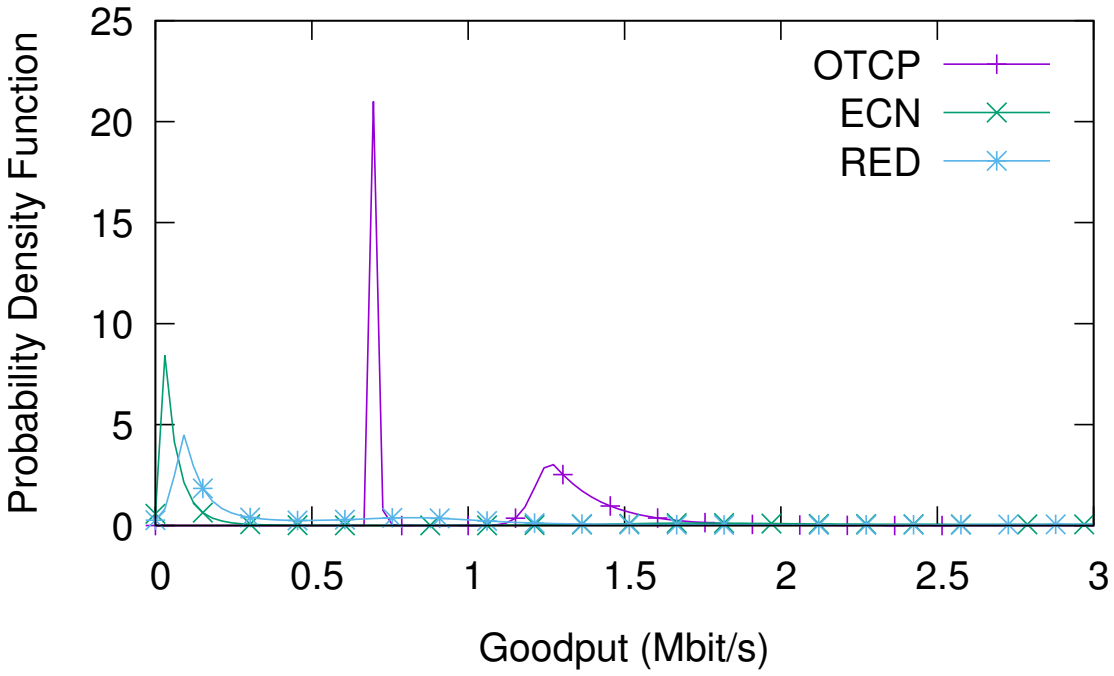


Figure 3.16: The impact of Active Queue Management mechanisms on the goodput of the system

Figure 3.16 shows the probability density function (PDF) of the goodput of individual flows, and highlights the fact that ECN and RED achieve an extremely low throughput under bursts of traffic and are therefore not able to mitigate TCP Incast Collapse. Due to the short-lived nature of the mice flows, ECN notifies the senders of a congestion event one RTT after the event has passed. This delay in congestion notification leads to extremely low and inconsistent throughput shown by most flows only reaching 100KBps goodput and the long tail of a few flows reaching up to 3MBps of goodput. RED behaves better than ECN but dropping packets from only certain flows introduces unfairness in flow goodput with most flows below 250KBps and a uniform distribution up to 1.2MBps. With a TCP stack configured to match the network properties, much higher throughput can be achieved with

fairer allocation of resources. In this experiment, most flows achieve a goodput between 1.2 and 1.7 MBps and flows experiencing congestion achieve 700kBps.

3.5.6 Background traffic

The previous experiments have shown that OTCP can prevent the overflow of the traversed buffers by reducing the initial window to a value close to the BDP as well as recover quickly from packet loss using tuned retransmission timers. In OTCP, long-lived background flows can increase their congestion window up to the clamp value CWNDmax (Table 3.3) that matches the BDP of the path, but when multiple concurrent long-lived flows share the same bottleneck path, the total number of unacknowledged packets can increase above the BDP and therefore create queue build-up. Most flows in DCs are mice flows, but most of the traffic is carried by a minority of background flows that are throughput sensitive instead of delay sensitive called elephant flows. These background flows used for data consistency, migration or replication are long lived and cause queue build-up in the switches.

This section further evaluates the FCT of flows experiencing incast collapse when the traffic traverse the core layer of the network and every layer is heavily utilised by background flows. The experimental setup is the same as before, to create the incast collapse, 10 source hosts generate a burst of 10 mice flows of 20kB. To generate the background traffic at every layer, an additional host is added to each switch running iperf in TCP mode to create a long lasting background TCP flow. A delay is added between starting the background flows and the incast collapse, to allow for the TCP background flows to reach the congestion avoidance mode and fully use the network resources available. Without this delay, the incast collapse event could happen while the background flow is still in slow-start phase and consequently when the queues are not yet fully occupied. Finally, for each congestion control algorithm, the experiment has been repeated 10 times for consistency.

Figure 3.17 shows the mean FCT of the four different congestion control algorithms. For this evaluation both mice and elephant flows are managed by the same congestion control algorithm. At every layer, the background traffic is generated from multiple hosts resulting

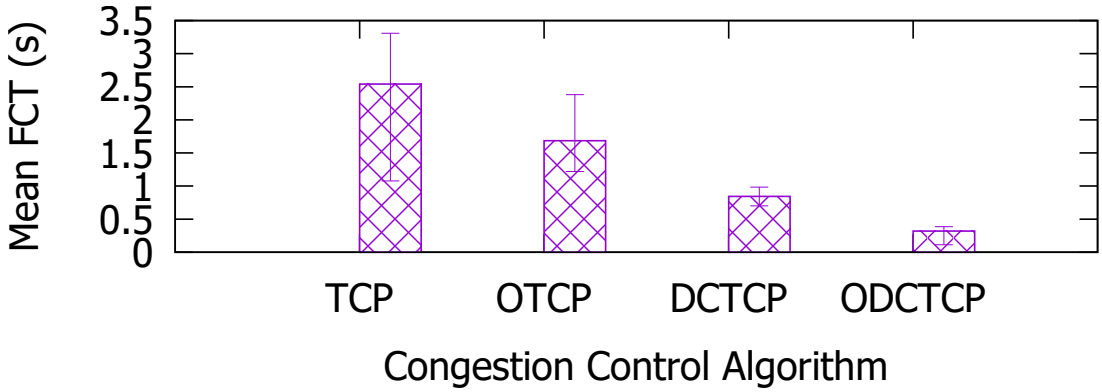


Figure 3.17: Mean FCT of flows under incast with an heavily utilised network.

in queue build-up increasing end-to-end latency and resulting in larger number of packet drops during the incast event as no buffer space is available. In this scenario, TCP is known to perform poorly as it does not prevent queue build-up, while the 10 segments from the IW and long RTO worsen the problem [116]. DCTCP was specifically designed to address the queue build-up from the background flows but still suffers from large IW and long RTO. OTCP suffers from the queue build-up from the background flows due to its reliance on TCP, but reduces incast collapse by only sending few packets in the initial window and recovering quickly from dropped packets. Finally, this work demonstrates that OTCP over DCTCP (ODCTCP) complement each other: OTCP allows finely-tuned parameters to be used adjusting the initial window to match the BDP of the network and the retransmission timers to match the latency of the network, while DCTCP prevents queue build-up at the switches improving mean FCT by 8×.

3.6 Summary

In this chapter, the research presented experimentally evaluates the impact of TCP throughput incast collapse and associates the throughput degradation with the static and inadequate congestion control parameters. These parameters are orders of magnitude higher or lower than what is required for TCP to operate efficiently in the low-latency and high-thoughput environment of Data Centre networks. This work introduces Omniscient TCP (OTCP), an

SDN-based scheme to fine-tune TCP based on the operating environment of a DC. Through the centrally available topological and operational information measured and collected by a central SDN controller, route-specific congestion control parameters can be calculated.

The experimental results show that by fine-tuning the TCP parameters to the operating environment, OTCP can significantly outperform TCP for short-lived, soft-realtime flows, with a $12\times$ improvement in flow completion time at the mean and $31\times$ at the 95^{th} percentile. Additionally, OTCP provides a lower and stable end-to-end latency, as well as higher mean and fairer goodput. Furthermore, the work demonstrates that the proposed approach can be used to improve other variants of TCP such as DCTCP by preventing large queue build-up and TCP thoughput incast collapse under traffic bursts. This is showing that fine-tuned parameters based on the operating environment, in this case for the TCP congestion control, can significantly improve resource utilisation and hence application performance.

Chapter 4

Data Plane Programmability for Software Defined Networks

4.1 Limitations of today's SDN

The previous chapter on OTCP, demonstrated that significant benefits in resource utilisation and performance can be obtained in DC environments when network-wide informed decisions are made. The objective has been to improve the network utilisation of a DC environment with a partition-aggregate traffic pattern. These improvements have been achieved by adapting the network stack in the end-hosts to the environment in which the servers are operating. Through these modifications, the TCP networking stack can leverage known charasteristics instead of operating with conservative values that result in degraded performance. By tuning the congestion windows and retransmission timers to the network characteristics, TCP throughput incast collapse can be mitigated, resulting in higher goodput between hosts and lower latency.

To perform the measurements and collect the necessary metrics to tune the TCP stack, OTCP relies on OpenFlow 1.3 which provides significant insight into the network. But OpenFlow is still too limiting to fully manage and orchestrate the infrastructure dynamically. In OTCP, the latency measurements are convoluted due to OpenFlow limitations, requiring multiple

rounds of measurements to calculate the switch-to-switch and host-to-switch latencies. During these measurements the latency measured includes both the production and management network as well as the OpenFlow encapsulation and decapsulation. Additionally, the performance of the controller software, the overhead of the operating system and the varying resources utilisation can degrade the accuracy of the measurements. Through the approach presented in this work, most of these parameters can be managed, and the calculated values are close enough to the baseline to provide significant benefits. However, each independent environment might require bespoke fine tuning to account for the switch and controller implementations. Additionally, the latency measurements must be performed when the network is idle in order to measure the latency when all the queues are empty. Performing this measurement, even though infrequent, as it is required only during topological changes, is impractical, as operators have little incentive to shutdown the infrastructure for any period of time. To perform the latency measurements while the production traffic is still present would require the measurement traffic to have priority over production traffic necessitating custom queues for specific traffic patterns.

An important parameter in the operation of OTCP is the value of α that represents the number of flows that will concurrently share the same path. Using α, the BDP of the paths between two servers can be shared fairly amongst competing flows resulting in low jitter, and hence, a highly predictable flow completion time. Setting α to a fixed value as in OTCP might be representative of the operating environment at a particular point in time, but with the rapid deployment of applications and services the number of concurrent flows can change rapidly. Furthermore, the number of established flows between servers might not be fully representative of the current network state in the case of active but non-transmitting flows. To adapt α to the changing network characteristics, it is necessary to account for the number of active and transmitting flows in the network which is impossible using OpenFlow 1.3, and therefore OTCP must rely on a static value based on the expected traffic pattern. OpenFlow 1.5, introduces the concept of TCP flag matching, which allows flow rules to match on particular flags of the TCP header. Using this approach, it would be possible to send a notification to the controller on TCP SYN and FIN packets, allowing the controller to keep

an accurate account of active flows. Nonetheless, even though OpenFlows 1.5 adds support for TCP flag matching, it requires each flow establishment and termination packets to be sent to the controller. This requirement makes it impractical in environments of the scale of DCs, with a very large number of short lived flows, as the controller would have to deal with a multiple requests for every single flow, quickly overwhelming it.

As well as the number of active flows, other network metrics are important in order to better tune the TCP stack for a specific environment. The calculation in OTCP is based on the MSS, which makes sense when the Nagle algorithm [127] is enabled to aggregate packets into MSS-sized frames. However, in some scenarios, especially with realtime traffic, the dominant packet size in the infrastructure might be less than a MSS, and therefore the parameters must be modified accordingly. Another factor for the MSS is the variation of traffic pattern between peak hours and off hours: during peak-hours, the traffic will be dominated by the production apps, while during off-hours the traffic might be dominated by background tasks, such as replication or backups. Furthermore, in OTCP it is assumed that the traffic is mostly TCP, which is realistic for the majority of DCs, but in specific operating environments the volume of UDP traffic might be much more prevalent to provide VoIP services or VPN tunnels. Since UDP does not include any congestion control mechanism, it is unfair towards TCP, and a high volume of UDP traffic can significantly impact the performance of TCP. In an environment with a TCP stack operating at a very low queue occupancy, UDP might worsen the performance of TCP even further by greedily filling all the switches' queues. Public DC providers such as Amazon and Google throttle by default UDP traffic by orders of magnitude lower than the link-rate [82]. This segregation of traffic requires some level of management and control of the queuing discipline and the queues themselves, which is very limited in OpenFlow.

Another critical aspect of OTCP is to tune TCP based on the BDP of the path to reduce the queue occupancy. In order to demonstrate the impact of buffers on latency, throughput and jitter, the experiments described in this work have been performed on a NetFPGA platform that provides the programmability and interface necessary to measure and collect the cur-

rent buffers' state. These experiments have been conducted on NetFPGAs as it is one of the few platforms to openly provide the means to perform such measurements. OpenFlow includes very few functions to interact with the queues and buffers, and vendors proprietary SDKs (e.g., Broadcom) are next to impossible to obtain. This limitation highlights that even though OpenFlow has enabled programmability of the routing and forwarding, the data plane and internal state of the devices still remains a blackbox. Furthermore, the BDP is calculated using the end-to-end bandwidth between a pair of servers, requiring an accurate measurement of the end-to-end maximum bandwidth. Even though measuring the available bandwidth seems straightforward, realistically it is challenging: the statistics in the devices might report the maximum throughput observed so far, but this might not represent the maximum capacity, especially on underutilised links. The approach taken by OTCP is to use the reported port status by OpenFlow, but the port speed and the actual link speed might be different due to overlay networks, intermediary switches, or middleboxes. The port status is also influenced in virtualised environments where the speed is not physically limited. OvS, the leading software switch implementation, always reports a 10Gbps port speed regardless of the achievable speed, which might be orders of magnitude lower or higher than the measured maximum throughput in a virtualised network.

Although the results presented in the previous chapter clearly highlight the benefits of environment-specific tuning, due to the inherent limitations of today's networking technologies, the proposed approach is quite involved, requiring custom modifications for individual deployments. These limitations to the proposed design are due to the lack of flexibility and programmability in current networks, in general, even with SDN as it currently stands. Numerous steps must be performed in order to calculate the latency of a link, as OpenFlow does not include any built-in mechanism to measure latency or the means to report timestamps to the controller. To calculate an adequate initial window in a changing environment, the number of active flows must be accounted for, which is impossible in the mainstream version of OpenFlow, and impractical with the latest revision. However, in a programmable environment, flow accounting should be delegated, if necessary, to the network device, simply reporting at regular intervals the number of active flows. Using the reported number of

active flows, the controller should be able to calculate fine-tuned congestion control parameters for the current traffic characteristics. Additionally, network devices should also provide the ability to collect and report additional metrics, such as the average packet size, the packet size distribution and the maximum throughput between devices.

4.2 Evolution of OpenFlow

Table 4.1: Number of fields supported by OpenFlow for each protocol revision and associated storage required for individual flow entries.

OF Version	Release date	Match fields	Depth	Size (bits)
<1.0	Mar 2008	10	10	248
1.0	Dec 2009	12	12	264
1.1	Feb 2011	15	15	320
1.2	Dec 2011	36	9–18	603
1.3	Jun 2012	40	9–22	701
1.4	Oct 2013	41	9–23	709
1.5	Dec 2014	44	10–26	773

In order to deal with its biggest limitations, OpenFlow has significantly evolved between its first release in 2008 and the current 1.5 specification released in December 2014. It evolved from a single match table with 10 match fields to multiple stage table matching and ingress and egress pipeline separation with 44 different match fields. The number of match fields have been continuously updated to add matching capability to new header fields, such as IPv6 addresses, MPLS, and VLAN headers. Table 4.1 shows each major revision of the OpenFlow protocol alongside the number of supported fields, the maximum number of fields a single flow can match on (referred to as "depth"), and the maximum number of bits a single flow entry requires. Figure 4.1 shows the graph of all mandatory match fields between L1 and L4 defined in OpenFlow 1.3 specification and necessary for any OpenFlow compliant switch implementation. The large increase in size between versions 1.1 and 1.2 is related to the addition of IPv6 support. However, expanding OpenFlow to support more fields requires TCAM size in switches to grow accordingly, or otherwise incur a significant reduction in the number of flows installed at the switch at any given time. On a standard OpenFlow Top-of-Rack switch, such as a Pronto 3290 (FireBolt 3 switching ASIC) running OpenFlow 1.0,

around 2000 flows can be simultaneously inserted [128]. If the same switch was to support OpenFlow 1.5 and following the growth shown in Table 4.1, only approximately 700 flow entries could be accommodated. The growth over time of supported match fields is unlikely to slow down as network requirements continuously evolve and support for upcoming and operator-specific protocols is required. This constant evolution of the OpenFlow specifications to cope with the demand for new match fields highlights two of its big limitations: flexibility and future-proofness.

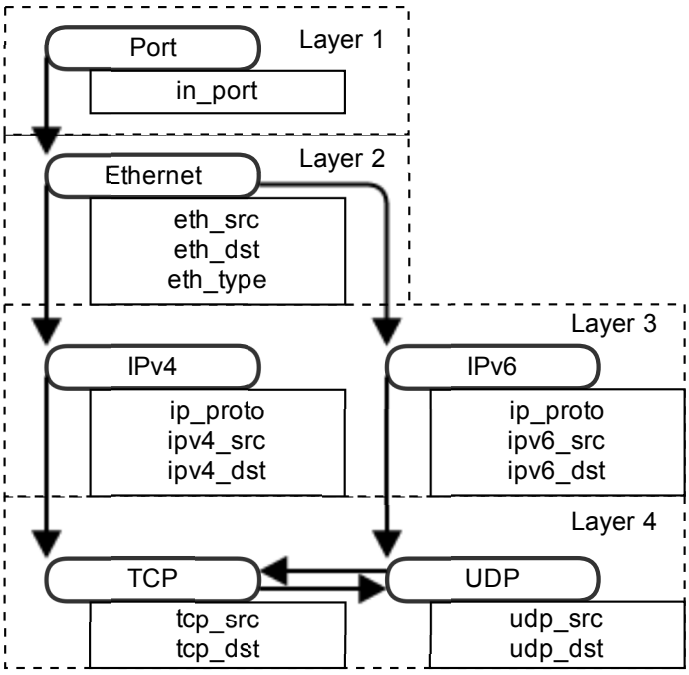


Figure 4.1: Mandatory match fields required by OpenFlow 1.3 represented as a tree. A depth of 9 for L1-L4 matching can be seen, in relation to the depth column of table 4.1.

Flexibility in OpenFlow is not only limited by the fixed set of match fields but also by the very limited set of actions, monitoring, metering and matching capabilities that can be performed on every packet. In OpenFlow, matching can be performed with or without a bit mask depending on the particular field. This matching approach is directly related to the hardware available in today's devices. Using the CAM and TCAM memory, the packet header fields can be matched against the associated flow entry with a single table lookup. However as a consequence, matching can only express strict or partial equality and is unable to reflect more complex algebraic operations such as "less than", "more than or equal to" or "not equal to". This limitation is akin to a common hash table, it is easy to identify if an entry is present

in the table, but finding the closest value in the table requires traversing all the entries. This limitation can quickly render seemingly simple applications impractical or infeasible. Monitoring and metering in OpenFlow is also limited to a fixed set of metrics defined in the specifications, namely flow duration, and packet and byte count, preventing OpenFlow to be used for large-scale fine-grained monitoring. As of OpenFlow 1.5, the statistics capabilities have been extended to partially address these issues, and the protocol allows other statistics to be defined but requires the switch vendor to provide the implementation resulting in vendor lock-in and poor flexibility for the operators.

In order for the network fabric to evolve in conjunction with the rest of the infrastructure and provide the management and monitoring required, this work argues for a platform, protocol and language-independent instruction as a replacement for the static packet processing architecture currently available. Using such an instruction set to express the per-switch behaviour, the packet processing and forwarding can be configured dynamically, alleviating the limited matching, protocols support, and statistics in OpenFlow. To orchestrate the large number of networking devices and allow for runtime reconfiguration, a central controller maintains connection with the devices through an open control plane API providing control and insight over the network fabric. Through this interface, the controller is able to describe specifically the requirements for the switch to operate and the specific processing required resulting in a forwarding decision. As a result, the network fabric functionality can be extended and reconfigured at runtime to support the evolving demand of the infrastructure providing the flexible routing, processing, forwarding and monitoring that is now expected of an SDN environment. Using the instruction set, lightweight functions can be implemented on the devices across the fabric. Functions, such as e,g, arbitrary statistic gathering and reporting, packet tracing, network telemetry, load balancing and anomaly detection, are necessary for the better management and improved resource usage of the infrastructure.

With the large variety of possible switches that are currently deployed, from traditional high-density switches to programmable Network Processors (NPU) to Field Programmable Gate Array (FPGA), and with the recent emergence of software switches in virtual networks, it

is necessary for the proposed design to be target-independent. Target independence allows the same data plane function to be executed across different devices, simplifying orchestration, maintenance and development as well as preventing vendor lock-in. To enable future-proofness, the design should be protocol independent, supporting existing and future protocols by programatically expressing the packet parse graph without constraints on which packet headers can be matched. Finally, expressing the data and control plane behaviour should be language independent to allow any High Level Language (HLL) to express behaviour following a particular underlying data plane instruction set and control plane APIs.

This chapter presents BPFabric, a framework designed to address the aforementioned limitations of SDN by providing a platform, protocol and language-independent architecture for data plane processing and centralised orchestration. This chapter is structured as follows. In section 4.3, the entire design of the proposed architecture is presented in a top-to-bottom approach. In section 4.4, the details of the two software switch implementations and the controller are provided. In section 4.5, BPFabric is demonstrated through the implementation of forwarding, telemetry and anomaly detection data plane functions. Section 4.6 demonstrates the performances of the proposed system and section 4.7 concludes the chapter.

4.3 System Design

This section, takes a top-to-bottom approach to explain the design of BPFabric, the proposed novel SDN architecture, from the user-level definition of the network program, to the compilation, installation and execution of the program in the dataplane, and the orchestration of such infrastructure through the control plane.

