Enabling low-latency applications using programmable networks

Turkovic, B.

DOI
10.4233/uuid:10656c8a-d9f0-45bb-a012-e84c8ee745ac

Publication date
2022

Document Version
Final published version

Citation (APA)
Turkovic, B. (2022). *Enabling low-latency applications using programmable networks*. [Dissertation (TU Delft), Delft University of Technology]. https://doi.org/10.4233/uuid:10656c8a-d9f0-45bb-a012-e84c8ee745ac

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Enabling Low-Latency Applications Using Programmable Networks

Enabling Low-Latency Applications Using Programmable Networks

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. T.H.J.J. van der Hagen,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op woensdag 26 oktober 2022 om 12:30 uur

door

Belma Turković

Master of Science in Electrical Engineering,
University of Sarajevo, Sarajevo, Bosnië en Herzegovina,
geboren te Zenica, Bosnië en Herzegovina.

Dit proefschrift is goedgekeurd door de promotoren.

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
prof. dr. K.G. Langendoen,	Technische Universiteit Delft, promotor
prof. dr. ir. F.A. Kuipers,	Technische Universiteit Delft, promotor
Onafhankelijke leden:	
Prof. dr. G. Smaragdakis,	Technische Universiteit Delft
Prof. dr. A.J. Kassler,	Universiteit van Karlstad, Zweden
Prof. dr. ir. D. Colle,	Universiteit Gent, België
Prof. dr. ir. G.J. Heijenk,	Universiteit Twente
Dr. P. Grosso,	Universiteit van Amsterdam
Prof. dr. ir. A.J. van der Veen	Technische Universiteit Delft, reservelid

Keywords: Programmable networks, Low-latency applications, Programmable data-planes, Software-Defined networking

ISBN 000-00-0000-000-0

An electronic version of this dissertation is available at
http://repository.tudelft.nl/.

To my parents Berina and Irfan,
for their endless love, support and encouragement.

CONTENTS

SUMMARY

Throughout the last decades, communication networks have become embedded into almost every aspect of our day-to-day lives (e.g., watching movies, online shopping, sharing moments with friends and family). Moreover, as the support for the transport of audio and video became the norm, new application domains have kept emerging every day. One of these, the Tactile Internet, enables the transport of the sense of touch. Consequently, it allows the end-users to interact with a remote environment in the same way they would if they were present locally. While such applications could revolutionize many industries by enabling users to transport their skills (e.g., surgical skills) across the globe, they pose many new challenges to communication networks, such as the need for very low latency. Yet, providing low latency is fundamentally different from providing high bandwidth, and, as this thesis demonstrates, existing solutions developed for bandwidth-oriented services are not directly applicable to low-latency services.

This thesis explores how programmable networks can be used to facilitate emerging low-latency services. Specifically, it combines the advantages of (1) Software-Defined-Networking (SDN), a paradigm in networking that centralizes the control plane, and (2) programmable data planes, which enable an on-the-fly deployment of novel algorithms to the network switches. In particular, this thesis explores what SDN controller tasks are feasible to be offloaded to the data plane, the trade-offs in doing so, and their benefits on low-latency applications. Moreover, it takes advantage of the more fine-grained monitoring possibilities of programmable data planes and incorporates these measurements into the data plane algorithms. As a result, this thesis develops a set of solutions that enable network switches to react to short-term changes in the networking traffic and act independently (or with limited input), improving the Quality of Service (QoS) of low-latency flows.

First, we investigate the limitations of programmable switches and ways to overcome them by developing an application to detect heavy hitters (e.g., flows that consume most resources in the network). Next, we explore the concept of network slicing, i.e., reserving part of a physical network for a specific service. We demonstrate that network switches can combine data plane measurements and limited (preconfigured) input from the central controller to enable elasticity, i.e., the ability to automatically scale the assigned network resources based on the flows' requirements with negligible delay. Next, we analyze the co-existence and interactions between flows using different congestion control algorithms and/or having different RTTs. We use this information to develop a data plane algorithm to improve their interactions. Finally, we demonstrate how congestion detection and avoidance can be achieved in the data plane without any assistance from the end-hosts.