4.3.1 Architecture Overview

Figure 4.2 presents an overview of the network infrastructure in an example BPFabric deployment. In this environment, the network operators have implemented four data plane functions that can be deployed to the switches, such as an Intrusion Detection System (IDS),

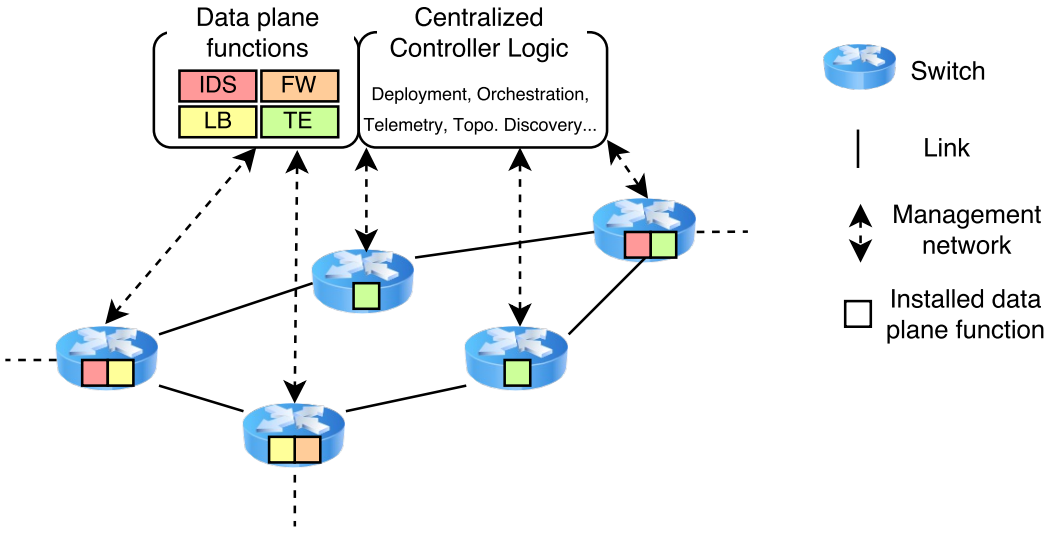


Figure 4.2: Overview of BPFabric management and data plane function deployment over a network infrastructure.

a Firewall (FW), a Load Balancer (LB), and a Telemetry module (TE). Using the BPFabric southbound API between the controller and the switches, the centralised controller is able to dynamically deploy and replace the data plane functions across the fabric and monitor the internal state of the switches. Through this approach, the controller can deploy operator-defined functions, allowing existing routing and forwarding functions to be deployed in addition to more complex middlebox-like functions that have so far been delegated to highly specialised and costly hardware. The management and orchestration of such infrastructure can be decomposed to the following steps:

- **Data plane behaviour:** Using a High Level Language (HLL), the per-switch data plane behaviour is defined, dictating the parsing, matching, and forwarding decision that is performed on every packet.

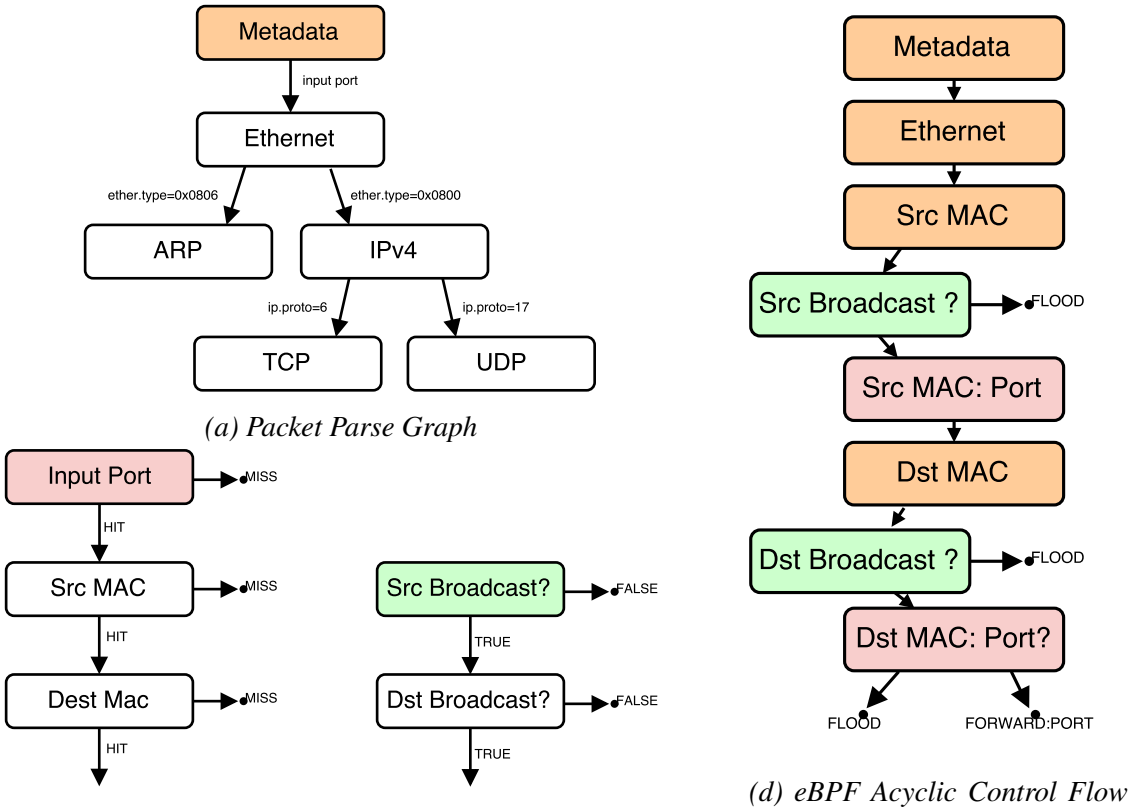
- **Behaviour compilation:** The HLL definition is compiled to a platform and protocol independent instruction set, and encapsulated into an Executable and Linkable Format (ELF) binary with the associated metadata.

- **Controller Install Request:** Using the southbound interface, the controller sends an install request to the switch containing the binary.

- **Agent ELF Loader:** The agent running on the switch receives the binary, processes it into a suitable format for the switch's platform, and allocates the necessary tables.
- **Receiving Packet:** Every packet received at the switch will be passed to the execution engine, querying and updating tables, raising notifications and finally making a forwarding decision.
- **Forwarding decision:** Based on the execution engine return value, the packet is forwarded to the relevant port, sent to the controller, dropped or flooded to all other ports.
- **Controller operation:** On event notification or forwarding to the controller, the latter is delegated the responsibility to decide the action to perform.

In OpenFlow, the pipeline with the multiple match-action tables is fixed by the switch vendor and therefore so is the data plane. However, using the southbound API provided by the OpenFlow protocol, the data plane behaviour can be modified by the controller by adding and removing entries in the tables as long as it complies to the device's pipeline. In BPFabric, the data plane provides the execution engine to run any pipeline defined by the network operator using the protocol and platform independent instruction set. Consequently, the data plane does nothing until its behaviour is defined, including parsing packets, updating meters and statistics, looking and updating tables, or forwarding packets to the controller.

The controller is logically centralised, but can be physically distributed across multiple nodes to cope with a large number of devices, more requests per second or provide shorter RTT. BPFabric allows most devices to keep the management of their own state and therefore removes a lot of burden in keeping a consistent global state when distributing the controller across multiple nodes as it is the case in OpenFlow. Each switch connects to a single controller with fallback to another controller if the connection cannot be established. The controllers can then handle only a portion of the network devices and share state globally, if necessary, using a distributed data store such as ZooKeeper or Redis. If multiple controllers are used, it is the responsibility of the controller applications to use the distributed data store to achieve coordination between controllers.



(a) Packet Parse Graph

(b) Table Flow Graph

(c) Conditional Flow Graph

(d) eBPF Acyclic Control Flow Graph

Figure 4.3: L2/L3 eBPF Acyclic Control Flow Graph decomposed into the underlying packet parse graph, table flow graph and conditional flow graph.

4.3.2 Dataplane behaviour definition

In BPFabric, the first step is to define the behaviour of each individual switch within the infrastructure and what actions should be performed once a packet is received on a port. This program being executed for every packet in the data plane must be fast to offer line-rate performance and provide guarantees on the execution time. This behaviour being executed on every switch it is installed upon does not provide the global view of the network. However, following the SDN paradigm, the program can delegate decision-making to the central controller if it is unable to make a local decision on what action should be performed on a packet.

Data Plane Instruction Set

In order for programs to be seamlessly deployed across heterogeneous hardware devices and software switches, it is necessary for the compiled programs to be platform independent. In 1992, McCane and Jacobson defined the Berkley Packet Filter (BPF)[1][129], a widely-used instruction set specifically designed for packet filtering within the kernel. cBPF has been widely deployed in a large number of packet processing libraries such as libpcap, used by programs like tcpdump or wireshark. Even though cBPF is over 20 years old, it is still widely used and recent improvements with extended BPF (eBPF) have been integrated in the mainstream Linux kernel (3.18+). Some of the most notable differences between cBPF and eBPF is the move from 32 bits to 64 bits, an increase from 2 to 16 registers, the support for table operations and function calls, as well as its extension to more of the Linux kernel than just the networking stack [130].

An often forgotten fact is that cBPF was defined in 1992 as a "pseudo-machine", nowadays called virtual-machine, with the main goal to provide protocol and platform independence through a simple but general instruction set designed for fast interpretation and execution. To allow BPF programs to be executed at a low cost, the instruction set was defined to closely match the instructions of a register machine making interpretation and translation such as just-in-time (JIT) compilation, fast and straightforward. BPF does not have any knowledge of the different network protocols and packet structure, and parsing of each packet is performed by loading some of the packet header fields into registers and performing the necessary comparisons. As a result of this simple load and compare approach, the BPF program can be decomposed into a parse graph expressing how the header fields should be extracted (Figure 4.3a), a table flow graph expressing the match table sequence (Figure 4.3b), and a conditional flow graph expressing the algebraic boolean comparisons (Figure 4.3c).

One of the main design criteria of cBPF has been to prevent backward jumps in the execution and to prevent loops that could lead to a non-deterministic execution-time unsuitable for real-time processing. eBPF has relaxed this requirement allowing bound-loops, there-

[1]BPF has been retrospectively called cBPF to differentiate it from eBPF. For the remaining of this thesis BPF will be used to reference both cBPF and eBPF.

fore depending on the timing requirements, a verifier can be used to statically analyse the program and provide an upper-bound on execution time (critical-path). As a consequence of this forward-only execution, the eBPF program can be synthesized into an Acyclic Control Flow Graph (aCFG), shown in Figure 4.3d, by combining the underlying parse, table, and conditional graphs.

A major addition in eBPF is the ability for the virtual-machine to expose a set of functions to the eBPF program and hence to allow complex and blackbox features to be used. An example of such blackbox feature is the checksum calculation that will often be provided through dedicated hardware. In the Linux kernel implementation of eBPF, the most notable functions that have been exposed are to insert, lookup, and delete entries in pre-allocated tables. Multiple types of tables including arrays and hashmaps can be created and assigned to an eBPF program (more details in section 4.3.2); once created, the eBPF program can query and update the tables to maintain state of the data plane function. In traditional switches, tables are commonly used to keep track of state and to allow for match-action type execution, for instance in a simple layer 2 learning switch, a lookup table (e.g., hashtable) will be used to store the port associated to a specific MAC address. The most common types of tables are lookup-tables, usually implemented in hardware with content-addressable memory (CAM), wildcard lookup-tables allowing partial matching usually using ternary content-addressable memory (TCAM), and Longest-Prefix Match tables (LPM) commonly used for IP routing implemented using CAM or high speed memory with a Trie data structure. Using eBPF the specialised tables of hardware devices can be abstracted away, providing the high performance without increasing the complexity of the data plane function implementation.

In the proposed architecture, platform and protocol independence are critical factors to allow a single program to define the behaviour of the network fabric. Therefore this work suggests to use eBPF as the intruction set for the compiled data plane functions as it provides protocol and platform independence, memory safety, real-time processing guarantees and table operations. Futhermore, eBPF is a known, understood and well-documented instruction set specifically designed for packet processing and can be simply executed on a wide range of

targets. Although eBPF is suggested as the instruction set, as it is likely the best candidate, any instruction set providing similar features could be envisaged in the architecture proposed.

High Level Language Definition

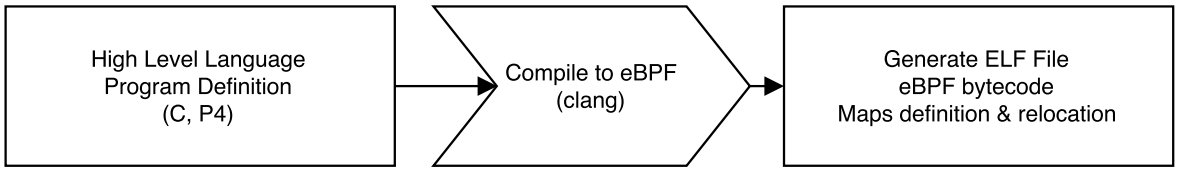


Figure 4.4: Program definition, from high-level language to ELF encapsulated bytecode and metadata

eBPF provides both the platform and protocol independence required, but it is an instruction set modeled around a small set of instructions and registers. Similar to any instruction set, such as the widely used x86_64, programming directly with the instruction set or its closest representation assembly is a complex task, requiring long development effort, being prone to errors and potentially less efficient than compiler-generated code.

High-level programming languages (HLL) should therefore be used to define the behaviour of the data plane following the usual development steps, as shown in Figure 4.4. Currently, the most common programming language for eBPF is a restricted subset of C, due to its very close ties to the Linux kernel. As of clang 3.7, the eBPF backend was added to the compiler to allow straightforward compilation from purpose-made C source code to an eBPF binary file. An application-specific HLL can also be envisaged to better match the behaviour of the data plane such as P4 [18], a HLL specifically designed for protocol-independent packet processors. Existing efforts within the IOvisor open source project have resulted in a partial P4-to-ebpf compiler [131].

ELF

Most eBPF programs will contain one or multiple tables for packet matching and to keep track of states during the execution of the data plane function. The simplest approach is to

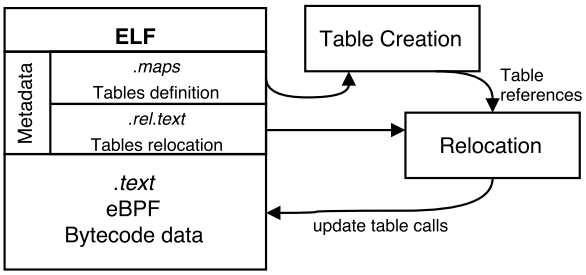


Figure 4.5: eBPF table relocation from the ELF binary

statically allocate the memory required for the tables on the stack and use software implementation of the tables. However, this approach does not satisfy the platform independence requirement and might under-utilise some of the resources available on the device. A traditional hardware switch contains hardware tables that can perform queries and updates at line-rate, and often have a relatively small amount of general purpose memory. Therefore, to truly achieve platform independence, the tables should be allocated by the device itself based on the available resources, and the eBPF program should be updated to relocate the table operations to the newly allocated tables. Using this approach, each device's resources can be fully utilised by allocating hardware and software tables as necessary.

A program compiled into eBPF will contain the compiled eBPF bytecode to be executed for each packet, but will also require some additional metadata for each table specifying the table unique identifier, type, size of keys, size of values and maximum number of entries. In order to carry both the executable binary and the necessary metadata, the architecture proposed relies on the Executable and Linkable Format (ELF), a platform and system independent file format widely used in Unix systems as the standard binary file format. ELF is a standard file format, design to encapsulate programs and associated data in a well-defined structure that is independent of the instruction set. Additionally to the program itself, the ELF contains a wide range of information necessary for the program to execute such as the functions that will be linked to it during execution, the memory location of the strings, the debug symbols and in the case of eBPF the tables to allocate. The resulting ELF file allows true platform independence, it defines which tables should be created, and how the created tables should be linked to the eBPF program and the eBPF instructions. A simple representation of the relocation process from the ELF file into the eBPF bytecode is shown in Figure 4.5.

4.3.3 Switch architecture

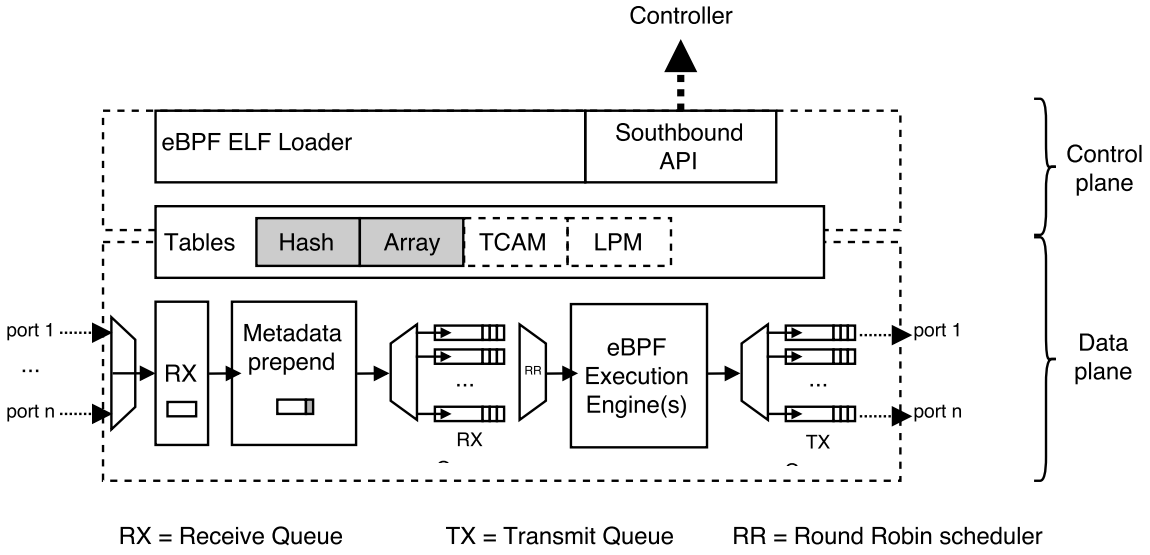


Figure 4.6: A simplified diagram of the switch architecture with the separation of logic between the control plane hosting the agent and the dataplane processing the packets arriving at the ingress.

Figure 4.6 provides an architectural overview of the control and data planes within the switch. Following the SDN approach made popular by OpenFlow, the device is designed to be "dumb", or more specifically, it does not contain any internal logic (e.g., self-learning, peering, discovery, etc.). As a result, the control plane can be kept lightweight, hosting only an agent akin to the OpenFlow agent, responsible for interfacing the underlying data plane with the central controller through the southbound API. Using this approach, the behaviour of a switch is only defined by the explicitly installed eBPF program or through requests issued by the controller.

One of the critical components within the control plane is the eBPF ELF Loader that is responsible for instantiating the eBPF maps required by the data plane function, relocating the call to the maps accordingly and finally transforming the eBPF bytecode into a fast and usable format suitable for the device. The eBPF loader is device-specific and therefore should perform the operations that are the most suitable for the target device. The simplest approach is to keep the bytecode as is and use a software interpreter as the eBPF execution engine, but this approach is slow and unsuitable for high throughput, low-latency packet processing. Just-in-Time compilation can be used to transform the bytecode into a platform-

native instruction set, providing near-native performance. More complex approaches can also be considered, such as translating the bytecode to a specific Network Processing Unit (NPU), synthesizing the control flow graph into an FPGA core or into a match-action pipeline such as RMT [16], or running natively eBPF instruction on a dedicated ASIC. With the very high speed achievable with programmable match-action pipelines such as RMT, being able to translate eBPF data plane functions would allow a high level of programmability for backbone networks with very large aggregate throughput.

An optional stage in the control plane is the verifier that checks the validity, security and performance of the eBPF program being loaded. The verifier can be used to statically and deterministically verify if a program terminates correctly, the memory accesses are bound to the address space allocated, loops are not present, and the maximum depth of the execution path to calculate the worst-case execution time of the program.

Data Plane

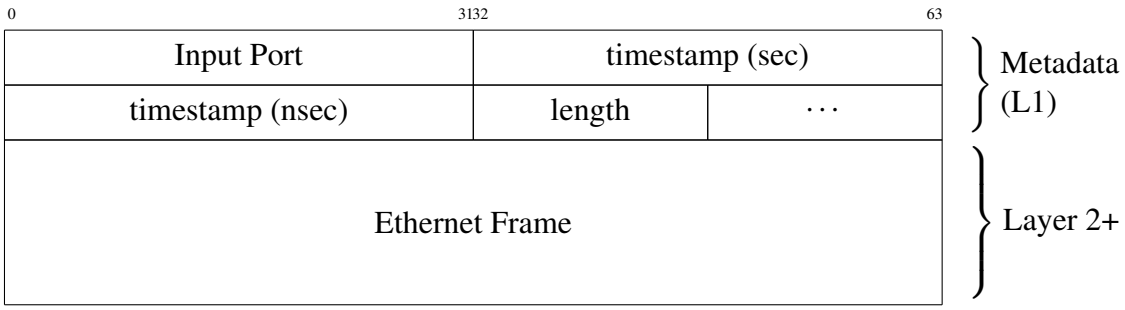


Figure 4.7: Packet format within the dataplane. Metadata information is appended to the frame received.

The data plane receives packets from the input interfaces and stores them into receive queues with some prepended metadata providing link layer information and a receive timestamp. Figure 4.7 shows the structure stored in the receive queues with the metadata prepended before the ethernet frame. The reason for prepending the metadata to the packet is to be able to query and compare against some of the fields in the eBPF execution engine. Originally BPF was designed as a per-socket program and therefore link-layer information were out of scope. With its evolution over time and the requirement for metadata, the Linux kernel has

used specific memory addresses to represent the metadata, but this approach is a workaround and in a switch where the metadata will always be required, prepending the information to the frame itself is the most suitable approach. Using this approach, the eBPF program can load the metadata information as any other header field, for example, the 32 bit value at address 0 can be loaded and compared to make a forwarding decision based on the input port of the packet.

Subsequently, once one of the eBPF execution engines is available, a packet and its associated metadata is retrieved from the receive queues. The order in which the packets are retrieved from the input queues is dependent on the packet scheduling algorithm which is out-of-scope in this work (the existing implementations are using round-robin). The packet retrieved from the queues is passed to the available eBPF execution engine to establish a forwarding decision for this particular packet. At this point, the program that was previously loaded by the agent through the eBPF loader is executed.

Name	Value	Description
FLOOD	0xFFFFFFFD	Send the packet on all the ports except the input port.
CONTROLLER	0xFFFFFFFE	Send the packet to the control plane for it to be encapsulated in a *packet_in* message and sent to the controller.
DROP	0xFFFFFFFF	The packet is dropped.

Table 4.2: eBPF return values used for special actions.

The executed program performs the load and jump operations as well as the table lookups necessary to make a forwarding decision. The forwarding decision is based on the exit value of the eBPF code returned by the execution engine. An exit value below the hexadecimal value of 0xfffffffd is considered a forward to the port specified by the returned value which can be either a physical or virtual port (e.g., a tunnel). Otherwise, the value is considered a special action to flood, drop or send the packet to the controller as defined in Table 4.2. During execution, the eBPF program can raise a notification to the control plane with a specific identifier and payload used to notify that a user-defined event has occurred.

Once a forwarding decision is made, the packet is dropped or enqueued on the relevant transmit queue(s). In either the receive or transmit scenario, a tail-drop mechanism is used

in the case of a fully occupied buffer, but this is implementation independent and, similar to the packet scheduler, any suitable mechanism can be used.

4.3.4 Control Southbound API

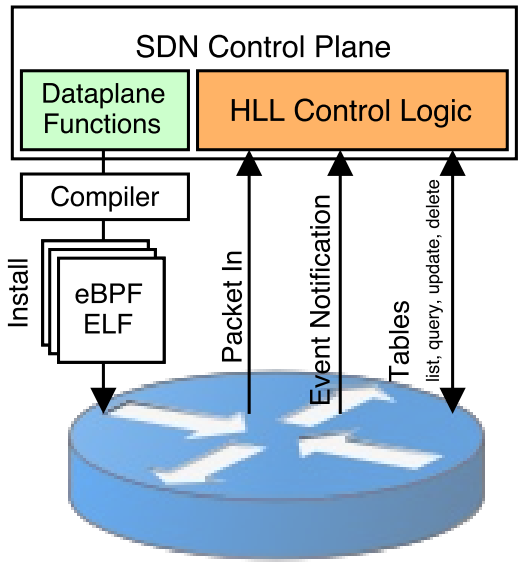


Figure 4.8: BPFabric controller-to-switch communication

Similar to OpenFlow, a controller provides an overarching view and control over the network by maintaining a persistent connection with the controlled switches as shown in Figure 4.8. In SDN terminology, this communication interface between the controller and the devices has been referred to as the Southbound API with the most well-known implementation being OpenFlow, and other alternatives being the Network Configuration Protocol (NETCONF) or Cisco OpFlex. In BPFabric, both in-band and out-of-band control planes are supported, and whether to use one or the other is depending on the infrastructure available, and the reliability and security levels required.

In the proposed architecture, this persistent connection between the controller and the switches is used by the controller to issue synchronous requests to the switches, and for the switch to raise events and notifications when the controller must be involved. The first step of the communication is to set up a handshake between the controller and the device to establish the version compatibility of the endpoints, negotiate the supported features and the datapath

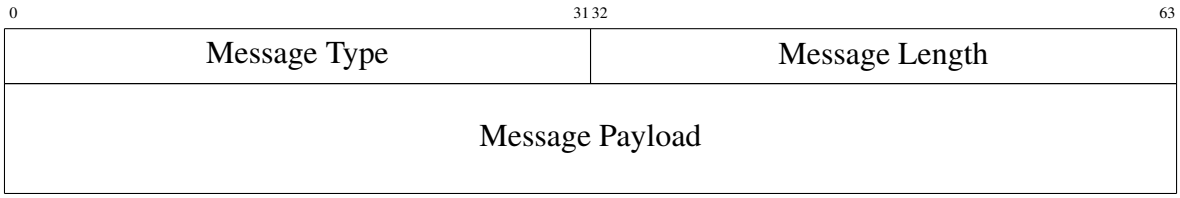


Figure 4.9: Southbound API message structure

identifier (identity) of the switch. Once the connection is made, per-switch data plane behaviour can be installed dynamically to reflect the constantly changing demand on the fabric. With each program keeping its internal state in a set of tables, the controller is able to request the content of the tables or specific entries allowing the global view to be maintained. If necessary, the controller can also update those entries if a controller-local decision is made. Finally, the controller is able to craft or resend packets to any connected switch through the southbound API much like the **packet_out** message in OpenFlow.

Using the same communication channel between the agent and the controller, the switches can also emit asynchronous requests to the controller. The protocol in its current format defines two different types of asynchronous requests, the first one is triggered when the data plane program in eBPF delegates the decision-making to the controller logic, similar to a table miss or an explicit forward-to-controller action in OpenFlow. This request contains the packet that triggers the miss as well as the associated metadata and is therefore similar to a **packet_in** message in OpenFlow. The second type of asynchronous message is a notify message that can be triggered by the data plane program to notify the controller of an event that took place without impacting the default forwarding mechanism of the packet. A use-case for the notify message might be to alert the controller that a particular threshold has been exceeded, for example, in case particular hosts are rate or quota-limited. Each notify event sent from the switch to the controller contains a user-defined identifier to specify the type of notification and an arbitrary user-defined payload that can be used by the controller.

In its current implementation, the southbound API is very straightforward, the TCP transport protocol is used between the controller acting as the server and accepting multiple concurrent connections from the switches acting as clients. Each message contains a short header and the payload of the message, as shown in Figure 4.9. The header is only 64 bits, containing

Type	Direction	Description
Hello	S $\leftrightarrow$ C	Handshake message between switch and controller
Install	C $\rightarrow$ S	Install eBPF ELF on the switch
Packet In	S $\rightarrow$ C	Packet sent from a switch to the controller
Packet Out	C $\rightarrow$ S	Send a packet from the controller to a switch's output port
Tables List	C $\rightarrow$ S	List all the instantiated tables
Table List	C $\rightarrow$ S	List the content of the table specified
Table Entry Get	C $\rightarrow$ S	Get a single entry from the table specified
Table Entry Insert	C $\rightarrow$ S	Update or Insert an entry with the key and value provided
Table Entry Delete	C $\rightarrow$ S	Remove an entry from the table specified
Notify	S $\rightarrow$ C	Asynchronous notification of an event with user defined payload

Table 4.3: Supported messages over the southbound API between the controller (C) and the controlled switches (S)

two 32-bit fields, the message identifier (opcode), and the length of the payload to be expected after the header. The payload itself is dependent on the message type and can be any of the messages defined in Table 4.3. To follow the aim of the proposed architecture to be platform and language-independent, the protocol messages have been defined using Google protocol buffers. Google protocol buffers is a language and platform neutral approach to serialise structured data and has been used widely to describe protocols' wire format.

In the current implementation the security model used is similar to the one of OpenFlow. In a production network the communication between the controller and the switches would use TLS, to authenticate the end-hosts and provide encryption over the network. The management network should be separate from the data network to isolate potentially new attack vectors created by the southbound and northbound API endpoints. Using TLS for authentication and encryption, as well as a separate management network for isolation, provide a strong protection in a data centre network. If an attacker manages to gain access to the management network, for instance by managing a privilege escalation outside the virtual machine, (s)he would still be unable to issue controller requests due to the TLS authentication. The TLS encryption, prevents any malicious user or third-party to listen to the management information and intercept potentially sensitive information, such as the BPF programs that are installed on the switches or the table entries in the devices.

A potential new attack vector present in BPFabric that is not present in OpenFlow or current SDN protocols is the ability to install data plane functions onto the network devices. If a malicious user gains access to the controller or is able to access and edit the data plane functions that are installed on the devices, it would be possible for malicious code to be appended to, or replace, legitimate functions. With a full access to the data plane functions, the attacker could mirror traffic, collect statistics or trigger a denial of service by dropping all the traffic to a specific host. The approach to resolve this issue is to apply the same security procedures to data plane functions than any other software as recommended by SDNSEC [56]. To ensure the integrity of the program and prevent tempering, the data plane functions should be digitally signed using a key pair. The private key should be securely held, possibly by an air-gap, and used to sign the data plane functions, while the public key is used by the controller and the network devices to check the signature and ensure the integrity of the data plane functions.

4.4 Implementation

4.4.1 Switch

As part of this work, two different versions of BPFabric software switches have been implemented. The first implementation uses the Linux raw socket interface to provide in user-space the functionalities described in the previous sections. The second implementation relies on the Intel DPDK framework [132] to perform high-speed and low-latency packet processing.

The first implementation has been designed to be used on any conventional Linux machine as a way to quickly test and evaluate a data plane program in eBPF without much consideration on the packet processing performance. The entire switch pipeline to receive, process and transmit the packets is within the same thread hence limiting the overall performance. However, by using mmap to share the receive and transmit ring buffers between user space and kernel space, the performance is on par or superior to user-space OpenFlow switches [133].

As this implementation is fully in user-space and based on the default Linux kernel socket interface, it can be used in containers and namespaces to allow large-scale experiments to be designed on a single machine. For this purpose, support for BPFabric in Mininet was also added [123], allowing existing network topologies and network scripts to be executed over the widely used Mininet environment. A current limitation of this implementation is the limited headspace between the *tpacketv2* header used in mmap to hold link level frame meta information and the Ethernet frame that has been received, limiting the packet metadata header to 14 bytes [134].

Although this implementation is able to show the use-cases for BPFabric, user-space software switches are designed for experimentation, as due to the networking stack overhead they are unable to cope with the line-rate speed of production networks. Therefore, a second high-performance software-switch implementation is provided using Intel DPDK, a set of libraries for fast packet processing in user space bypassing the traditional kernel networking stack. This implementation relies on the Intel DPDK poll-mode drivers and therefore can be used with a limited set of physical NICs and a few virtualized NICs such as in Xen [135], QEMU [136], VMWare ESXi, and Amazon ENA. Using the DPDK threading model, a single eBPF execution engine per core can be instantiated and by using DPDK memory ringbuffers with Linux hugepages, transmit and receive queues can operate without pagefaults. By using the libraries provided by DPDK and the concurrent execution engine, the resources of the machine executing the switch implementation can be fully used.

To highlight the clear separation between the control and data plane, both switch implementations execute the same control plane agent but each run their individual data plane for receiving, processing and forwarding the packets: Linux raw socket for the first one and Intel DPDK for the second one.

In both switch implementations, the userspace JIT compiler ubpf has been used as the eBPF engine. [137]. ubpf is a just-in-time compiler for eBPF to x86_64 architecture. It was originally designed as an Apache-licenced library for executing eBPF programs, and has now been included as part of the IOvisor project [131]. This implementation uses a modified

version of ubpf that allows the eBPF maps to be allocated if they have not been created, and relocate them before the code is JIT compiled. Other modifications have been performed to solve issues with the stack allocation in the JITed code and partial string relocation for easier debugging. In the IOvisor implementation of ubpf, function calls are supported but maps are not, therefore the code has been modified, initially to perform the appropriate kernel system calls to allocate and query the tables. Through experimentation, it was noticed that constant system calls between the user-space and kernel space requiring the data to be copied were a serious performance bottleneck and therefore the tables have been moved to user-space, allowing the entire eBPF execution to be kernel-independent. A consequence of extracting the tables to user-space is that new types of maps can be easily added that might be relevant in the data plane switching scenario addressed here but irrelevant for an eBPF kernel task such as a TCAM-type table.

Alongside the two software implementations, the early design of an eBPF native processor was written in HDL, targeting hardware platforms such as the NetFPGA or the SolarFlare Application Onload Engine.

4.4.2 Hardware Implementation

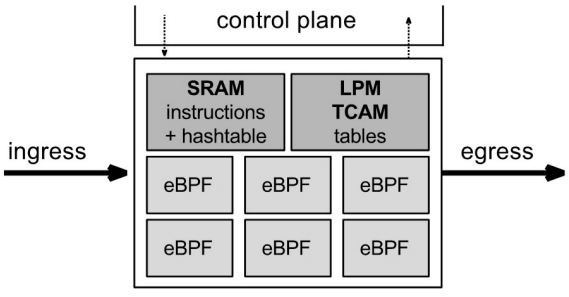


Figure 4.10: Hardware switch implementation overview

The execution of eBPF programs is limited in scope to individual packets that are received on the ingress interface and therefore parallelizing of the process is straightforward. Multiple eBPF execution engines can be collocated to process multiple packets in parallel and

increase the aggregate throughput the device is able to perform. Figure 4.10 shows a high-level representation of the switch seperating the ingress and egress queues with the processing pipeline. Packets are received on the ingress ports and queued following traditional switch design strategies (e.g., FIFO), and the eBPF engines, when ready, fetch and process the first packet from the queue.

The eBPF instruction set is simple and therefore designing a processor capable of executing the instruction set natively is relatively straightforward. Based on the instruction set, a RISC style processor can be designed with only a few stages to fetch, decode and execute the instructions. The two main components of the processor being a Register Transfer Machine (RTM) storing 11 general purpose 64 bits register and an arithmetic logic unit (ALU). One of the main considerations for the design of this processor is the performance with respect to the complexity. The speed at which a single core can process a packet is very important as the switching latency should be kept as low as possible, but with an increase in complexity to reach higher throughput, more hardware real-estate will be used that could be used for collocated eBPF cores. Hence, the objective is to minimise the packet processing time while collocating as many eBPF execution cores as possible.

A very widely used optimisation for processors is the inclusion of instruction pipelining, allowing instruction-level parallelism within a processor by spliting the instruction cycle in multiple logical blocks, thus forming a pipeline. Pipelining can increase the throughput of a single processor, but increases the complexity and the overall latency of the instruction path due to the necessary intermediary registers in the pipeline. Another consequence is the necessity for the pipeline to be stalled or even flushed when the next instruction is unknown, such as a conditional jump, further increasing latency. The latter is critical for eBPF, as most packet processing functions are branching heavily to conditionally parse the large variety of packets headers. To keep the latency and complexity low, the design of the proposed hardware implementation is a single instruction execution stage without pipelining. This design allows for minimum impact on packet processing latency while using little hardware real-estate.

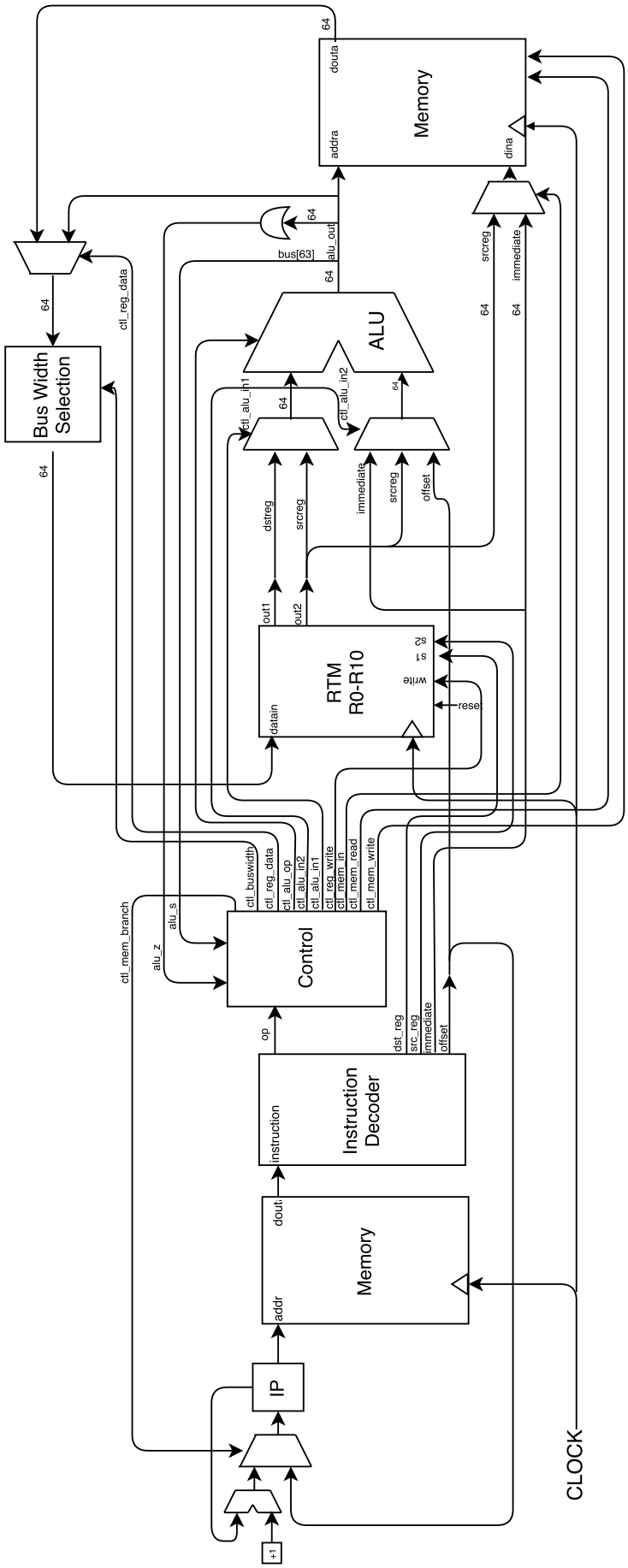


Figure 4.11: Verilog Implementation of an eBPF execution unit
https://netlab.dcs.gla.ac.uk/bpfabric

In Figure 4.11, the high level diagram of the Verilog implementation of a single eBPF execution engine is shown. This diagram shows the necessary control lines, required buses and multiplexing to support the entirety of the eBPF instruction set, excluding the CALL instruction. The implementation has been done in Verilog as a soft processor core using Xilinx ISE in order to simplify future integration with the NetFPGA 10G development kit. Once a NetFPGA implementation is done, demonstrating the potential and performance of eBPF, a dedicated ASIC with a larger port density, higher switching speed and lower power consumption would be possible. This implementation has been evaluated through software synthesis to show that all the operations are working as desired, with a single clock cycle for all operations except jumps and memory read requiring two cycles. A rough resource utilisation of the eBPF engine can be extracted from Xilinx ISE and compared against the available resources in the Virtex-5 available in the NetFPGA 10G platform. This simple design uses only about 3% of the slices in the FPGA, 12% of the buffers and 10% of the 48-bit Digital Signal Processor (DSP). Using this resource utilisation as reference, approximately 8 eBPF cores can be placed on the FPGA, but this does not include the necessary space requirements for the NetFPGA packet pipeline or additional memory such as TCAM. To better understand the resource requirements and possible number of eBPF engines, future work in FPGA integration would be required.

The design of the processor as well as multiple parameters from the NetFPGA existing implementations can be used to estimate the achievable performance on an FPGA implementation. The performance of the proposed processor is limited by the clock speed of the core. Assuming that most instructions can be executed in a single clock cycle and that the FPGA is clocked at 250MHz, a single instruction can be performed in 4 nanoseconds. Assuming a complex eBPF function with a critical path of 100 instructions, the per packet execution time is 400ns. From this estimation, the line-rate throughput achievable can be calculated in the worst-case scenario of back-to-back minimum size segments (64 bytes), and the best-case scenario of back-to-back maxmimum size segments (1500 bytes), as shown in equation 4.1. Hence based on these assumptions, a single eBPF core can sustain a throughput of 1.28Gbps in the worst-case and up to 30Gbps in the best case scenario.

$$Throughput(bps) = \frac{PacketSize(b)}{eBPFExecutionTime(s)} \tag{4.1}$$

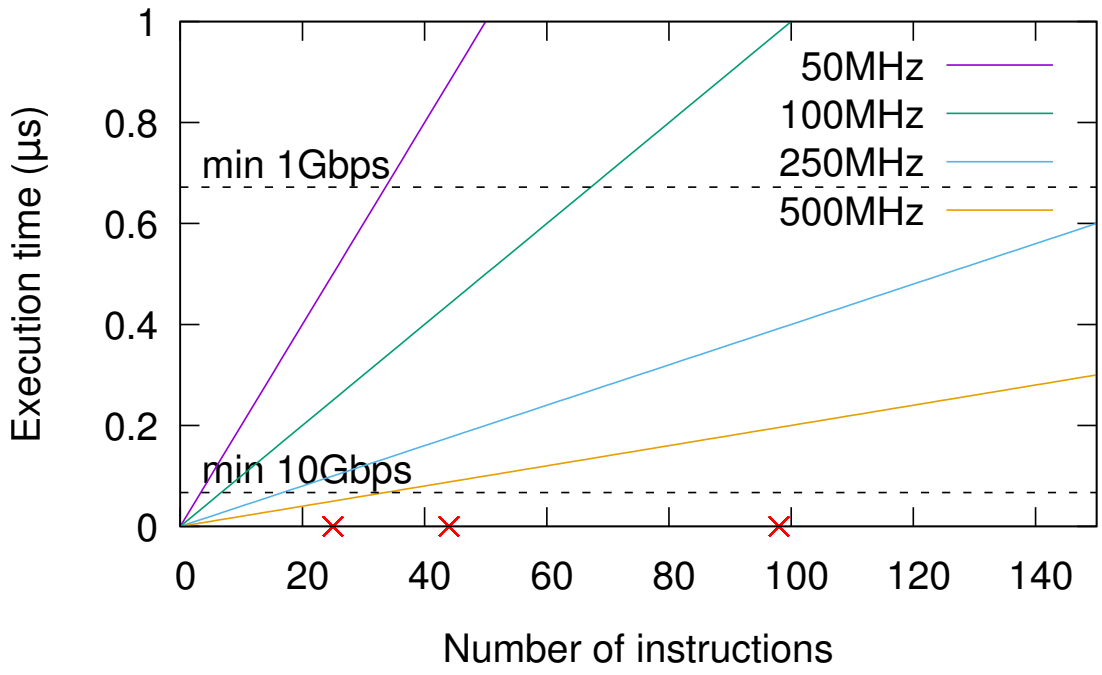


Figure 4.12: Maximum number of eBPF instructions supported to achieve line-rate depending on the FPGA frequency.

Figure 4.12 shows the time taken to execute a single eBPF program depending on the clock frequency of the hardware implementation and the number of instructions to execute in the program. The three red crosses on the x axis represent the number of instructions in the critical path of three example eBPF data plane functions (layer 2 learning switch, packet size distribution and Exponentional Weighted Moving Average (EWMA) based anomaly detection). Additionally, horizontal dotted lines have been plotted to show the maximum allowed processing time for 1Gbps and 10Gbps networks when operating in the worst case scenario with only minimum sized packets. The processing time is calculated as the maximum time between two packets, in the case of Ethernet networks demonstrated here, it is the time to receive the frame and the inter-packet gap totalling to 84 bytes worth of transmission time. This figure demonstrates that at 1Gbps, even with a fairly low clock speed, it is possible to execute relatively large eBPF programs even in the worst operating conditions. At 10Gbps, the processing requirements become more complex, but this can be addressed by allocating

more eBPF processing cores on the FPGA fabric and leveraging the increased clock speed of high-end FPGAs. The processing performances shown in this figure are **per** eBPF core and therefore as the number of cores increases the time per packet increases accordingly.

4.4.3 Controller

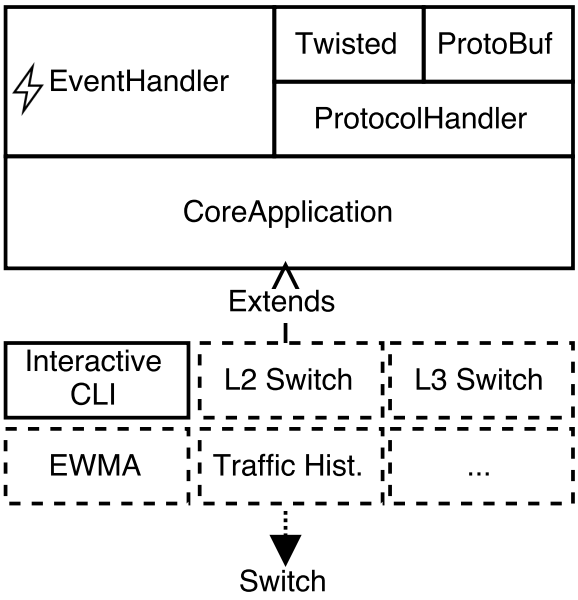


Figure 4.13: BPFabric controller architecture

The central controller has been implemented using Python to provide an easy application framework to develop and extend new controller functions. The architecture of the controller is shown in Figure 4.13 and shows the separation between the core and the applications that have been built using the core. The controller was designed using the python Twisted framework for asynchronous packet handling to easily allow many simultaneous switch connections at low latency. For the communication and serialization of the messages defined previously in section 4.3.4, the Google protocol buffer is used. The twisted event handler and the protobuf message definition are used by the ProtocolHandler to establish the persistent connection between the controller and the switch, and serialize/deserialize the packets. The CoreApplication is the parent object to any other application, and defines the minimum logic to establish the server socket and perform the hello handshake when a new switch establishes a connection to the controller.

Following the approach of popular Python OpenFlow controllers such as Ryu or POX, all control logic internally and between the modules is event-based. This approach allows the application modules to listen and react to specific events generated by the switches such as the *Hello* message sent during the connection handshake that can be used to automatically install an eBPF program onto a switch. The EventHandler is used to create an independent event for each message type as well as on connect and disconnect events, signifying that a TCP connection has been established or closed, respectively.

```
$> python ./controller.py
------------------------------------------------------------------------------
    eBPF Switch Controller Command Line Interface - Netlab 2016
    Simon Jouet <simon.jouet@glasgow.ac.uk> - University of Glasgow
------------------------------------------------------------------------------

Documented commands (type help <topic>):
========================================
help

Undocumented commands:
======================
EOF  connections

(Cmd) Connection from switch 00000001, version 1
Installing ./learningswitch.o on 1
[..]

(Cmd) 1 tables list
(Cmd)
     name    type    key size    value size    max entries
=========  ======  ==========  ============  =============
  inports    HASH           6             4            256
=========  ======  ==========  ============  =============

(Cmd)
```

Figure 4.14: CLI Interface

The main application that is part of the codebase for the controller is the interactive Command Line Interface (cli.py) that allows a user to control multiple switches interactively without any knowledge of the topology or program to be installed. In the CLI, a user can manually install eBPF programs to any connected switch, list the tables and their associated type and size, as well as, query, update, and remove entries within the tables and display the notification events. Using this generic CLI, the entire state of the network and the eBPF programs on the switches can be monitored at run-time, which is useful for experimentation and debugging. Controller applications, similar to what can be written for OpenFlow, can also be created to perform specific functions, automatically install an eBPF program onto the

switches, and react to **packet_in** and **notify** events.

Using a simple southbound API and relying on widely used libraries for the design of the controller, the code required to interact with the Agent in the switches can be kept simple. The entire implementation of the core is less than 200 lines of code, and therefore adapting this to any other language is a trivial implementation task. Figure 4.14 demonstrates the use of the CLI to automatically install the `learningswitch` data plane function to the switch with a datapath identifier of **1** and listing the tables instantiated by this particular function.

4.5 Use Cases

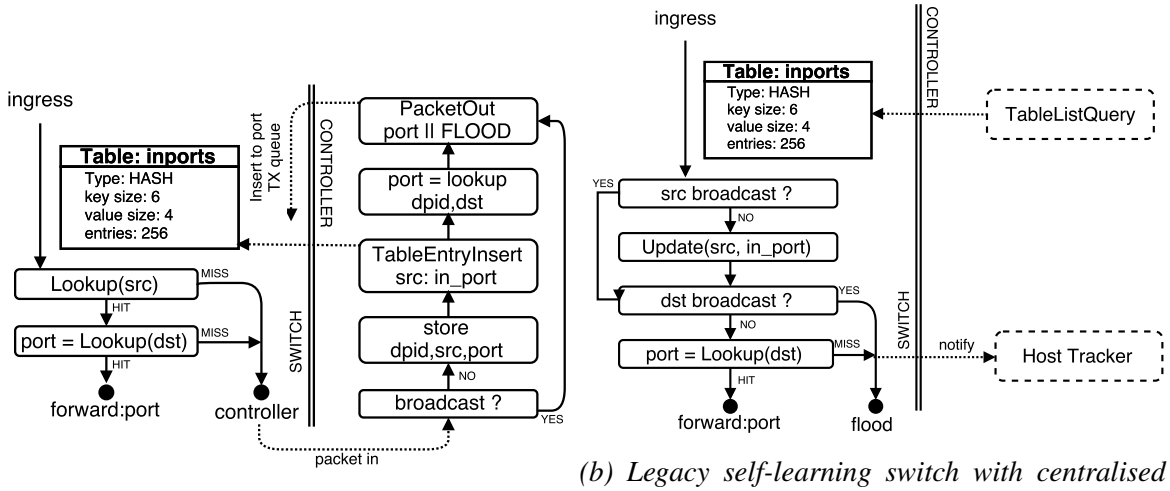
Program	**# LoC**	**# Instr.**	**# Tables**	**# Table Lookups**	**# Table Updates**	**Table Space (bytes)**
Centralised L2 Switch	20	25	1	2	0	$10m$
Learning L2 Switch	22	28	1	1	1	$10m$
Pkt. size distribution	10	44*	1	1	1	$8n$
Pkt. inter arrival time	37	98*	2	3	2	$24 + 8n$
EWMA	24	99*	1	1	1	$32p$
TCP latency	75	161*	1	1	1	$24 * f$
TCP flow arrival	43	82*	1	1	1	12

m = number of unique MAC addresses, p = number of ports on the device, n = number of buckets in the histogram, f = number of flows
* is including the self-learning switch logic

Table 4.4: Example programs implemented using BPFabric, with associated complexity, table operations and memory requirements.

In this section, some example data plane and control plane tasks that can be implemented using BPFabric are demonstrated. These functions have been designed to highlight the utility of the proposed framework for a wide range of applications. The following example programs can be run indifferently on either switch implementation. Table 4.4 shows the complexity in number of lines of code (C) and resulting number of eBPF instructions, the maximum number of table operations required, and the memory requirements for state keeping. Demonstrating that complex functions can be written with low complexity is crucial, as the flexibility must not be outweighted by the development effort by the network operator.

4.5.1 Layer 2 Learning Switch



(a) SDN centralised learning switch

(b) Legacy self-learning switch with centralised monitoring

Figure 4.15: Implementations of a learning switch in an SDN centralised manner and in a legacy self-learning approach.

The typical example that has been used as the "hello world" for new SDN or controller implementations is a simple centralised layer 2 learning switch. Demonstrating a layer 2 switch is a useful and practical way to demonstrate the ability of individual switches to delegate packet forwarding responsibility to the controller, make a central decision based on the global view of the network, and update the switch accordingly. Figure 4.15 shows two different implementations of a learning switch, showing both the ability to delegate responsibility to the controller and secondly the ability to self-update.

Centralised Learning Switch: The first implementation shown in Figure 4.15a represents the centralised SDN approach, with the switch delegating the entire decision process to the controller. In this scenario, a lookup table is maintained to hold the mapping between an Ethernet MAC address and the input port of the packet. When a packet is received and the source or destination address is not present in the lookup table, the controller is notified. The controller program stores the switch identifier, associated source address and input port in a table, then emits a table insert request to update the switch lookup table. Finally, the controller looks up the port associated to the destination address and emits a **Packet Out** request to the switch specifying either the looked up port or a flood action if the entry was not found. To demonstrate the simplicity of implementing such data plane function, the relevant

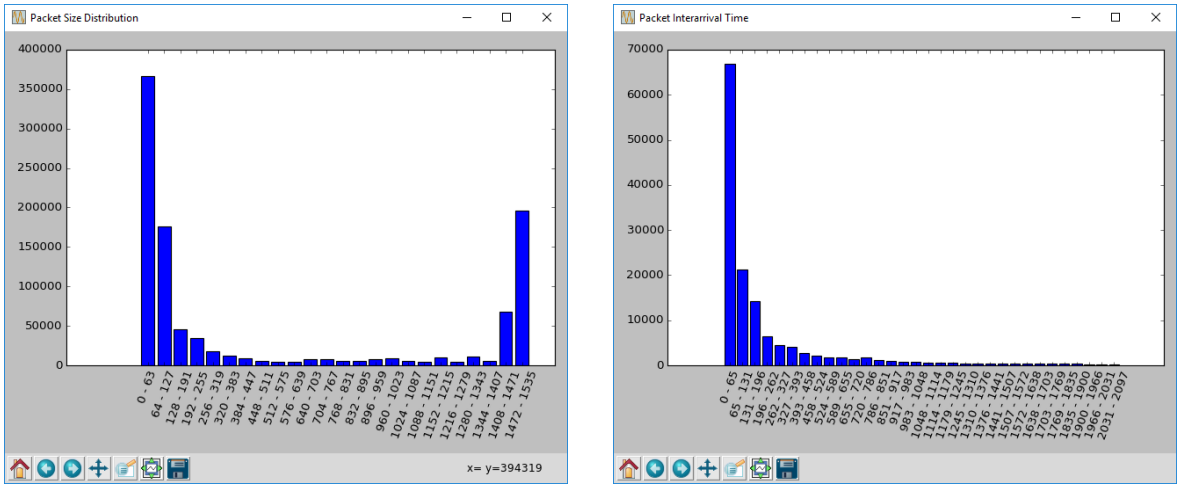
code is listed in appendix A.1.1. From the code, it can be seen that the boilerplate code to include relevant headers and define the maps is longer than the core function implementation. Additionally, the centralised controller code handling the **Packet In** events raised by the data plane function can be seen in appendix A.1.2.

Self-learning Switch: Secondly, Figure 4.15b demonstrates that a traditional (legacy) self-learning switch can be easily implemented using the data plane ability to self-insert and update entries in the switch tables. For every incoming packet, the lookup table is updated if the source address is unicast and the destination port looked up, if an entry is found the packet is forwarded or otherwise it is flooded. In this scenario, the controller is not responsible for making the forwarding decision, but a big difference from a traditional switch is the ability of the controller to be notified if a new entry is added, and the ability to query the content of the tables at any point in time. The support for self-updating tables is necessary for stateful packet processing and to delegate some of the processing to the network devices. However, self-updating is not supported in current SDN implementation: OpenFlow delegates the entire responsibility to the controller, while P4 considers self-updating as a possible future improvement.

4.5.2 Network Telemetry

Network telemetry is used to verify that the network is behaving as intended and to monitor any changes occurring in the network over time. Through the process of continuously collecting network metrics, the central controller or third-party management applications can gain granular visibility into the network behaviour in order to model and predict future trends. This insight into the network behaviour can be used to adapt the fabric as the demand evolves, migrate applications or virtual machines (VMs) to improve overall network utilisation [138], as well as improve policies to meet customers' Service Level Agreements (SLA). Moreover, the telemetry data can be correlated with other logs to identify potential anomalies, security risks and for debugging purpose.

Telemetry support has been present in network devices for a long time through different



(a) Packet size distribution histogram *(b) Packet interarrival time histogram*

Figure 4.16: Telemetry histograms based on reported values from dedicated data plane functions

implementations and providing different levels of insight into the network traffic. The most common implementation of telemetry might be the Simple Network Management Protocol (SNMP) that allows a monitoring station to pull vendor-specific values from the devices. However, with the growth of network sizes and the associated cost-incentive to better utilise the available resources, telemetry is currently being heavily revisited and reimplemented in a push-model where the devices stream events directly to data collection points.

Using BPFabric, two telemetry examples are demonstrated using both the pull and push approaches to demonstrate that user-defined telemetry metrics can be defined and collected from the infrastructure. Using the pull approach, the controller can request specific data from particular devices, preventing continous utilisation of the management network when the metrics are rarely used. The metrics gathered using a pull approach are often as a consequence of a specific task being executed in the controller that requires a particular metric to continue operation. The push approach on the other hand, continuously report metrics of interest to the controller. Using this approach the management network and controller are more heavily loaded but it allows the controller to have an up-to-date near realtime view of the data plane operation.

Packet Size Distribution: A data plane program has been implemented to store the per-switch packet size distribution as a histogram. Using an array table to store the buckets

for the histogram and the length available in the packet's metadata, it is possible to keep track of the packet distribution over time. Using the table list control message, the controller can query (pull) the current state of the histogram in any switch where this program was deployed. This program can provide insight of the nature of the traffic, for instance if the traffic is from real-time applications with overall small packet size or batch-based with MSS-sized, packets as well as determining trends over long periods of time to determine network normal behaviour. Figure 4.16a shows the overall packet size distribution monitored by the different switches and collected by the controller in a BPFabric environment. The user interface is updated every few seconds on the controller to display an up-to-date view of the packet size distribution.

Packet Inter Arrival Time: The second telemetry example measures the packet inter arrival time at a particular switch. The function is designed around two tables, one to store the histogram of the inter arrival time and a second one to keep track of the last packet received. At a regular interval, every 2048 packets in this case, the histogram data is pushed to the controller using the southbound notify message. The inter-arrival time of packets, and the associated mean and jitter can provide information about the congestion state of the network, the burstiness of the traffic impacting real-time streams, and help with traffic classification. Figure 4.16b shows the histogram distribution of the packet interarrival time as seen by the controller. Differently from the the packet size distribution, this histogram is updated automatically when switches report new metrics to the controller.

4.5.3 Lightweight Anomaly Detection

In the previous sections, this work has shown the forwarding and telemetry aspects that have been commonly associated to the function of a network switch. This example demonstrates that some functions that have been typically delegated to special middleboxes in the network can be implemented as part of the switching fabric.

In this example, a lightweight anomaly detection algorithm is implemented using Exponential Weighted Moving Average (EWMA). EWMA is a statistic to average data over time and

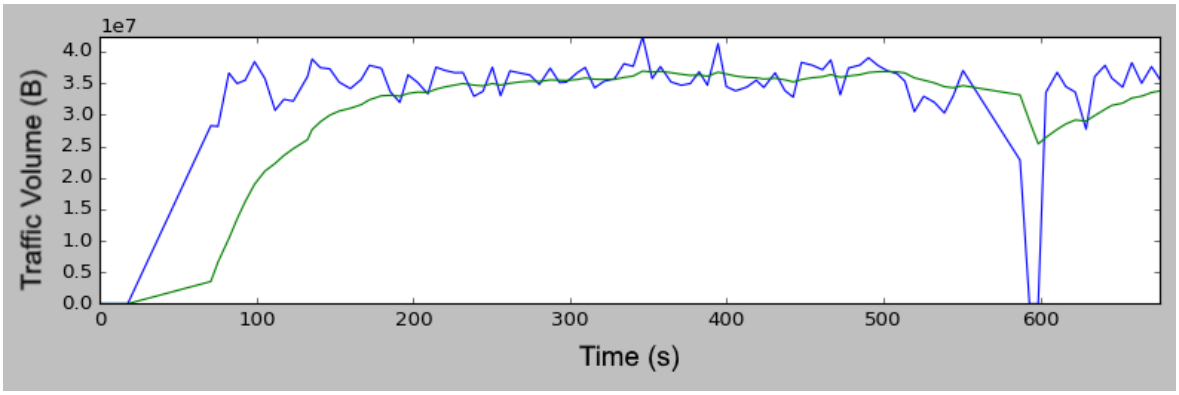


Figure 4.17: Reported EWMA volume average in bytes (blue) and predicted volume (green) over time by the switch with an introduced anomaly (link failure) at t=600s

reduce the weight of previous measurements the further away in time they are. Using this statistical value on some networks metrics, provides insight on the normal operating condition. Significant deviation from this normal behaviour can then highlight anomalies within the network. Multiple network metrics can be used to represent the network behaviour, such as the number of packets, their size, inter arrival time or the volume of traffic transmitted or received.

This implementation computes the EWMA value of the volume of traffic received on every port of the switch for every incoming packet. The information is maintained in an array map containing one entry for every port. At a specific time interval, 5 seconds in the current implementation, the EWMA value is calculated and compared against the threshold values. If the computed value is not within the expected bounds, a notification is raised to the controller signaling an anomaly.

Using this approach, many lightweight network functions can be implemented, but the limitations of the device should be considered. As described in section 4.3.2, in order to provide a realtime execution environment, loops should be avoided which can be an issue when implementing more complex network functions. Also, switches typically do not have hardware support for floating point arithmetic as it is of no use for network operations and therefore limit the complexity and accuracy of the computations. An example of the latter is the weight in the calculation of the EWMA $weight$ that should be between 0 and 1. A widely deployed implementation of EWMA is the RTT estimator used for TCP congestion control as defined

in RFC793 [139] with a suggested weight of 0.1. However, Van Jacobson and Karels suggest to use 0.125 in practical implementations as it can be implemented using shifts and is close enough to the suggested 0.1 [140]. This EWMA implementation uses the same weight as suggested by Van Jacobson and Karels to allow the computation to be quick without with floating point support ($weight = 0.125 = 2^{-3}$). The proposed approach is designed to perform lightweight packet processing at line-rate, therefore more complex functions (e.g., requiring loops, floating point operations, large memory) are not designed to be executed as part of the network devices. However, these complex functions can rely on BPFabric to classify and redirect traffic at line-rate towards dedicated high-performance middleboxes or virtual Network Functions (vNFs).

4.5.4 Omniscient TCP

The driving motivation for BPFabric and the argued benefits and necessity of data plane programmability results from the work on centralised resource management and the current limitations of existing SDN implementations. The previous use-cases have demonstrated that new types of data plane functions can be implemented in BPFabric that are currently unachievable with OpenFlow. This section leverages this increased programmability and insight into the network, to better provide network operating metrics to fine tune the TCP congestion control parameters. As described previously, the main limitations of OTCP have been: the end-to-end latency measurement, the ability to dynamically and continuously account for the number of active flows along a path, and the size distribution of packets. The packet size distribution data plane function has been described in the telemetry examples and can be used in OTCP to tune the congestion control parameters.

To monitor the latency with OpenFlow and make it available to the controller, the measurements must be computed using a combination of production and management network which can induce inaccuracies and make the measurement process increasingly complex. In BPFabric, the measurement of the latency can be performed passively by the network devices and reported to the controller when necessary. The passive measurement performed in this

data plane function has been inspired by Ruru, a real-time TCP latency measurement tool using DPDK [141]. In this approach, the latency is measured at a specific point along the path between the source and destination. By relying on the three-way handshake of TCP, the intermediary point can measure the delay between the initial SYN packet, the SYN—ACK response and the first data ACK. The RTT between the two hosts can then be calculated using the time difference between the ACK and SYN, as well as, the individual latency between the intermediary device and each host.

The implementation of this function in BPFabric is straightforward and can very accurately measure in real-time the individual latency of TCP flows in the network. For every TCP 3-way handshake packet that is forwarded through the switch, the timestamp available in the packets' metadata is stored in a map. Each entry in the map is defined by a key mapping the TCP 4 entry tuple (src ip, dst ip, src port, dst port) to the individual timestamps for the SYN, SYN—ACK and ACK packets. Once the third packet of the handshake is received, a *Notify* event is emitted to the controller advertising the latency for this flow and the entry is removed from the table. This function can be instantiated on just a few of the network switches along the different paths in the network for the purpose of OTCP. However, the same network function can be used for other purposes such as network debugging. By deploying this function to multiple switches along one path, the individual latencies between the hosts and the switch can be measured and used to troubleshoot network anomalies.

Figure 4.18 shows a histogram representation of the TCP flow latencies reported to the controller by the switch with the above data plane function deployed. The experiment was performed by replaying a 5-minute packet trace from a production data centre [93] in a Mininet environment. Over the 5-minute trace, 9200 TCP flows have been established between different end-hosts. The average latency between the end-hosts is close to 80ms and can be seen for most of the low data points in the histogram. This figure provides additional insight into the operating environment, for instance, the flows with latencies between 400ms and below 500ms imply that the SYN—ACK or final ACK had to be retransmitted. This retransmission time close to 400ms is indicative that the end-host retransmitting the packet is running

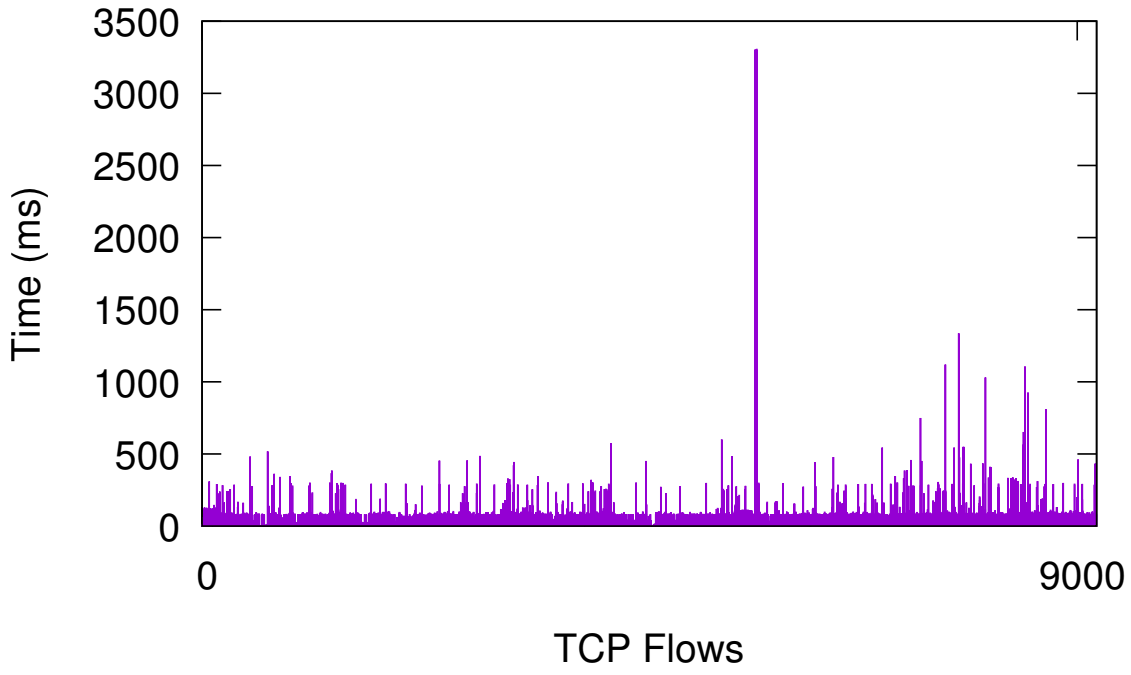


Figure 4.18: Reported TCP latencies to the controller

a Windows operating system, a retransmission closer to 200ms would indicate a Linux or BSD operating system. The large latency spike of ca. 3 seconds signals that one SYN packet was lost and the connecting host relied on RTOinit to retransmit the packet. Finally, the multiple spikes in latency in the final quarter of the figure suggest that at this point in time a switch along the path was oversubscribed, resulting in more retransmissions and exponential backoff.

In order to provide the flow accounting necessary for OTCP, the data plane implementation proposed relies on counting the number of flow arrivals and departures within a specific time window. The data plane function matches TCP packets with the SYN or FIN flag set in the header, respectively incrementing the arrival and departure count. After a specific time window, currently 5 seconds, a notification is raised by the switch informing the controller and the count is reset to zero. This simple approach allows the controller to maintain a full view of the flow departure and arrival rates. Figure 4.19 shows the plotted arrival and departure rates of TCP flows reported by a single switch to the controller. Using this approach each switch in the network can report to the controller the number of flows observed at a particular point in the network, allowing the controller to keep an up-to-date view of the number of ac-

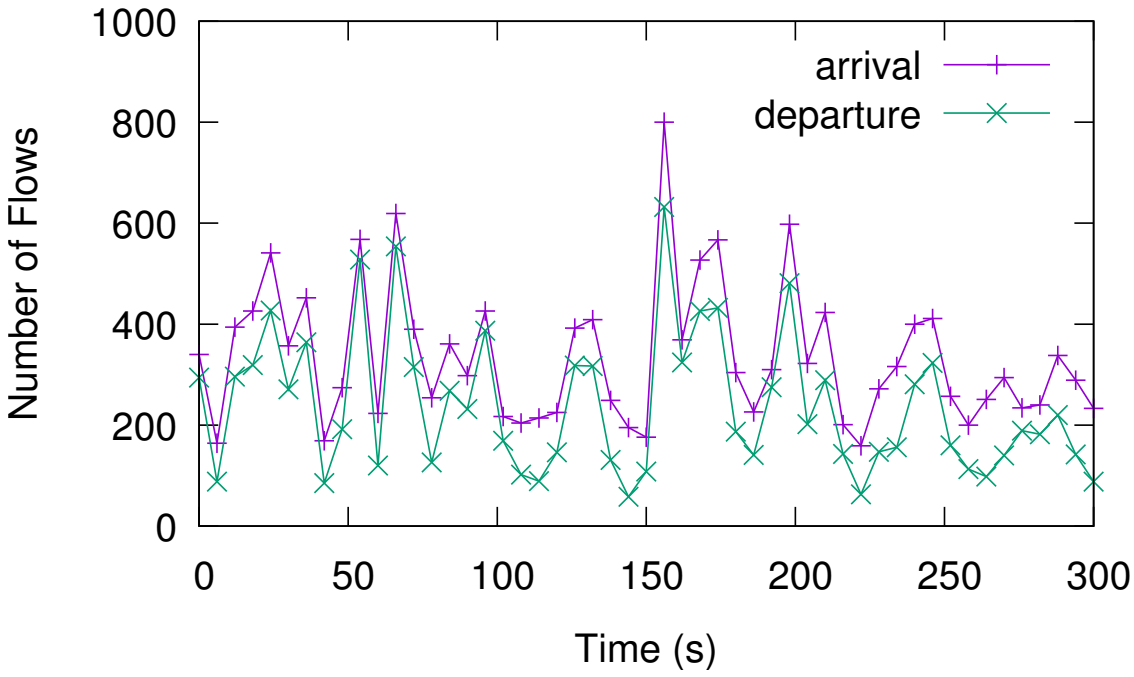


Figure 4.19: Real-time monitored active flows

tive flows per link. This evaluation shows the flow arrival rate and departure of a production data centre using the same data centre traces as the previous experiment.

To show the simplicity of this implementation and the ability of BPFabric to efficiently provide insight in the network operation, the source code of these data plane functions is listed in Appendices A.2 and A.3. These data plane implementations demonstrate that, using BPFabric, the highly dynamic characteristics of a DC network can be measured and reported to the controller at a very fine granularity. Using this insight into the network, the controller can shape the operating environment and resources to match the temporal conditions of the infrastructure and consequently improve resource usage.

4.6 Evaluation

The presented framework provides a level of flexibility to the network operators by allowing a large number of functions to be implemented using a high level programming language. It comes to question whether the added flexbility provided by this approach comes at a performance cost. In this section, the performance of BPFabric is compared against state of the art

SDN switch implementations.

4.6.1 Throughput

Benchmark against the State of the Art

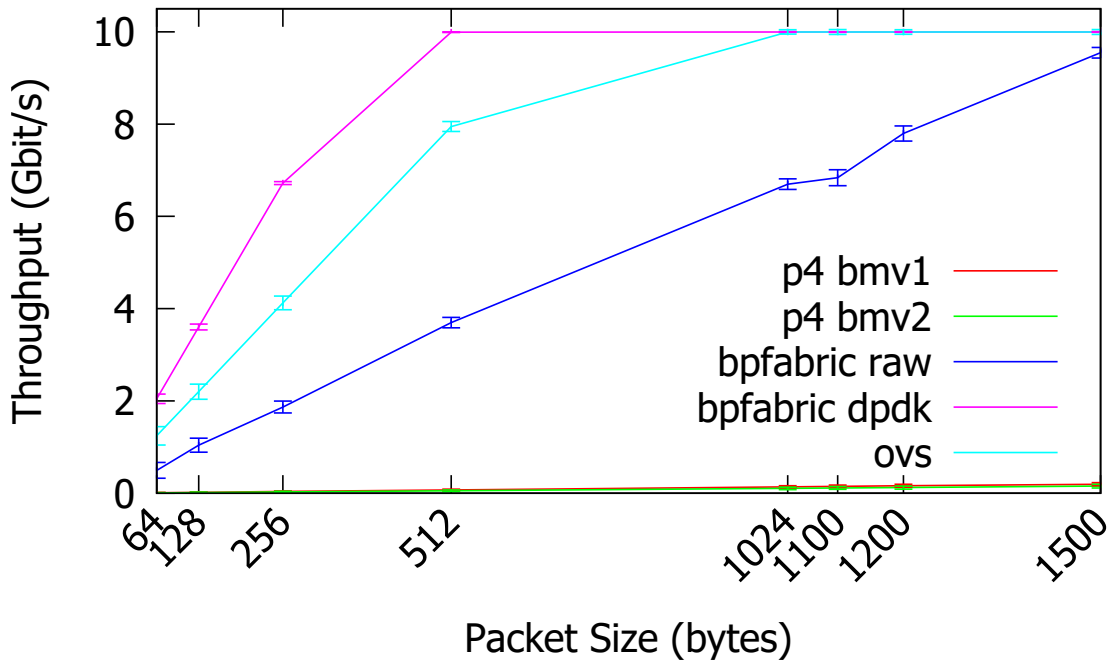


Figure 4.20: Performance comparison of P4 behaviour models, OpenvSwitch and BPFabric for a layer 2 learning switch.

Figure 4.20 highlights the performance of BPFabric compared to state of the art SDN software switch implementations. BPFabric is compared against Open vSwitch, the most widely deployed software switch for SDN environments, and both implementations of the P4 behaviour model. The P4 language has quickly evolved over the last few years and results in two different switch implementations, the first one is the implementation presented in [18], the second one is a complete rewrite using C++11 that was designed for more flexibility and target independence.

These experiments have been run on a high-end modern desktop machine with a 6th generation Intel Skylake processor (6700k), 32GB of DDR3 memory clocked at 3GHz and PCI express 3.0 for the network card. The network card is an Intel X710, with four 10Gbps inter-

faces that is compatible with Intel DPDK. This machine was used for these experiments as DPDK's performance is related to the per-core clock speed due to the polling mode drivers, hence, the 4GHz clock speed of these processors is more suitable than a larger core count at lower frequency of most Xeon Server processors. The machine is running Linux with a kernel 4.4 with full Intel Skylake support. The throughput measurements have been performed using Intel DPDK pktgen, a high performance and highly reliable traffic generator [142]. Each measurement was performed using pktgen *rfc2544.lua* script designed specifically for throughput testing on a second machine with the same specification as above and repeated 10 times for consistency. The network has been setup as a loopback, with two links between the traffic generator and the switch has shown in Figure 1 of the benchmarking topology of RFC 2544 [143].

To fairly compare the switching and packet processing performances betwen these different switches, OvS operates in a normal learning switch mode while BPFabric and P4 behaviour models both execute an equivalent learning switch data plane function.

In this experiment, the P4 behavioural models never exceed a throughput of 200 Mbps. Both P4 behaviour models have been designed as experimental software switches and therefore are not expected to perform at very high speed. The raw socket implementation of BPFabric performs linearly based on the packet size, with most of the execution cost being the numerous calls to the kernel when packets are sent and received. The slight performance degradation for packet size of 1100 bytes can be associated to the cost of accessing a second page in memory which is quickly mitigated as the packet size increases. The kernel space implementation of Open vSwitch is able to reach 1Gbps for minimum sized packets and reach the line-rate of the NIC for 1kB packet. Finally, the DPDK implementations of BPFabric outperforms both OvS and P4 behavioural models, performing two times better than OvS for minimum sized packets and achieving line-rate with 512 bytes packets.

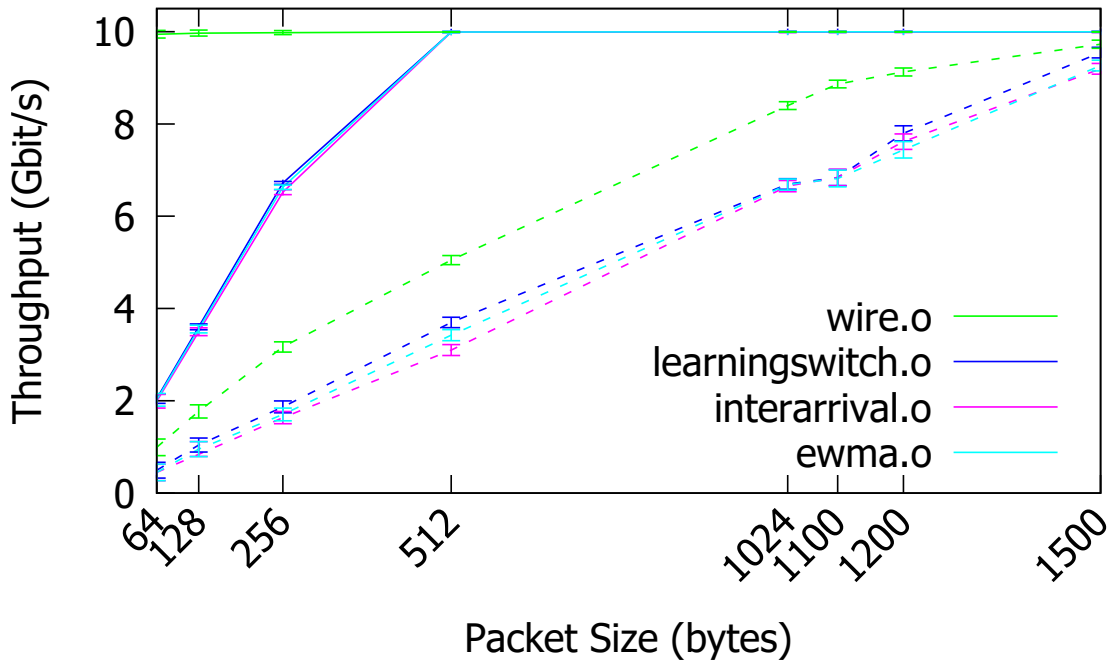


Figure 4.21: Performance comparison of example programs between Raw socket and DPDK implementation.

Impact of table Operations

As the complexity of data plane function increases and the number of table operations expands, the cost of execution rises accordingly. Figure 4.21 shows the performance achievable with some of the example data plane implementations described previously. This evaluation was performed using the same environment and testing methodology as for throughput and each data plane function was evaluated on both BPFabric implementations. Each switch was restarted between experiment rounds with the data plane function installed on startup. The solid lines represent the DPDK implementation and the dashed lines the raw socket implementation.

The most simple data plane function evaluated is the *wire* implementation, simply forwarding packets between two ports and hence not requiring any table to operate. Without any table operations, the DPDK implementation is able to achieve line-rate even for minimum sized packets, while the raw socket implementation performs better than other functions with table operations. This difference in performance implies that the performance degradation is a consequence of the software implementation of the tables rather than an inherent limitation

of BPFabric. Most data plane functions will require tables to maintain state between packets, therefore this work also evaluates a learning switch, a packet inter arrival time telemetry function, and a lightweight EWMA anomaly detection. The number of lookup and update operations each function requires per packet has been shown previously in Table 4.4. From this experiment, it can be seen that the cost of table operations prevents the DPDK implementation to achieve line-rate until 512 bytes packets. However, the performance of the learning switch with 2 table operations and the interarrival time with 5 operations are marginally different.

This high cost of table operations is to be expected in a software switch implementation since no optimised memory hardware is available as it the case in hardware switches. A commodity switch will have specialised memory available, such as TCAM, in order to perform lookups and updates at line-rate. This performance limitation is not inherent to BPFabric, and OpenFlow switch implementations are facing the same limitation with a significant performance penalty when the number of table lookups increases [144].

Performance Scaling

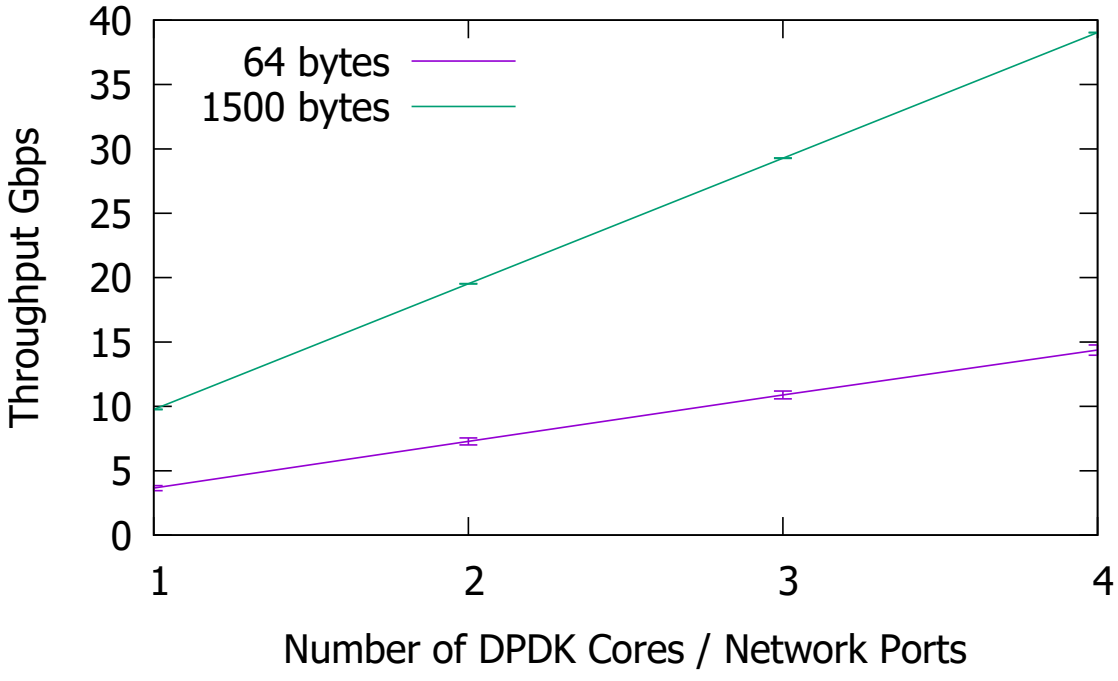


Figure 4.22: DPDK per-core performance

Intel DPDK has been designed to leverage the multiple cores of modern CPUs in order to achieve very high performance at low latency. In most implementations a single queue is attached per network interface to handle the packets and then each queue is delegated to a core on the processor. The consequence of this distribution is that the performance of one network interface is directly dependent on the per-core performance. With the number of network interfaces increasing and the number of cores scaling accordingly the performance should be close to linear. These scaling properties are related to the fact that the processing of each packet is decoupled and therefore it is never necessary to wait until another packet has been processed. This experiment evaluates these scaling properties with the same machine as described previously. Each of the NIC's ports have been allocated to a processing core on the CPU and each queue has its own 2GB hugepage to store packets.

Figure 4.22 demonstrates the scaling performance of the DPDK switch implementation running the learning switch data plane function. This evaluation compares the maximum throughput achievable in the worst case scenario of forwarding only minimum sized packets and in the best case scenario with MSS sized packets. From the figure, it can be seen that the processing performance scales linearly as the number of cores and ports increases. With 4 cores, BPFabric can forward MSS-sized packets at 40Gbps, and achieve up to 14Gbps in the worst case scenario with min-sized packets.

4.6.2 Controller performance

Although the current design of the BPFabric controller has not been optimised for high processing performance at the controller, knowing the achievable performance is necessary when designing the data plane functions. Some of the design decisions, in particular relying on Python for the controller, can impact the maximum processing performance of the controller. Popular OpenFlow controllers written in Python such as Pox and Ryu have demonstrated the benefits of simple APIs for SDN programmability at the cost of lower packet processing performance. BPFabric follows the same approach, but a major difference is that, by relying on Google protobuf to define the packet format, a dedicated high-performance

controller can be easily implemented once prototyped in the Python implementation. In OpenFlow, the protocol is lengthy and convoluted, and migrating one particular implementation to a different codebase is a difficult task, especially if the target is not one of the mainstream controller implementations.

This evaluation measures the performance of the BPFabric controller in handling requests from the switches. With a large number of devices or a low performing controller, the number of requests emitted by the switches could overwhelm the controller resulting in degraded performance. The protocol in its current form allows 3 messages to be issued by the switches to the controller. The 3 messages are: the *Hello* message issued during handshake, the *PacketIn* event when a forwarding decision needs to be done at the controller, and the *Notify* event raised by the switches to provide additional information to the controller. The *Hello* message should only be issued once per connection, although a malicious user, a bug in the implementation, or a management network failure could result in a large amount of requests to the controller. The *PacketIn* message can be issued as much as once per packet and per switch, it is therefore the most critical aspect of the controller and is most likely to be the one bounding the performance of the controller. The performance of the *PacketIn* handler is highly dependent on the controller implementation, in this evaluation the handler processes act as a centralised layer 2 learning switch. Finally, the *Notify* event can be raised at any time by the data plane function but is designed to be used sparsely to report information to the controller.

The experimental evaluation performed is using the same environment as previously. The measurement performed is based on the number of requests that can be sequentially issued to the controller by the switch each time waiting for the previous request to be completed before issuing the next one. The *Hello* packet performance is measured based on the time between issuing the message and receiving the matching *Hello* by the controller. The *PacketIn* is measured based on the time difference between issuing the message and waiting for the *TableEntryInsert* as a result. Finally the *Notify* messages are one-way only, not requiring the controller to reply with a response, therefore the performance is based on sending the packet

and receiving the TCP ackowledgement. For the three experiments, custom scripts have been written to generate switch requests as fast as possible, and making sure that the CPU is not the bottleneck at generating requests. Each script emits 100,000 requests to the controller sequentially, and measures the time delta to issue those requests.

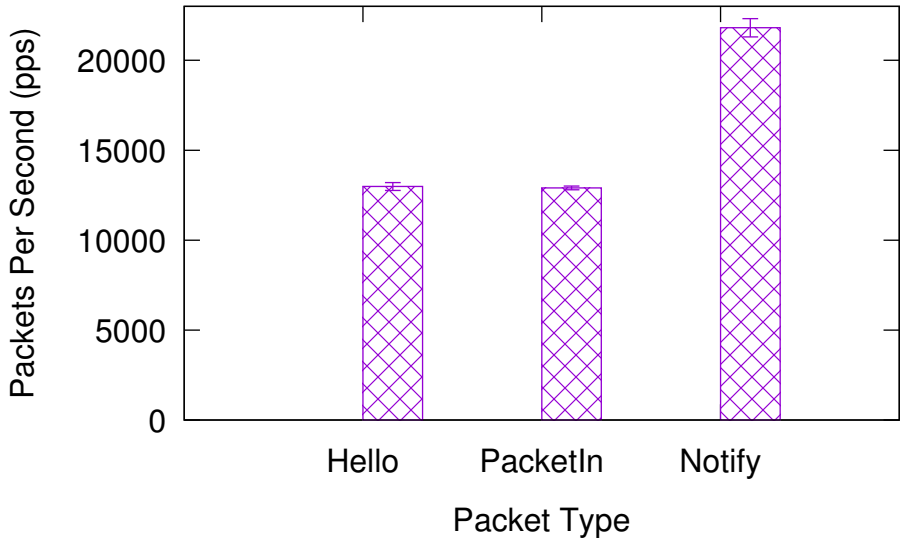
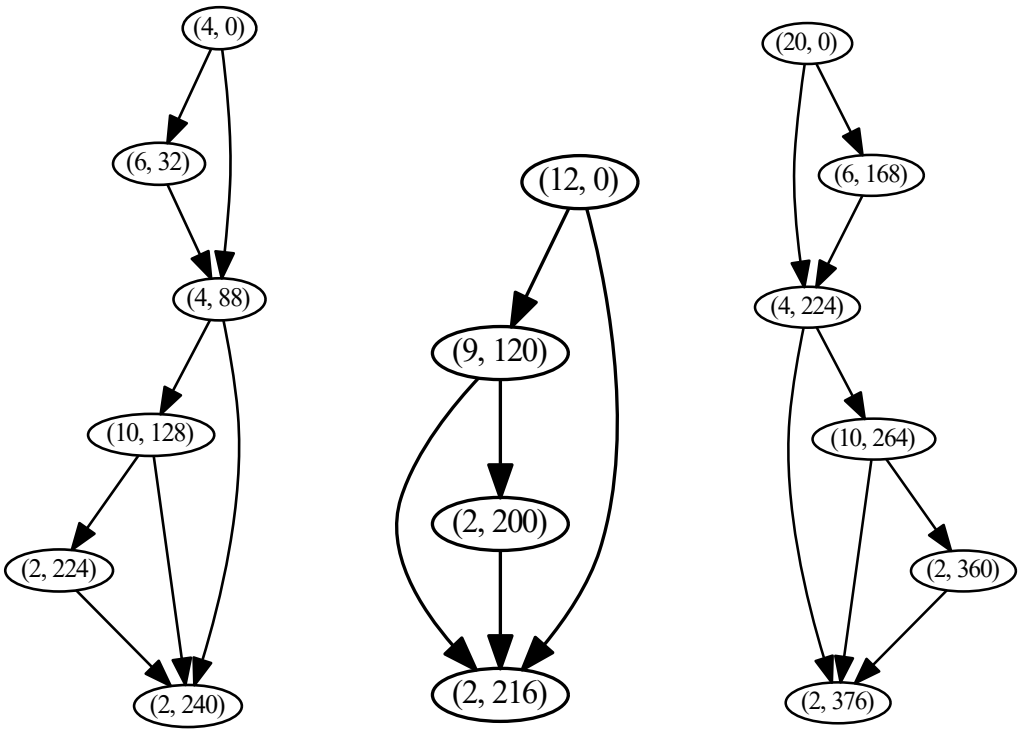


Figure 4.23: BPFabric controller performance in handling incoming requests

Figure 4.23 highlights the mean performances of the controller for the three packet types over 30 consecutive runs. Both *Hello* and *PacketIn* are processed at around 12,000 requests per second and *Notify* slightly below 22,000 requests per second. The performance of BPFabric is comparable to Python-based OpenFlow controllers such as Ryu and Pox [145]. However, in a BPFabric environment the number of switch-to-controller messages can be kept much lower than in OpenFlow as most node local processing and decisions can be done by the specific data plane functions. Considering the simplicity of the controller implementation and the relatively high performance achievable, this implementation of the BPFabric controller is perfectly suitable to demonstrate and experiment with the data plane programmable framework presented in this chapter.



(a) Learning Switch *(b) Learning Switch Centralised* *(c) Packet Size Distribution*

Figure 4.24: Acyclic Control Flow Graph and Complexity of data plane functions. Each tuple represents (# instructions, offset) per node

4.6.3 Program Complexity

So far the experiments presented have evaluated the performance of BPFabric controller and switches implementations for the set of experimental data plane functions provided. In this section the cost and complexity of data plane functions is evaluated with respect to the complexity of their flow execution graph. Using static analysis, bounds on the execution time and execution cost can be determined. This analysis can be performed either by the controller before issuing an install request or by the switch itself when receiving the request from the controller. The complexity and execution time of a data plane function can be determined by the depth of its critical path. The critical path is defined in computer architecture as the longest possible valid execution path of sequential operations. Therefore, using the critical path of a program, it is possible to determine the maximum number of instructions that can be executed by a specific function. Once the number of instructions are known, determining the execution time is relatively simple, especially considering the simple execution of eBPF.

The critical path of a specific program can be determined by keeping track of the branch operations and the jump offset associated. Once the execution tree with all the branches is constructed, the critical path is the longest path through the flow graph.

Figure 4.24 shows the acyclic control flow graphs (aCFGs) of some of the example data plane functions discussed in this chapter. These aCFGs have been generated dynamically from the eBPF ELFs by a python static analysis tool. Each node within the aCFG represents a sequential non-branching instruction block and the edges are transitions between blocks caused by jumps. In the simplest data plane implementations, each block can be associated with the parsing of one header field in the packet header, with the final branch operation depending on the value of the header field. The label for each node is a tuple containing the number of instructions in this block and the offset within the compiled eBPF program. From these aCFGs, it can be seen that the functions implemented have a critical path as long as the number of instructions in the program (i.e. in the worst case all the instructions in the program are executed). In more complex data plane functions performing different routing and forwarding operations depending on protocols headers will result in more conditional branching resulting in a critical path shorter than the number of instructions. However, the proposed functions have been kept simple to demonstrate BPFabric and only perform a single function resulting in a critical path as long as the number of instructions.

Further static analysis can be performed on the compiled data plane functions to determine the execution complexity. As discussed in the previous section, table operations can negatively impact the packet processing performance, especially for software switch implementations. Therefore, a software switch implementation could statically analyse the eBPF programs when received from the controller to dermine the number of table operations. Using the number of table operations and the number of instructions in the program, the switch can compute the worst-case execution time. Relying on this calculated execution time, the switch can notify the controller if sub-line-rate performance is to be expected. The same approach could be considered for hardware implementations in which multiply, divide and modulo operations might take multiple clock cycles.

4.7 Summary

OpenFlow, through its vendor-agnostic protocol and open implementation, has been able to show the benefits of SDN by centrally controlling and managing network devices. However, in its current implementation, the programmability is limited, providing a fixed and limited set of headers fields, actions, matching primitives, and a very rigid processing pipeline. With this approach, the control plane cannot define the packet processing logic, how and which header fields should be extracted and matched, preventing the support for new network protocols, statistics, monitoring, routing and other lightweight functions.

In this chapter, BPFabric has been proposed as an architecture that allows the control plane to specify the data plane behaviour of the switches on-the-fly as well as to query and manipulate the network state directly. Using a platform and protocol independent instruction set, the data plane behaviour can be expressed independently of the target device and without constraints on the HLL used while providing performance and safety guarantees.

Using a wide range of example use-cases, this work has shown how current and new data plane functionality can be designed to improve routing and forwarding, provide insight into the network behaviour, and the ability to implement lightweight functions. Through a simple but powerful southbound API, BPFabric demonstrates how the controller can listen for network events and maintain a fine-grain global view of the network state. Via two implementations, one for large-scale experimentation setup based on Mininet, and a high performance switch using Intel DPDK, the benefits and performance of the proposed framework have been evaluated. The experiments highlight that the performance of BPFabric is on par with today's static implementations while providing more flexbility to the network operator and offering new insights into the network characteristics than what is currently available with OpenFlow.

Chapter 5

Conclusion and Future Work

5.1 Overview

This chapter summarises and concludes this work, highlighting the contributions made, revisting the original thesis statement, discussing directions for potential future work, and listing the research outcomes. Overall, this research has been able to successfully meet the objectives presented in the opening chapter. The remainder of this chapter is structured as follows: Section 5.2 details the contributions made in this dissertation, and Section 5.3 revisits the original thesis statement and details how the objectives have been met. Finally, future research areas and improvements are detailed in Section 5.4, and Section 5.5 concludes this dissertation.

5.2 Contributions

This work has examined the benefits of centralised orchestration and management to improve resource utilisation of large-scale infrastructures. Through a novel framework for data plane programmability the operators are able to collect a wide range of metrics over varying timescales, from sub-millisecond packet interarrival time to yearly topological changes. This new insight and control on the operation of the infrastructure allows the providers to

better tune services and applications to the operating environment, and consequently improve the overall resource utilisation. This improved resource utilisation and orchestration could improve the RoI by reducing CAPEX and OPEX as discussed by Greenberg et al. in their analysis of the cost of data centre operation [60]. Through the benefits of central network resource management on performance tuning, this work has demonstrated significant improvement for TCP, successfully mitigating TCP incast throughput collapse, as well as a wide range of routing, forwarding, telemetry, and anomaly detection functions. By extending the paradigm of SDN beyond its current realisations, this work demonstrates how next generation programmable networks should be designed. It is unlikely for network operators to radically change how the infrastructure is managed and orchestrated, and a pragmatic approach is required to demonstrate the benefits and deployability of the proposed framework. To that end, this work demonstrates the design of the data plane programmability from the switch design both in hardware and software to allow deployment in dedicated ASICs in the long term, and in software for experimentation and overlay networks in the short term. Overall, the framework presented aims to pragmatically provide data plane programmability in modern network deployments in order to benefit the infrastructure operators and the end-users. The infrastructure operator can leverage the programmability to simplify and improve the management and orchestration, while the end-users can gather deeper insight into the operation of the network. The work described in this thesis contributes to the abstraction, development, and paradigm shift of next-generation SDN networks through:

- A critical and in-depth review of current widely deployed DC network topologies and associated limitations that results in low resource utilisation and high deployment and maintenance cost. These current limitations have been the driving motivation for SDN architectures providing management, control, and programmability to the operators.

- An analysis of TCP throughput incast collapse in a realistic hardware platform (NetFPGA), highlighting the issue of a mismatch between the congestion control parameters and the operating environment in large-scale DCs.

- The design and implementation of the OTCP controller, agent, and measurement al-

gorithms, demonstrating the benefits and feasibility of centralised parameter tuning based on a full view of the network operating conditions.

- A novel data plane programmable framework, providing platform, protocol, and language independent packet processing, addressing the limitations of today's legacy network and current SDN realisations.

- The use of the eBPF instruction set as the processing pipeline instruction set for protocol and platform independence, and the introduction of a simple and portable control plane API.

- Two software implementations of the proposed design, providing the necessary tools for large-scale emulation in Mininet as well as demonstrating the feasibility and performance achievable using commodity hardware.

- A proof-of-concept hardware implementation for FPGA-accelerated packet processing, and an evaluation of the execution complexity and performance trade-off to be considered in hardware implementations.

- Numerous data plane functions providing insight in the network operation that are irrealisable or impractictal in today's SDN implementations. The functions described range from packet forwarding to telemetry, debugging, and other middlebox-like functions such as anomaly detection.

5.3 Thesis Statement Revisited

In this section, the thesis statement presented at the beginning of this thesis is reiterated and reviewed, highlighting the ways in which it has been addressed.

The deployment of a programmable data plane in data centre SDN infrastructures will enable operators to improve network utilisation by dynamically provisionning the network parameters based on temporal demand.

This thesis initially presented the current problem space of very large-scale network infrastructures. It has presented the current management and orchestration limitations of such infrastructures and the difficulties and limitations that network operators are facing. To address these limitations, network operators have been transitionning towards more programmable infrastructures, but such migration is complex and this thesis describes in length these difficulties and the roadmap over the last two decades to obtain today's Software Defined Networks. To put this complexity in perspective and clearly demonstrate the problems that DC operators have been facing, this work has discussed the different network topologies that have been deployed, considering their scale, complexity, limitations, and reliability. It then discussed how this limited management is detrimental to the operation of such large-scale networks and how DC network operators are in a unique position, from their single ownership, to leverage this programmability to perform centrally informed DC-wide decisions.

A thorough investigation and evaluation of the TCP thoughput incast collapse has shown that the gross underutilisation of network resources in high-throughput, low-latency environments is associated to the mismatch between the congestion control parameters and the network characteristics. Using the NetFPGA hardware platform, this work has demonstrated this mismatch between the default congestion control parameters of TCP and the typical network infrastructure of large-scale DCs. Through extensive evaluation, this work demonstrated that TCP flow completion time and goodput can be improved by an order of magnitude if the retransmission timeouts and congestion window size are modified to match the latency and bandwidth available in the network fabric.

Using the OpenFlow API to manage and control the network, this work introduces Omniscient TCP (OTCP) as a means to centrally collect and compute the congestion control parameters based on known or discoverable network characteristics. OTCP is able to dynamically compute bounds on the retransmission timers and the congestion window, and propagate these computed values to the end-hosts in order to adapt the congestion control mechanism to the changing network characteristics. Using the OpenFlow API, a dedicated controller application can discover the network topology, the latency of individual links, and

the maximum bandwidth associated to each link. Using these metrics, the controller can then compute route-specific parameters and, through a northbound REST API, propagate them to the relevant end-host. OTCP demonstrates that, with a complete view of the network infrastructure, the applications and services can be tuned dynamically in response to network changes, resulting in significant improvement in resource utilisation. In the presented evaluations, OTCP is able to significantly outperform TCP for short-lived, soft-realtime flows with a 12× improvement in flow completion time at the mean and 31× at the 95th percentile.

Omniscient TCP demonstrates the benefits of centrally informed decision-making for better using the resources but also highlights the significant limitations of today's SDN realisations. Through an extensive discussion, this research describes the current limitations of OpenFlow in its ability to match packets and provide insight into the network behaviour. These limitations are directly associated to the pragmatic initial design of OpenFlow and simply highlight the new requirements for the next generation of SDN frameworks. In order to dynamically provision the network resources, network programmability must evolve, allowing network operators to measure and collect metrics on-demand based on particular needs.

To that end, this work introduces BPFabric a novel SDN framework that addresses the limitations of today's SDN realisations. Using BPFabric, network operators can create and deploy new switch data plane functions dynamically that are able to measure, compute, and report metrics of interest as well as provide lightweight functions for custom routing, forwarding, telemetry and even middlebox-like packet processing functions. Through this framework the management and orchestration of very large-scale networks is greatly simplified by allowing a single logically-centralised controller to have complete control over the network. Individual network devices can be dynamically reconfigured to reflect temporal changes in the environment and allow the network operator to collect metrics of interest to continuously adapt applications and services to the operating environment.

Leveraging the eBPF instruction set, this work has demonstrated how platform, protocol, and language independent data plane functions can be created by the network operator to perform arbitrary packet processing functions in the network devices. It also introduces a

simple southbound API allowing the network devices to communicate with the controller and provide the mechanisms for dynamically deploying new data plane functions, reporting events of interests, and providing a complete view of the network state. This work provides two implementations, one for a large-scale emulation framework designed for testing and development, and a second one as a high-performance production-grade software switch for overlay network deployment. Through evaluations, it has been shown that the proposed framework can achieve higher performance than today's state of the art SDN realisations while providing significantly higher control and insight over the network.

5.4 Future Work

5.4.1 Further use-cases

In this thesis, the work presented has been focussed on TCP throughput incast collapse and the benefits of centralised network resource management to improve network utilisation. Many other services and applications at different layers of the stack could benefit from fine-tuning at runtime. For instance, VMs have been widely deployed in DCs to better utilise the hardware resources available, but their placement over the infrastructure is often static, poorly using the network.

Alongside the work presented in this thesis, the impact of static allocation was investigated in "SDN-based virtual machine management for cloud data centers" [146] and demonstrates the benefits of centralised VM management. It would be interesting to demonstrate the benefits of the proposed data plane programmable framework for VM allocation and placement based on the very fine-grain view of the network resources.

Furthermore, the same approach could be beneficial to Network Function Virtualization, Smart Caching, and Protocol Offloading. NFV has gained significant popularity since OpenFlow has allowed Virtual Network Functions (VNFs) to be deployed on any commodity server. Using BPFabric, it is believed that the implementation, deployment, and chaining of

VNFs can be simplified by relying on the existing packet processing and forwarding mechanisms as well as the centralised management and orchestration. Smart caches have been relying on deep packet inspection and network characteristics to optimise the data to be cached which could be provided or improved by BPFabric. Finally, protocol offloading in the end-hosts has been widely available for TCP and UDP, but with the deployment of protocols such as TLS, DASH, and many more, the load on the end-hosts is becoming more and more significant, therefore it would be interesting to see how data plane programmability could be extended to the NIC in order to offload protocols. With the acquisition of Altera by Intel in 2016 and the next generation of CPUs to include a on-chip FPGA, the deployment of data-plane programmable functions to the end-hosts is promising and likely to be ubiquitous in only a few years [147, 148].

5.4.2 Hardware Implementation

The hardware implementation of BPFabric presented in this work provides insight on the design and packet processing performance achievable. However, in order to compare the performance of the proposed design to existing OpenFlow hardware switches, further work on this implementation is necessary. Hardware platforms such as the NetFPGA would be suitable to perform experiments, but the design and implementation is an extremely complex and time consuming task. Another limitation is the low port density of most hardware-accelerated NICs like NetFPGAs or SolarFlare Application Onload Engine, and the prohibitive cost of FPGA enabled high density switches like Metamako MetaMux or Arista 7124FX. However, once the benefits of the proposed approach prototyped in an FPGA, it is imaginable to see an ASIC implementation providing higher performance at a cheaper production cost.

The current Verilog implementation assumes that each complete packet is stored in memory and accessible by the eBPF core. However, typical forwarding pipelines such as the one provided for the NetFPGA assume that the packet will be streamed in fix-sized chunks following the AXI4 protocol. This approach allows non-blocking switches to be designed easily

as chunks are streamed until a decision is made and therefore a forwarding decision can be made before the entire packet is received. However, most SDN switches are blocking as they require the protocol headers to be available to perform lookups in the table. Simply adding a buffering stage from the AXI4 stream into some available memory is complex without a deep understanding of the streaming protocol, the queueing mechanisms and the different memory modules available, with each including different trade-off in throughput and latency. Additional constraints and issues must also be addressed, such as the multiplication and divide operations, the interrupt handlers to perform the different *call* operations or the possible memory consistency issues when collocating multiple eBPF cores on the same chip. Hence, developing a fully capable hardware implementation addressing all the potential limitations and open-issues requires a significant amount of research and implementation.

5.4.3 Formal Verification

Formal verification is the process of proving correctness of a system through its formal mathematical representation. By using eBPF as the instruction set, the execution graph can be decomposed into an acyclic control flow graph opening many intersting aspects for formal verification. In this thesis, the example verification proposed has been to avoid unexpected memory accesses and compute bounds on the function execution time. However, many other interesting aspects could be investigated, such as the instruction reordering, probabilistic branching estimation or model checking. By reordering the instructions to parse the layers in order, the code could be executed as the packet is received allowing the packet stream to be used directly and consequently resulting in a non-blocking switch. However, the feasibility of such reordering and the resulting complexity of the reordered program should be investigated futher.

5.5 Concluding Remarks

The work described in this dissertation opens a wide range of new research directions, from hardware design to application and service optimisation to data plane function design, to formal verification. Each of these aspects have the potential to benefit DC infrastructures and other large-scale networks, by improving resource utilisation, simplifying management and orchestration, and increasing reliability. As a consequence of any of these improvements, the RoI can be improved either by providing cheaper or better service to the end-users or by reducing the OPEX and CAPEX. Some of the most notable research directions envisaged during this work have been presented, and it is certain that many other opportunities have been omitted. Overall the work demonstrated in this thesis demonstrates the strong benefits of programmability and fine-grained resource management for the management and orchestration of large-scale network infrastructures, and proposes a novel approach to achieve such programmability for next generation networks.

5.6 Publications

The work reported on this thesis has led to following publications in the domains of resource management, Software Defined Networking, Data plane programmability and Network Function Vitualization (NFV).

- Simon Jouet and Dimitrios P. Pezaros. "BPFabric: Data Plane Programmability for Software Defined Networks". In: *Proceedings of the Symposium on Architectures for Networking and Communications Systems*. ANCS '17. Beijing, China: IEEE Press, 2017, pp. 38–48. ISBN: 978-1-5090-6386-4. DOI: 10.1109/ANCS.2017.14. URL: https://doi.org/10.1109/ANCS.2017.14

- Dimitrios P Pezaros, Richard Cziva, and Simon Jouet. "SDN for Cloud Data Centres". In: *IET Big-Data and Software Defined Networks*. IET Book Series on Big Data. IET, Sept. 2017

- Abeer Ali et al. “SDNFV-based DDoS detection and remediation in multi-tenant, virtualised infrastructures”. In: *Guide to Security in SDN and NFV - Challenges, Opportunities, and Applications*. GSSNOA ’16. Springer, 2017

- M. Iordache et al. “Distributed, multi-level network anomaly detection for datacentre networks”. In: *2017 IEEE International Conference on Communications (ICC)*. May 2017, pp. 1–6. DOI: 10.1109/ICC.2017.7996569

- A. Pamukchiev, S. Jouet, and D. P. Pezaros. “Distributed network anomaly detection on an event processing framework”. In: *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*. Jan. 2017, pp. 659–664. DOI: 10.1109/CCNC.2017.7983209 - **Best paper runner-up**

- R. Cziva et al. “SDN-Based Virtual Machine Management for Cloud Data Centers”. In: *IEEE Transactions on Network and Service Management* 13.2 (June 2016), pp. 212–225. ISSN: 1932-4537. DOI: 10.1109/TNSM.2016.2528220

- C. Mas Machuca et al. “Technology-related disasters: A survey towards disaster-resilient Software Defined Networks”. In: *2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM)*. Sept. 2016, pp. 35–42. DOI: 10.1109/RNDM.2016.7608265

- T. Gomes et al. “A survey of strategies for communication networks to protect against large-scale natural disasters”. In: *2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM)*. Sept. 2016, pp. 11–22. DOI: 10.1109/RNDM.2016.7608263

- Richard Cziva, Simon Jouet, and Dimitrios P Pezaros. “Roaming Edge vNFs Using Glasgow Network Functions”. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. Proceedings of the 2016 SIGCOMM Conference. Florianopolis, Brazil: ACM, 2016, pp. 601–602. ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2959067. URL: http://doi.acm.org/10.1145/2934872.2959067

- Fung Po Tso, Simon Jouet, and Dimitrios P Pezaros. "Network and server resource management strategies for data centre infrastructures: A survey". In: *Computer Networks* 106 (2016), pp. 209–225. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2016.07.002. URL: http://www.sciencedirect.com/science/article/pii/S1389128616302298

- S. Jouet, C. Perkins, and D. Pezaros. "OTCP: SDN-managed congestion control for data center networks". In: *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. Apr. 2016, pp. 171–179. DOI: 10.1109/NOMS.2016.7502810

- S. Jouet, R. Cziva, and D. P. Pezaros. "Arbitrary packet matching in OpenFlow". In: *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*. July 2015, pp. 1–6. DOI: 10.1109/HPSR.2015.7483106 - **Best paper award**

- R. Cziva, S. Jouet, and D. P. Pezaros. "GNFC: Towards network function cloudification". In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. Nov. 2015, pp. 142–148. DOI: 10.1109/NFV-SDN.2015.7387419

- R. Cziva et al. "Container-based network function virtualization for software-defined networks". In: *2015 IEEE Symposium on Computers and Communication (ISCC)*. July 2015, pp. 415–420. DOI: 10.1109/ISCC.2015.7405550

- F. P. Tso et al. "The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infrastructures". In: *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*. July 2013, pp. 108–112. DOI: 10.1109/ICDCSW.2013.25

- S. Jouet and D. P. Pezaros. "Measurement-based TCP parameter tuning in cloud data centers". In: *2013 21st IEEE International Conference on Network Protocols (ICNP)*. Oct. 2013, pp. 1–3. DOI: 10.1109/ICNP.2013.6733644

Appendices

Appendix A

BPFabric Use Cases

A.1 Centralised Learning Switch

A.1.1 Data Plane definition

```
#include <linux/if_ether.h>
#include "ebpf_switch.h"

struct bpf_map_def SEC("maps") inports = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = 6,
    .value_size = sizeof(uint32_t),
    .max_entries = 256,
};

uint64_t prog(struct packet *pkt)
{
    uint32_t *port;

    // If the packet src mac is unknown, tell the controller
    if (bpf_map_lookup_elem(&inports, pkt->eth.h_source, &port) == -1) {
        return CONTROLLER;
    }

    // Lookup the output port
    if (bpf_map_lookup_elem(&inports, pkt->eth.h_dest, &port) == -1) {
        // If no entry was found send to the controller
```

```
        return CONTROLLER;
    }

    return *port;
}
char _license[] SEC("license") = "GPL";
```

A.1.2 Control Plane logic

```
#!/usr/bin/env python
import struct

# Import BPFabric core classes
from core import eBPFCoreApplication, set_event_handler, FLOOD
from core.packets import *

# Simple Switch implementation
class SimpleSwitchApplication(eBPFCoreApplication):
    """Automatically install learning switch function on switch connection"""
    @set_event_handler(Header.HELLO)
    def hello(self, connection, pkt):
        self.mac_to_port = {}

        with open('../examples/learningswitch_centralised.o', 'rb') as f:
            print("Installing the eBPF ELF")
            connection.send(InstallRequest(elf=f.read()))

    """L2 Learning Switch behaviour"""
    @set_event_handler(Header.PACKET_IN)
    def packet_in(self, connection, pkt):
        metadatahdr_fmt = 'I10x'
        ethhdr_fmt = '>6s6sH'

        # Extract Ethernet layer information from the packet
        in_port, = struct.unpack_from(metadatahdr_fmt, pkt.data, 0)
        eth_dst, eth_src, eth_type = struct.unpack_from(ethhdr_fmt, \
            pkt.data, struct.calcsize(metadatahdr_fmt))

        self.mac_to_port.setdefault(connection.dpid, {})

        # Only perform L2 learning for (src) unicast packets
        if ord(eth_src[0]) & 1 == 0:
            # Learn the mapping between the MAC and port for this switch
            self.mac_to_port[connection.dpid][eth_src] = in_port
```

```
            # Update the switch' table entry
            connection.send(TableEntryInsertRequest(
                table_name="inports",
                key=eth_src,
                value=struct.pack('I', in_port)))

        # If the destination is unicast send to the relevant port, otherwise flood
        if ord(eth_dst[0]) & 1 == 1:
            out_port = FLOOD
        else:
            out_port = self.mac_to_port[connection.dpid].get(eth_dst, FLOOD)

        # Send back the packet that got in on the relevant port
        connection.send(PacketOut(data=pkt.data, out_port=out_port))

if __name__ == '__main__':
    SimpleSwitchApplication().run()
```

A.2 TCP Latency Measurement

```
#include <linux/if_ether.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include "ebpf_switch.h"

// Key for the latency
struct tcpflowtuple {
    uint32_t src;
    uint32_t dst;
    uint16_t srcport;
    uint16_t dstport;
};

struct tstamp {
    uint32_t sec;
    uint32_t nsec;
};

// Timestamps of the three-way handshake packets
struct tcplatency {
    struct tstamp syn;
    struct tstamp synack;
```

```
    struct tstamp ack;
};

// Store the tuple to timestamps mapping
struct bpf_map_def SEC("maps") latency = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(struct tcpflowtuple), // key is SRC:DST:SRCPORT:DSTPORT tuple
    .value_size = sizeof(struct tcplatency), // key is sec:nsec
    .max_entries = 256,
};

uint64_t prog(struct packet *pkt)
{
    // Check if the ethernet frame contains an ipv4 payload
    if (pkt->eth.h_proto == 0x0008) {
        // Check if the ip packet contains a TCP payload
        if (ipv4->ip_p == 6) {
            struct tcphdr *tcp = (struct tcphdr *)(((uint32_t *)ipv4) + ipv4->ip_hl);

            if ((tcp->th_flags & (TH_ACK | TH_SYN)) == TH_SYN) {
                // TCP SYN
                struct tcpflowtuple tuple = {
                    .src = ipv4->ip_src.s_addr,
                    .dst = ipv4->ip_dst.s_addr,
                    .srcport = tcp->th_sport,
                    .dstport = tcp->th_dport
                };

                // Initialize the timestamps
                struct tcplatency lat = {
                    .syn = { .sec = pkt->metadata.sec, .nsec = pkt->metadata.nsec },
                    .synack = 0,
                    .ack = 0
                };

                // Create the entry in the table
                bpf_map_update_elem(&latency, &tuple, &lat, 0);
            } else if ((tcp->th_flags & (TH_ACK | TH_SYN)) == (TH_SYN | TH_ACK)) {
                // TCP SYN|ACK
                struct tcpflowtuple tuple = {
                    .dst = ipv4->ip_src.s_addr,
                    .src = ipv4->ip_dst.s_addr,
                    .dstport = tcp->th_sport,
                    .srcport = tcp->th_dport
```

```
                };
                struct tcplatency* lat;

                // Find the matching entry for this tuple and update for synack
                if (bpf_map_lookup_elem(&latency, &tuple, &lat) != -1) {
                    lat->synack.sec = pkt->metadata.sec;
                    lat->synack.nsec = pkt->metadata.nsec;
                }
            } else if ((tcp->th_flags & TH_ACK) == TH_ACK) {
                // TCP ACK
                struct tcpflowtuple tuple = {
                    .src = ipv4->ip_src.s_addr,
                    .dst = ipv4->ip_dst.s_addr,
                    .srcport = tcp->th_sport,
                    .dstport = tcp->th_dport
                };
                struct tcplatency* lat;

                // Find the matching entry for this tuple and update for ack
                if (bpf_map_lookup_elem(&latency, &tuple, &lat) != -1) {
                    lat->ack.sec = pkt->metadata.sec;
                    lat->ack.nsec = pkt->metadata.nsec;

                    // Notify the controller of the measured latencies
                    bpf_notify(1,
                        ((uint8_t *)lat) - sizeof(struct tcpflowtuple) - 4,
                        sizeof(struct tcplatency) + sizeof(struct tcpflowtuple) + 4);

                    // Remove the entry
                    bpf_map_delete_elem(&latency, &tuple);
                }
            }
        }
    }

    // learning switch behaviour
}

char _license[] SEC("license") = "GPL";
```

A.3 TCP Flow Arrival

```
#include <linux/if_ether.h>
```

```
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include "ebpf_switch.h"

// Statistics to keep track of flow arrival and departure
struct arrival_stats {
    uint32_t lasttime;
    uint32_t arrival;
    uint32_t departure;
};

// Global storage map for the statistics
struct bpf_map_def SEC("maps") flowarrival = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(unsigned int),
    .value_size = sizeof(struct arrival_stats),
    .max_entries = 1,
};

uint64_t prog(struct packet *pkt)
{
    // Check if the ethernet frame contains an ipv4 payload
    if (pkt->eth.h_proto == 0x0008) {
        struct ip *ipv4 = (struct ip *)(((uint8_t *)&pkt->eth) + ETH_HLEN);

        // Check if the ip packet contains a TCP payload
        if (ipv4->ip_p == 6) {
            struct tcphdr *tcp = (struct tcphdr *)(((uint32_t *)ipv4) + ipv4->ip_hl);

            if (tcp->th_flags & (TH_SYN | TH_FIN)) {
                struct arrival_stats *astats;

                // Get the current flow arrival statistics
                unsigned int key = 0;
                bpf_map_lookup_elem(&flowarrival, &key, &astats);

                if (tcp->th_flags & TH_SYN) {
                    // SYN flag, increase arrival rate
                    astats->arrival += 1;
                } else if (tcp->th_flags & TH_FIN) {
                    // FIN flag, increase departure rate
                    astats->departure += 1;
                } else if (tcp->th_flags & TH_RST) {
                    // RST flag, increase departure rate (connection denied)
```

```
                    astats->departure += 1;
                }

                // If the last packet was more than 5 seconds ago
                if (pkt->metadata.sec - astats->lasttime > 5) {
                    // Notify the controller of the current statistics
                    bpf_notify(0, astats, sizeof(struct arrival_stats));

                    // Reset the counters
                    astats->lasttime = pkt->metadata.sec;
                    astats->arrival = 0;
                    astats->departure = 0;
                }
            }
        }
    }

    // learning switch behaviour
}

char _license[] SEC("license") = "GPL";
```

Bibliography

[1] N. Farrington and A. Andreyev. “Facebook’s data center network architecture”. In: *2013 Optical Interconnects Conference*. May 2013, pp. 49–50. DOI: 10.1109/OIC.2013.6552917.

[2] Alexey Andreyev. *Introducing data center fabric, the next-generation Facebook data center network*. [Online; accessed 27-09-2017]. Nov. 2014. URL: https://code.facebook.com/posts/360346274145943/.

[3] D. L. Tennenhouse et al. “A survey of active network research”. In: *IEEE Communications Magazine* 35.1 (Jan. 1997), pp. 80–86. ISSN: 0163-6804. DOI: 10.1109/35.568214.

[4] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse. “ANTS: a toolkit for building and dynamically deploying network protocols”. In: *1998 IEEE Open Architectures and Network Programming*. Apr. 1998, pp. 117–129. DOI: 10.1109/OPNARC.1998.662048.

[5] D. S. Alexander et al. “The SwitchWare Active Network Architecture”. In: *Netwrk. Mag. of Global Internetwkg.* 12.3 (May 1998), pp. 29–36. ISSN: 0890-8044. DOI: 10.1109/65.690959. URL: http://dx.doi.org/10.1109/65.690959.

[6] Beverly Schwartz et al. “Smart Packets: Applying Active Networks to Network Management”. In: *ACM Trans. Comput. Syst.* 18.1 (Feb. 2000), pp. 67–88. ISSN: 0734-2071. DOI: 10.1145/332799.332893. URL: http://doi.acm.org/10.1145/332799.332893.

[7] L. Yang et al. *Forwarding and Control Element Separation (ForCES) Framework*. RFC 3746 (Informational). Internet Engineering Task Force, Apr. 2004. URL: http://www.ietf.org/rfc/rfc3746.txt.

[8] Matthew Caesar et al. "Design and Implementation of a Routing Control Platform". In: *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 15–28. URL: http://dl.acm.org/citation.cfm?id=1251203.1251205.

[9] TV Lakshman et al. "The SoftRouter architecture". In: *Proceedings ACM SIGCOMM Workshop on Hot Topics in Networking (HOTNET '04)*. Vol. 2004. 2004, pp. 1–6.

[10] Martin Casado et al. "SANE: A Protection Architecture for Enterprise Networks". In: *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*. USENIX-SS'06. Vancouver, B.C., Canada: USENIX Association, 2006. URL: http://dl.acm.org/citation.cfm?id=1267336.1267346.

[11] Martin Casado et al. "Ethane: Taking Control of the Enterprise". In: *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM '07. Kyoto, Japan: ACM, 2007, pp. 1–12. ISBN: 978-1-59593-713-1. DOI: 10.1145/1282380.1282382. URL: http://doi.acm.org/10.1145/1282380.1282382.

[12] Hong Yan et al. "Tesseract: A 4D Network Control Plane". In: *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*. NSDI'07. Cambridge, MA: USENIX Association, 2007, pp. 27–27. URL: http://dl.acm.org/citation.cfm?id=1973430.1973457.

[13] Nick McKeown et al. "OpenFlow: Enabling Innovation in Campus Networks". In: *Proceedings of the 2009 SIGCOMM Conference* 38.2 (Mar. 2008), pp. 69–74. ISSN: 0146-4833. DOI: 10.1145/1355734.1355746. URL: http://doi.acm.org/10.1145/1355734.1355746.

[14] A. Doria et al. *Forwarding and Control Element Separation (ForCES) Protocol Specification*. RFC 5810 (Proposed Standard). Internet Engineering Task Force, Mar. 2010. URL: http://www.ietf.org/rfc/rfc5810.txt.

[15] Justin Pettit et al. "Virtual switching in an era of advanced edges". In: *2nd Workshop on Data Center-Converged and Virtual Ethernet Switching (DC CAVES)*. Sept. 2010.

[16] Pat Bosshart et al. "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN". In: *Proceedings of the 2013 SIGCOMM Conference*. SIGCOMM '13. Hong Kong, China: ACM, 2013, pp. 99–110. ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486011. URL: http://doi.acm.org/10.1145/2486001.2486011.

[17] Haoyu Song. "Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane". In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. HotSDN '13. Hong Kong, China: ACM, 2013, pp. 127–132. ISBN: 978-1-4503-2178-5. DOI: 10.1145/2491185.2491190. URL: http://doi.acm.org/10.1145/2491185.2491190.

[18] Pat Bosshart et al. "P4: Programming Protocol-independent Packet Processors". In: *Proceedings of the 2014 SIGCOMM Conference* 44.3 (July 2014), pp. 87–95. ISSN: 0146-4833. DOI: 10.1145/2656877.2656890. URL: http://doi.acm.org/10.1145/2656877.2656890.

[19] Vimalkumar Jeyakumar et al. "Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility". In: *SIGCOMM Computer Communication Review* 44.4 (Aug. 2014), pp. 3–14. ISSN: 0146-4833. DOI: 10.1145/2740070.2626292. URL: http://doi.acm.org/10.1145/2740070.2626292.

[20] S. Jouet, R. Cziva, and D. P. Pezaros. "Arbitrary packet matching in OpenFlow". In: *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*. July 2015, pp. 1–6. DOI: 10.1109/HPSR.2015.7483106.

[21] Muhammad Shahbaz et al. “PISCES: A Programmable, Protocol-Independent Software Switch”. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM ’16. Florianopolis, Brazil: ACM, 2016, pp. 525–538. ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934886. URL: http://doi.acm.org/10.1145/2934872.2934886.

[22] Simon Jouet and Dimitrios P. Pezaros. “BPFabric: Data Plane Programmability for Software Defined Networks”. In: *Proceedings of the Symposium on Architectures for Networking and Communications Systems*. ANCS ’17. Beijing, China: IEEE Press, 2017, pp. 38–48. ISBN: 978-1-5090-6386-4. DOI: 10.1109/ANCS.2017.14. URL: https://doi.org/10.1109/ANCS.2017.14.

[23] M. Handley. “Why the Internet Only Just Works”. In: *BT Technology Journal* 24.3 (July 2006), pp. 119–129. ISSN: 1358-3948. DOI: 10.1007/s10550-006-0084-z. URL: http://dx.doi.org/10.1007/s10550-006-0084-z.

[24] J. S. Turner and D. E. Taylor. “Diversifying the Internet”. In: *GLOBECOM ’05. IEEE Global Telecommunications Conference, 2005.* Vol. 2. Dec. 2005, pp. 760–766. DOI: 10.1109/GLOCOM.2005.1577741.

[25] Jakub Czyz et al. “Measuring IPv6 Adoption”. In: *Proceedings of the 2014 SIGCOMM Conference*. SIGCOMM ’14. Chicago, Illinois, USA: ACM, 2014, pp. 87–98. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2626295. URL: http://doi.acm.org/10.1145/2619239.2626295.

[26] Google. *Google IPv6 Adoption Statistics*. [Online; accessed 27-09-2017]. 2017. URL: https://www.google.com/intl/en/ipv6/statistics.html.

[27] Sangtae Ha, Injong Rhee, and Lisong Xu. “CUBIC: A New TCP-friendly High-speed TCP Variant”. In: *SIGOPS Operating Systems Review* 42.5 (July 2008), pp. 64–74. ISSN: 0163-5980. DOI: 10.1145/1400097.1400105. URL: http://doi.acm.org/10.1145/1400097.1400105.

[28] Saverio Mascolo et al. “TCP Westwood: Bandwidth Estimation for Enhanced Transport over Wireless Links”. In: *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*. MobiCom ’01. Rome, Italy: ACM, 2001, pp. 287–297. ISBN: 1-58113-422-3. DOI: 10.1145/381677.381704. URL: http://doi.acm.org/10.1145/381677.381704.

[29] Neal Cardwell et al. “BBR: Congestion-Based Congestion Control”. In: *ACM Queue* 14, September-October (2016), pp. 20–53. URL: http://queue.acm.org/detail.cfm?id=3022184.

[30] G. Papastergiou et al. “De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives”. In: *IEEE Communications Surveys Tutorials* 19.1 (2017), pp. 619–639. ISSN: 1553-877X. DOI: 10.1109/COMST.2016.2626780.

[31] Gregory Detal et al. “Revealing Middlebox Interference with Tracebox”. In: *Proceedings of the 2013 Conference on Internet Measurement Conference*. IMC ’13. Barcelona, Spain: ACM, 2013, pp. 1–8. ISBN: 978-1-4503-1953-9. DOI: 10.1145/2504730.2504757. URL: http://doi.acm.org/10.1145/2504730.2504757.

[32] K. J. Grinnemo et al. “Towards a flexible Internet transport layer architecture”. In: *2016 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*. June 2016, pp. 1–7. DOI: 10.1109/LANMAN.2016.7548846.

[33] Ryan Hamilton et al. *QUIC: A UDP-Based Secure and Reliable Transport for HTTP/2*. Internet-Draft draft-hamilton-early-deployment-quic-00. Work in Progress. Internet Engineering Task Force, July 2016. 36 pp. URL: https://datatracker.ietf.org/doc/html/draft-hamilton-early-deployment-quic-00.

[34] Mike Belshe and Roberto Peon. *SPDY Protocol*. Internet-Draft draft-mbelshe-httpbis-spdy-00. Work in Progress. Internet Engineering Task Force, Feb. 2012. 51 pp. URL: https://datatracker.ietf.org/doc/html/draft-mbelshe-httpbis-spdy-00.

[35] Mike Belshe, Roberto Peon, and Martin Thomson. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540 (Proposed Standard). Internet Engineering Task Force, May 2015. DOI: 10.17487/RFC7540. URL: http://www.ietf.org/rfc/rfc7540.txt.

[36] Poul-Henning Kamp. "HTTP/2.0 - The IETF is Phoning It In". In: *ACM Queue* 13.2 (Dec. 2014), 10:10–10:12. ISSN: 1542-7730. DOI: 10.1145/2732266.2716278. URL: http://doi.acm.org/10.1145/2732266.2716278.

[37] K. L. Calvert et al. "Directions in active networks". In: *IEEE Communications Magazine* 36.10 (Oct. 1998), pp. 72–78. ISSN: 0163-6804. DOI: 10.1109/35.722139.

[38] Samrat Bhattacharjee, Kenneth L. Calvert, and Ellen W. Zegura. "An Architecture for Active Networking". In: *Seventh International Conference on High Performance Networks (HPN '97)*. Ed. by Ahmed Tantawy. Boston, MA: Springer US, 1997, pp. 265–279. ISBN: 978-0-387-35279-4. DOI: 10.1007/978-0-387-35279-4_17. URL: http://dx.doi.org/10.1007/978-0-387-35279-4_17.

[39] T. Wolf and J. S. Turner. "Design issues for high-performance active routers". In: *IEEE Journal on Selected Areas in Communications* 19.3 (Mar. 2001), pp. 404–409. ISSN: 0733-8716. DOI: 10.1109/49.917702.

[40] D. S. Alexander et al. "A secure active network environment architecture: realization in SwitchWare". In: *IEEE Network* 12.3 (May 1998), pp. 37–45. ISSN: 0890-8044. DOI: 10.1109/65.690960.

[41] J. Van der Merwe et al. "Dynamic Connectivity Management with an Intelligent Route Service Control Point". In: *Proceedings of the 2006 SIGCOMM Workshop on Internet Network Management*. INM '06. Pisa, Italy: ACM, 2006, pp. 29–34. ISBN: 1-59593-570-3. DOI: 10.1145/1162638.1162643. URL: http://doi.acm.org/10.1145/1162638.1162643.

[42] Nick Feamster et al. "The Case for Separating Routing from Routers". In: *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*. FDNA '04. Portland, Oregon, USA: ACM, 2004, pp. 5–12. ISBN: 1-58113-

942-X. DOI: 10.1145/1016707.1016709. URL: http://doi.acm.org/10.1145/1016707.1016709.

[43] Patrick Verkaik et al. "Wresting Control from BGP: Scalable Fine-grained Route Control". In: *Proceedings of the USENIX Annual Technical Conference*. ATC'07. Santa Clara, CA: USENIX Association, 2007, 23:1–23:14. URL: http://dl.acm.org/citation.cfm?id=1364385.1364408.

[44] Steven Bellovin et al. *A Clean-Slate Design for the Next-Generation Secure Internet*. Pittsburgh, PA: Report for NSF Global Environment for Network Innovations (GENI) workshop. July 2005. URL: http://www.netsec.ethz.ch/publications/papers/bellovin_clark_perrig_song_nextGenInternet.pdf.

[45] Nick McKeown et al. *Stanford Clean Slate Program*. [Online; accessed 27-09-2017]. URL: http://cleanslate.stanford.edu/.

[46] Kalapriya Kannan and Subhasis Banerjee. "Compact TCAM: Flow Entry Compaction in TCAM for Power Aware SDN". In: *Proceedings of the 14th International Conference Distributed Computing and Networking (ICDCN 2013)*. Ed. by Davide Frey et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 439–444. ISBN: 978-3-642-35668-1. DOI: 10.1007/978-3-642-35668-1_32. URL: http://dx.doi.org/10.1007/978-3-642-35668-1_32.

[47] Sushant Jain et al. "B4: Experience with a Globally-deployed Software Defined Wan". In: *SIGCOMM Computer Communication Review* 43.4 (Aug. 2013), pp. 3–14. ISSN: 0146-4833. DOI: 10.1145/2534169.2486019. URL: http://doi.acm.org/10.1145/2534169.2486019.

[48] Teemu Koponen et al. "Network Virtualization in Multi-tenant Datacenters". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. NSDI'14. Seattle, WA: USENIX Association, 2014, pp. 203–216. ISBN: 978-1-931971-09-6. URL: http://dl.acm.org/citation.cfm?id=2616448.2616468.

[49] Teemu Koponen et al. “Onix: A Distributed Control Platform for Large-scale Production Networks”. In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. OSDI’10. Vancouver, BC, Canada: USENIX Association, 2010, pp. 351–364. URL: http://dl.acm.org/citation.cfm?id=1924943.1924968.

[50] Pankaj Berde et al. “ONOS: Towards an Open, Distributed SDN OS”. In: *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*. HotSDN ’14. Chicago, Illinois, USA: ACM, 2014, pp. 1–6. ISBN: 978-1-4503-2989-7. DOI: 10.1145/2620728.2620744. URL: http://doi.acm.org/10.1145/2620728.2620744.

[51] Gary Kinghorn Shashi Kiran. *Cisco Open Network Environment: Bring the Network Closer to Applications*. Tech. rep. [Online; accessed 27-09-2017]. Jan. 2014. URL: https://www.cisco.com/c/en/us/products/collateral/switches/nexus-1000v-switch-vmware-vsphere/white_paper_c11-728045.html.

[52] Juniper Networks. *Transforming to DevOps with Junos OS*. Tech. rep. [Online; accessed 27-09-2017]. 2015. URL: https://www.juniper.net/assets/fr/fr/local/pdf/whitepapers/2000586-en.pdf.

[53] M. Bjorklund. *YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. RFC 6020 (Proposed Standard). Internet Engineering Task Force, Oct. 2010. URL: http://www.ietf.org/rfc/rfc6020.txt.

[54] Intel. *Intel Ethernet Switch FM5000/FM6000*. [Online; accessed 27-09-2017]. Apr. 2014. URL: https://www.intel.co.uk/content/dam/www/public/us/en/documents/specification-updates/ethernet-switch-fm5000-fm6000-spec-update.pdf.

[55] Caviant. *XPliant: Ethernet Switch Product Family*. [Online; accessed 27-09-2017]. URL: http://www.cavium.com.

[56] Sam D. Hartman and Dacheng Zhang. *Security Requirements in the Software Defined Networking Model*. Internet-Draft draft-hartman-sdnsec-requirements-01. Work in Progress. Internet Engineering Task Force, Apr. 2013. 11 pp. URL: https://datatracker.ietf.org/doc/html/draft-hartman-sdnsec-requirements-01.

[57] Mohamed Boucadair and Christian Jacquenet. *Software-Defined Networking: A Perspective from within a Service Provider Environment*. RFC 7149 (Informational). Internet Engineering Task Force, Mar. 2014. URL: http://www.ietf.org/rfc/rfc7149.txt.

[58] Christopher S Yoo. "Cloud computing: Architectural and policy implications". In: *Review of Industrial Organization* 38.4 (2011), pp. 405–421. ISSN: 1573-7160. DOI: 10.1007/s11151-011-9295-7. URL: https://doi.org/10.1007/s11151-011-9295-7.

[59] Rich Miller. *Who has the Most Web Servers*. [Online; accessed 27-09-2017]. July 2013. URL: http://www.datacenterknowledge.com/archives/2009/05/14/whos-got-the-most-web-servers/.

[60] Albert Greenberg et al. "The Cost of a Cloud: Research Problems in Data Center Networks". In: *SIGCOMM Computer Communication Review* 39.1 (Dec. 2008), pp. 68–73. ISSN: 0146-4833. DOI: 10.1145/1496091.1496103. URL: http://doi.acm.org/10.1145/1496091.1496103.

[61] Cisco Systems. *Data Center: Load Balancing Data Center Services Solutions Reference Network Design*. [Online; accessed 27-09-2017]. Mar. 2004. URL: https://learningnetwork.cisco.com/docs/DOC-3438.

[62] Albert Greenberg et al. "VL2: A Scalable and Flexible Data Center Network". In: *Proceedings of the 2009 SIGCOMM Conference*. SIGCOMM '09. Barcelona, Spain: ACM, 2009, pp. 51–62. ISBN: 978-1-60558-594-9. DOI: 10.1145/1592568.1592576. URL: http://doi.acm.org/10.1145/1592568.1592576.

[63] Alexander Loukissas Mohammad Al-Fares and Amin Vahdat. “A Scalable, Commodity Data Center Network Architecture”. In: *Proceedings of the 2008 SIGCOMM Conference*. SIGCOMM ’08. Seattle, WA, USA: ACM, 2008, pp. 63–74. ISBN: 978-1-60558-175-0. DOI: 10.1145/1402958.1402967. URL: http://doi.acm.org/10.1145/1402958.1402967.

[64] D. Thaler and C. Hopps. *Multipath Issues in Unicast and Multicast Next-Hop Selection*. RFC 2991 (Informational). Internet Engineering Task Force, Nov. 2000. URL: http://www.ietf.org/rfc/rfc2991.txt.

[65] Rui Zhang-Shen and Nick McKeown. “Designing a Predictable Internet Backbone with Valiant Load-Balancing”. In: *Proceedings of the 13th International Workshop on Quality of Service – IWQoS 2005*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 178–192. ISBN: 978-3-540-31659-6. DOI: 10.1007/11499169_15. URL: http://dx.doi.org/10.1007/11499169_15.

[66] Soudeh Ghorbani et al. “Micro Load Balancing in Data Centers with DRILL”. In: *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*. HotNets-XIV. Philadelphia, PA, USA: ACM, 2015, 17:1–17:7. ISBN: 978-1-4503-4047-2. DOI: 10.1145/2834050.2834107. URL: http://doi.acm.org/10.1145/2834050.2834107.

[67] Arjun Singh et al. “Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google’s Datacenter Network”. In: *SIGCOMM Computer Communication Review* 45.4 (Aug. 2015), pp. 183–197. ISSN: 0146-4833. DOI: 10.1145/2829988.2787508. URL: http://doi.acm.org/10.1145/2829988.2787508.

[68] Chuanxiong Guo et al. “BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers”. In: *SIGCOMM Computer Communication Review* 39.4 (Aug. 2009), pp. 63–74. ISSN: 0146-4833. DOI: 10.1145/1594977.1592577. URL: http://doi.acm.org/10.1145/1594977.1592577.

[69] Chuanxiong Guo et al. “Dcell: A Scalable and Fault-tolerant Network Structure for Data Centers”. In: *SIGCOMM Computer Communication Review* 38.4 (Aug. 2008), pp. 75–86. ISSN: 0146-4833. DOI: 10.1145/1402946.1402968. URL: http://doi.acm.org/10.1145/1402946.1402968.

[70] Rodrigo De Couto et al. “Reliability and Survivability Analysis of Data Center Network Topologies”. In: *Journal of Network Systems Management* 24.2 (Apr. 2016), pp. 346–392. ISSN: 1064-7570. DOI: 10.1007/s10922-015-9354-8. URL: http://dx.doi.org/10.1007/s10922-015-9354-8.

[71] Arjun Singh 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.

[72] Rob Sherwood et al. “Flowvisor: A network virtualization layer”. In: *OpenFlow Switch Consortium, Tech. Rep* (2009). [Online; accessed 27-09-2017], pp. 1–13. URL: http://archive.openflow.org/downloads/technicalreports/openflow-tr-2009-1-flowvisor.pdf.

[73] Jorge Carapinha and Javier Jiménez. “Network Virtualization: A View from the Bottom”. In: *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures*. VISA ’09. Barcelona, Spain: ACM, 2009, pp. 73–80. ISBN: 978-1-60558-595-6. DOI: 10.1145/1592648.1592660.

[74] F. P. Tso et al. “Longer Is Better: Exploiting Path Diversity in Data Center Networks”. In: *2013 IEEE 33rd International Conference on Distributed Computing Systems*. July 2013, pp. 430–439. DOI: 10.1109/ICDCS.2013.36.

[75] F. P. Tso and D. P. Pezaros. “Improving Data Center Network Utilization Using Near-Optimal Traffic Engineering”. In: *IEEE Transactions on Parallel and Distributed Systems* 24.6 (June 2013), pp. 1139–1148. ISSN: 1045-9219. DOI: 10.1109/TPDS.2012.343.

[76] M. Zhang et al. “GreenTE: Power-aware traffic engineering”. In: *The 18th IEEE International Conference on Network Protocols*. Oct. 2010, pp. 21–30. DOI: 10.1109/ICNP.2010.5762751.

[77] D. Kliazovich, P. Bouvry, and S. U. Khan. “DENS: Data Center Energy-Efficient Network-Aware Scheduling”. In: *Green Computing and Communications (GreenCom), 2010 IEEE/ACM Int’l Conference on Int’l Conference on Cyber, Physical and Social Computing (CPSCom)*. Dec. 2010, pp. 69–75. DOI: 10.1109/GreenCom-CPSCom.2010.31.

[78] Peyman Kazemian et al. “Real Time Network Policy Checking Using Header Space Analysis”. In: *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX, 2013, pp. 99–111. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/kazemian.

[79] F. P. Tso et al. “Implementing Scalable, Network-Aware Virtual Machine Migration for Cloud Data Centers”. In: *2013 IEEE Sixth International Conference on Cloud Computing*. June 2013, pp. 557–564. DOI: 10.1109/CLOUD.2013.82.

[80] F. P. Tso et al. “Scalable Traffic-Aware Virtual Machine Management for Cloud Data Centers”. In: *2014 IEEE 34th International Conference on Distributed Computing Systems*. June 2014, pp. 238–247. DOI: 10.1109/ICDCS.2014.32.

[81] Richard Cziva et al. “SDN-based Virtual Machine management for Cloud Data Centers”. In: *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*. Oct. 2014, pp. 388–394. DOI: 10.1109/CloudNet.2014.6969026.

[82] R. Cziva, S. Jouet, and D. P. Pezaros. “GNFC: Towards network function cloudification”. In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. Nov. 2015, pp. 142–148. DOI: 10.1109/NFV-SDN.2015.7387419.

[83] Richard Cziva, Simon Jouet, and Dimitrios P Pezaros. "Roaming Edge vNFs Using Glasgow Network Functions". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 601–602. ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2959067. URL: http://doi.acm.org/10.1145/2934872.2959067.

[84] R. Cziva et al. "Container-based network function virtualization for software-defined networks". In: *2015 IEEE Symposium on Computers and Communication (ISCC)*. July 2015, pp. 415–420. DOI: 10.1109/ISCC.2015.7405550.

[85] T. Gomes et al. "A survey of strategies for communication networks to protect against large-scale natural disasters". In: *2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM)*. Sept. 2016, pp. 11–22. DOI: 10.1109/RNDM.2016.7608263.

[86] C. Mas Machuca et al. "Technology-related disasters: A survey towards disaster-resilient Software Defined Networks". In: *2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM)*. Sept. 2016, pp. 35–42. DOI: 10.1109/RNDM.2016.7608265.

[87] Shelly Cadora. *The limits of SNMP*. [Online; accessed 27-09-2017]. June 2016. URL: http://blogs.cisco.com/sp/the-limits-of-snmp.

[88] David Josephsen. *Building a Monitoring Infrastructure with Nagios*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2007. ISBN: 0132236931.

[89] S. Jouet and D. P. Pezaros. "Measurement-based TCP parameter tuning in cloud data centers". In: *2013 21st IEEE International Conference on Network Protocols (ICNP)*. Oct. 2013, pp. 1–3. DOI: 10.1109/ICNP.2013.6733644.

[90] S. Jouet, C. Perkins, and D. Pezaros. "OTCP: SDN-managed congestion control for data center networks". In: *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*. Apr. 2016, pp. 171–179. DOI: 10.1109/NOMS.2016.7502810.

[91] Brendan Gregg. “Linux 4.x Performance Using BPF Superpowers”. [Online; accessed 27-09-2017]. Feb. 2016. URL: http://www.brendangregg.com/blog/2016-03-05/linux-bpf-superpowers.html.

[92] Luis Andre Barroso, Jimmy Clidaras, and Urs Hoelzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2013. DOI: 10.2200/S00516ED2V01Y201306CAC024.

[93] Maltz David A. Benson Theophilus Akella Aditya. “Network Traffic Characteristics of Data Centers in the Wild”. In: *Proceedings of the 2013 SIGCOMM Internet Measurement Conference*. IMC ’10. Melbourne, Australia: ACM, 2010, pp. 267–280. ISBN: 978-1-4503-0483-2. DOI: 10.1145/1879141.1879175. URL: http://doi.acm.org/10.1145/1879141.1879175.

[94] Vijay Vasudevan et al. “Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication”. In: *Proceedings of the 2009 SIGCOMM Conference*. SIGCOMM ’09. Barcelona, Spain: ACM, 2009, pp. 303–314. ISBN: 978-1-60558-594-9. DOI: 10.1145/1592568.1592604. URL: http://doi.acm.org/10.1145/1592568.1592604.

[95] Mohammad Alizadeh et al. “Data center TCP (DCTCP)”. In: *SIGCOMM Computer Communication Review* 41.4 (Aug. 2010), pp. 63–74. ISSN: 0146-4833. URL: http://dl.acm.org/citation.cfm?id=2043164.1851192.

[96] Y. Peng et al. “Towards Comprehensive Traffic Forecasting in Cloud Computing: Design and Application”. In: *IEEE/ACM Transactions on Networking* 24.4 (Aug. 2016), pp. 2210–2222. ISSN: 1063-6692. DOI: 10.1109/TNET.2015.2458892.

[97] S. Joy and A. Nayak. “Improving flow completion time for short flows in datacenter networks”. In: *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. May 2015, pp. 700–705. DOI: 10.1109/INM.2015.7140358.

[98] Amar Phanishayee et al. “Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems”. In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. FAST’08. San Jose, California: USENIX Association, 2008, 12:1–12:14. URL: http://dl.acm.org/citation.cfm?id=1364813.1364825.

[99] Jiao Zhang et al. “Taming TCP incast throughput collapse in data center networks”. In: *2013 21st IEEE International Conference on Network Protocols (ICNP)*. Oct. 2013, pp. 1–10. DOI: 10.1109/ICNP.2013.6733609.

[100] W. Stevens. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. RFC 2001 (Proposed Standard). Internet Engineering Task Force, Jan. 1997. URL: http://www.ietf.org/rfc/rfc2001.txt.

[101] Y. Bernet et al. *A Framework for Integrated Services Operation over Diffserv Networks*. RFC 2998 (Informational). Internet Engineering Task Force, Nov. 2000. URL: http://www.ietf.org/rfc/rfc2998.txt.

[102] Yanpei Chen et al. “Understanding TCP Incast Throughput Collapse in Datacenter Networks”. In: *Workshop on Research on Enterprise Networking 2009*. WREN ’09. Barcelona, Spain: ACM, 2009, pp. 73–82. ISBN: 978-1-60558-443-0. DOI: 10.1145/1592681.1592693. URL: http://doi.acm.org/10.1145/1592681.1592693.

[103] J. Chu et al. *Increasing TCP’s Initial Window*. RFC 6928 (Experimental). Internet Engineering Task Force, Apr. 2013. URL: http://www.ietf.org/rfc/rfc6928.txt.

[104] M. Allman, S. Floyd, and C. Partridge. *Increasing TCP’s Initial Window*. RFC 3390 (Proposed Standard). Internet Engineering Task Force, Oct. 2002. URL: http://www.ietf.org/rfc/rfc3390.txt.

[105] S. Varma. *Internet Congestion Control*. Elsevier Science, 2015. ISBN: 9780128036006. URL: https://books.google.co.uk/books?id=gbPoBgAAQBAJ.

[106] Nandita Dukkipati and Nick McKeown. “Why Flow-completion Time is the Right Metric for Congestion Control”. In: *Proceedings of the 2006 SIGCOMM Conference* 36.1 (Jan. 2006), pp. 59–62. ISSN: 0146-4833. DOI: 10.1145/1111322.1111336. URL: http://doi.acm.org/10.1145/1111322.1111336.

[107] Y. Zhang and T. R. Henderson. “An implementation and experimental study of the explicit control protocol (XCP).” In: *INFOCOM 2006*. IEEE, Apr. 2006, pp. 1037–1048. URL: http://dx.doi.org/10.1109/INFCOM.2005.1498332.

[108] David X. Wei et al. “FAST TCP: Motivation, Architecture, Algorithms, Performance”. In: *IEEE/ACM Trans. Netw.* 14.6 (Dec. 2006), pp. 1246–1259. ISSN: 1063-6692. DOI: 10.1109/TNET.2006.886335. URL: http://dx.doi.org/10.1109/TNET.2006.886335.

[109] M. Podlesny and C. Williamson. “Solving the TCP-Incast Problem with Application-Level Scheduling”. In: *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. Aug. 2012, pp. 99–106. DOI: 10.1109/MASCOTS.2012.21.

[110] A. Aggarwal, S. Savage, and T. Anderson. “Understanding the performance of TCP pacing”. In: *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No.00CH37064)*. Vol. 3. Mar. 2000, 1157–1165 vol.3. DOI: 10.1109/INFCOM.2000.832483.

[111] W. Chen et al. “Ease the Queue Oscillation: Analysis and Enhancement of DCTCP”. In: *2013 IEEE 33rd International Conference on Distributed Computing Systems*. July 2013, pp. 450–459. DOI: 10.1109/ICDCS.2013.22.

[112] Mo Dong et al. “PCC: Re-architecting Congestion Control for Consistent High Performance”. In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 395–408. ISBN: 978-1-931971-218. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/dong.

[113] Keith Winstein and Hari Balakrishnan. “TCP Ex Machina: Computer-generated Congestion Control”. In: *Proceedings of the 2013 SIGCOMM Conference*. SIGCOMM ’13. Hong Kong, China: ACM, 2013, pp. 123–134. ISBN: 978-1-4503-2056-6. DOI: 10.1145/2486001.2486020. URL: http://doi.acm.org/10.1145/2486001.2486020.

[114] Jonathan Perry et al. “Fastpass: A Centralized ”Zero-queue” Datacenter Network”. In: *Proceedings of the 2014 SIGCOMM Conference*. SIGCOMM ’14. Chicago, Illinois, USA: ACM, 2014, pp. 307–318. ISBN: 978-1-4503-2836-4. DOI: 10.1145/2619239.2626309. URL: http://doi.acm.org/10.1145/2619239.2626309.

[115] “Cisco Application Centric Infrastructure Fundamentals”. In: *Cisco Application Centric Infrastructure Fundamentals*. Ed. by Inc. Cisco Systems. 2017. Chap. Forwarding Within the ACI Fabric, p. 106.

[116] Kathleen Nichols Jim Gettys. “Bufferbloat: Dark Buffers in the Internet”. In: *ACM Queue* 9.11 (Nov. 2011), 40:40–40:54. DOI: 10.1145/2063166.2071893. URL: http://doi.acm.org/10.1145/2063166.2071893.

[117] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. “Sizing Router Buffers”. In: *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’04. Portland, Oregon, USA: ACM, 2004, pp. 281–292. ISBN: 1-58113-862-8. DOI: 10.1145/1015467.1015499. URL: http://doi.acm.org/10.1145/1015467.1015499.

[118] Ravi S. Prasad, Constantine Dovrolis, and Marina Thottan. “Router Buffer Sizing for TCP Traffic and the Role of the Output/Input Capacity Ratio”. In: *IEEE/ACM Trans. Netw.* 17.5 (Oct. 2009), pp. 1645–1658. ISSN: 1063-6692. DOI: 10.1109/TNET.2009.2014686. URL: http://dx.doi.org/10.1109/TNET.2009.2014686.

[119] V. Paxson et al. *Computing TCP's Retransmission Timer*. RFC 6298 (Proposed Standard). Internet Engineering Task Force, June 2011. URL: http://www.ietf.org/rfc/rfc2001.txt.

[120] Charalampos Rotsos et al. "OFLOPS: An Open Framework for Openflow Switch Evaluation". In: *Proceedings of the 13th International Conference on Passive and Active Measurement*. PAM'12. Vienna, Austria: Springer-Verlag, 2012, pp. 85–95. ISBN: 978-3-642-28536-3. DOI: 10.1007/978-3-642-28537-0_9. URL: http://dx.doi.org/10.1007/978-3-642-28537-0_9.

[121] S. Cheshire. *IPv4 Address Conflict Detection*. RFC 5227 (Proposed Standard). Internet Engineering Task Force, July 2008. URL: http://www.ietf.org/rfc/rfc5227.txt.

[122] Mohammad Al-Fares et al. "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. URL: http://dl.acm.org/citation.cfm?id=1855711.1855730.

[123] Nick McKeown Bob Lantz Brandon Heller. "A Network in a Laptop: Rapid Prototyping for Software-defined Networks". In: *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. Hotnets-IX. Monterey, California: ACM, 2010, 19:1–19:6. ISBN: 978-1-4503-0409-2. DOI: 10.1145/1868447.1868466. URL: http://doi.acm.org/10.1145/1868447.1868466.

[124] Nikhil Handigol et al. "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. ISBN: 978-1-4503-1775-7. DOI: 10.1145/2413176.2413206. URL: http://doi.acm.org/10.1145/2413176.2413206.

[125] P. Arberg et al. *Accommodating a Maximum Transit Unit/Maximum Receive Unit (MTU/MRU) Greater Than 1492 in the Point-to-Point Protocol over Ethernet (PP-*

PoE). RFC 4638 (Informational). Internet Engineering Task Force, Sept. 2006. URL: http://www.ietf.org/rfc/rfc4638.txt.

[126] *tc-red(8) - red - Random Early Detection*. [Online; accessed 27-09-2017]. July 2017. URL: http://man7.org/linux/man-pages/man8/tc-red.8.html.

[127] J. Nagle. *Congestion Control in IP/TCP Internetworks*. RFC 896. Internet Engineering Task Force, Jan. 1984. URL: http://www.ietf.org/rfc/rfc896.txt.

[128] Pica8 – Open Networking. *OpenFlow Data Center, A case study*. [Online; accessed 27-09-2017]. Feb. 2014. URL: https://www.slideshare.net/nvirters/openflow-data-center-pica8.

[129] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. San Diego, California: USENIX Association, 1993, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=1267303.1267305.

[130] Alexei Starovoitov Jay Shulist Daniel Borkmann. *Linux Socket Filtering aka Berkeley Packet Filter (BPF)*

. [Online; accessed 27-09-2017]. URL: https://www.kernel.org/doc/Documentation/networking/filter.txt.

[131] IOvisor. *IOvisor Project*. [Online; accessed 27-09-2017]. URL: https://www.iovisor.org/.

[132] Intel. *Intel DPDK: Data Plane Development Kit*. [Online; accessed 27-09-2017]. URL: http://www.dpdk.org.

[133] Rothenberg Christian Esteve Fernandes Eder Leão. *OpenFlow 1.3 Software Switch*. [Online; accessed 27-09-2017]. 2014. URL: https://github.com/CPqD/ofsoftswitch13.

[134] Linux Kernel. *packet - packet interface on device level*. PACKET 7. [Online; accessed 27-09-2017]. Sept. 2017. URL: http://man7.org/linux/man-pages/man7/packet.7.html.

[135] Paul Barham et al. "Xen and the Art of Virtualization". In: *SIGOPS Operating Systems Review* 37.5 (Oct. 2003), pp. 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462. URL: http://doi.acm.org/10.1145/1165389.945462.

[136] Fabrice Bellard. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, pp. 41–46. URL: http://dl.acm.org/citation.cfm?id=1247360.1247401.

[137] Rich Lane IOvisor. *uBPF: Userspace eBPF VM*. [Online; accessed 27-09-2017]. URL: https://github.com/iovisor/ubpf.

[138] R. Cziva et al. "SDN-Based Virtual Machine Management for Cloud Data Centers". In: *IEEE Transactions on Network and Service Management* 13.2 (June 2016), pp. 212–225. ISSN: 1932-4537. DOI: 10.1109/TNSM.2016.2528220.

[139] J. Postel. *Transmission Control Protocol*. RFC 793 (Standard). Updated by RFCs 1122, 3168, 6093. Internet Engineering Task Force, Sept. 1981. URL: http://www.ietf.org/rfc/rfc793.txt.

[140] V. Jacobson. "Congestion Avoidance and Control". In: *Symposium Proceedings on Communications Architectures and Protocols*. Proceedings of the 2008 SIGCOMM Conference. Stanford, California, USA: ACM, 1988, pp. 314–329. ISBN: 0-89791-279-9. DOI: 10.1145/52324.52356. URL: http://doi.acm.org/10.1145/52324.52356.

[141] Richard Cziva, Christopher Lorier, and Dimitrios P. Pezaros. "Ruru: High-speed, Flow-level Latency Measurement and Visualization of Live Internet Traffic". In: *Proceedings of the SIGCOMM Posters and Demos*. SIGCOMM Posters and Demos '17. Los Angeles, CA, USA: ACM, 2017, pp. 46–47. ISBN: 978-1-4503-5057-0. DOI:

10.1145/3123878.3131981. URL: http://doi.acm.org/10.1145/3123878.3131981.

[142] Daniel Turull, Peter Sjödin, and Robert Olsson. "Pktgen: Measuring performance on high speed networks". In: *Computer Communications* 82.Supplement C (2016), pp. 39–48. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2016.03.003. URL: http://www.sciencedirect.com/science/article/pii/S0140366416300615.

[143] J. Myers. *SMTP Service Extension for Authentication*. RFC 2554 (Proposed Standard). Obsoleted by RFC 4954. Internet Engineering Task Force, Mar. 1999. URL: http://www.ietf.org/rfc/rfc2554.txt.

[144] László Molnár et al. "Dataplane Specialization for High-performance OpenFlow Software Switching". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 539–552. ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934887. URL: http://doi.acm.org/10.1145/2934872.2934887.

[145] Applied Research Center for Computer Networks (ARCCN). *Collection of scripts for OpenFlow controllers performance testing*. [Online; accessed 27-09-2017]. June 2013. URL: http://arccn.github.io/ctltest/.

[146] R. Cziva et al. "SDN-Based Virtual Machine Management for Cloud Data Centers". In: *IEEE Transactions on Network and Service Management* 13.2 (June 2016), pp. 212–225. ISSN: 1932-4537. DOI: 10.1109/TNSM.2016.2528220.

[147] PK Gupta. "Xeon+FPGA Platform for the Data Center". In: *The Fourth Workshop on the Intersections of Computer Architecture and Reconfigurable Logic*. June 2015.

[148] Nicole Hemsoth. *Intel Marrying FPGA, Beefy Broadwell for Open Compute Future*. https://www.nextplatform.com/2016/03/14/intel-marrying-fpga-beefy-broadwell-open-compute-future/. [Online; accessed 27-09-2017]. Mar. 2016.

[149] Dimitrios P Pezaros, Richard Cziva, and Simon Jouet. “SDN for Cloud Data Centres”. In: *IET Big-Data and Software Defined Networks*. IET Book Series on Big Data. IET, Sept. 2017.

[150] Abeer Ali et al. “SDNFV-based DDoS detection and remediation in multi-tenant, virtualised infrastructures”. In: *Guide to Security in SDN and NFV - Challenges, Opportunities, and Applications*. GSSNOA ’16. Springer, 2017.

[151] M. Iordache et al. “Distributed, multi-level network anomaly detection for datacentre networks”. In: *2017 IEEE International Conference on Communications (ICC)*. May 2017, pp. 1–6. DOI: 10.1109/ICC.2017.7996569.

[152] A. Pamukchiev, S. Jouet, and D. P. Pezaros. “Distributed network anomaly detection on an event processing framework”. In: *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*. Jan. 2017, pp. 659–664. DOI: 10.1109/CCNC.2017.7983209.

[153] C. Mas Machuca et al. “Technology-related disasters: A survey towards disaster-resilient Software Defined Networks”. In: *2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM)*. Sept. 2016, pp. 35–42. DOI: 10.1109/RNDM.2016.7608265.

[154] T. Gomes et al. “A survey of strategies for communication networks to protect against large-scale natural disasters”. In: *2016 8th International Workshop on Resilient Networks Design and Modeling (RNDM)*. Sept. 2016, pp. 11–22. DOI: 10.1109/RNDM.2016.7608263.

[155] Richard Cziva, Simon Jouet, and Dimitrios P Pezaros. “Roaming Edge vNFs Using Glasgow Network Functions”. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. Proceedings of the 2016 SIGCOMM Conference. Florianopolis, Brazil: ACM, 2016, pp. 601–602. ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2959067. URL: http://doi.acm.org/10.1145/2934872.2959067.

[156] Fung Po Tso, Simon Jouet, and Dimitrios P Pezaros. "Network and server resource management strategies for data centre infrastructures: A survey". In: *Computer Networks* 106 (2016), pp. 209–225. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2016.07.002. URL: http://www.sciencedirect.com/science/article/pii/S1389128616302298.

[157] R. Cziva, S. Jouet, and D. P. Pezaros. "GNFC: Towards network function cloudification". In: *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. Nov. 2015, pp. 142–148. DOI: 10.1109/NFV-SDN.2015.7387419.

[158] R. Cziva et al. "Container-based network function virtualization for software-defined networks". In: *2015 IEEE Symposium on Computers and Communication (ISCC)*. July 2015, pp. 415–420. DOI: 10.1109/ISCC.2015.7405550.

[159] F. P. Tso et al. "The Glasgow Raspberry Pi Cloud: A Scale Model for Cloud Computing Infrastructures". In: *2013 IEEE 33rd International Conference on Distributed Computing Systems Workshops*. July 2013, pp. 108–112. DOI: 10.1109/ICDCSW.2013.25.