SAMENVATTING

De afgelopen decennia zijn communicatienetwerken niet meer weg te denken uit ons dagelijks leven, van online winkelen tot het kijken van films, en het delen van speciale momenten met vrienden en familie. Sinds het transporteren van beeld en audio de norm is geworden, verschijnen er dagelijks nieuwe toepassingsdomeinen. Eén van deze nieuwe domeinen, het Tactiele Internet, maakt het mogelijk om het gevoel van aanraking te transporteren. Dit maakt het voor eindgebruikers mogelijk om over verre afstanden acties uit te voeren, op dezelfde manier als dat de gebruiker lokaal aanwezig zou zijn. Al kunnen zulke applicaties veel industrieën revolutioneren door gebruikers (e.g., chirurgen) hun vaardigheden overal ter wereld—op afstand—toe te laten passen, creëren ze nieuwe uitdagingen voor communicatienetwerken. Tactiel Internet vereist bijvoorbeeld een zeer lage latentie. Het verstrekken van een lage latentie is echter fundamenteel verschillend van het verstrekken van hoge bandbreedte, en, zoals deze dissertatie demonstreert, zijn de bestaande oplossingen voor bandbreedte georiënteerde voorzieningen niet direct toepasbaar op latentie georiënteerde voorzieningen.

Deze dissertatie onderzoekt hoe programmeerbare netwerken gebruikt kunnen worden om de opkomende latentie-gevoelige applicaties te ondersteunen. De dissertatie combineert de voordelen van (1) Software-Defined-Networking (SDN), een netwerkparadigma die de control plane centraal stelt, en (2) programmeerbare data planes, welke het mogelijk maken om direct (on-the-fly) nieuwe algoritmes in te zetten op de netwerkknopen. In het bijzonder verkent deze dissertatie welke taken van de SDN-controller overgenomen kunnen worden door de data plane en de geassocieerde voor- en nadelen. Hierbij ligt onze focus op applicaties die hoge eisen stellen aan latentie. Daarnaast, maakt deze dissertatie gebruik van de preciezere meetmogelijkheden van programmeerbare data planes en integreert deze metingen in de data plane algoritmes. Het resultaat van deze dissertatie is een set van oplossingen die netwerkknopen de mogelijk geven om zelfstandig (of met gelimiteerde instructies) te reageren op kotertermijn veranderingen in het netwerkverkeer. Deze oplossingen verhogen de Quality of Service (QoS) van netwerkverkeer met lage latentie eisen.

Allereerst onderzoeken we de beperkingen van programmeerbare netwerkknopen en hoe we deze kunnen overkomen door een applicatie te ontwikkelen die “heavy hitters” (netwerkstromen die de meeste netwerkcapaciteit gebruiken) detecteert. Vervolgens onderzoeken we “network slicing”, het reserveren van een deel van het fysieke netwerk voor een specifieke dienst. We demonstreren dat netwerkknopen data plane metingen kunnen combineren met een kleine (voorgeconfigureerde) hoeveelheid gegevens van de centrale controller om de elasticiteit van het netwerk (het vermogen om binnen verwaarloosbare tijd automatisch netwerkmiddelen op te schalen op basis van de eisen van netwerkstromen) te verhogen. Vervolgens analyseren we het naast elkaar bestaan van, en de interactie tussen, stromen die verschillende congestie control algoritmes gebruiken en/of verschillende retourtijden hebben. Met behulp van deze informatie ont-

wikkelen we een data plane algoritme die deze interactie verbetert. Tot slot demonstreren we hoe congestie detectie en vermijding bereikt kunnen worden in de data plane zonder assistentie van de eindhost.

1

INTRODUCTION

Due to the rising popularity of streaming services, the demand for bandwidth is growing fast. In fact, video traffic is currently responsible for well over half of all Internet traffic [1, 2]. To cope with this rising trend, multiple strategies that allow network operators to grow the available capacity exist. For example, operators could improve existing techniques thereby enabling higher data rates through existing fiber-optic links or add more fiber-optic links across congested routes. Hence, while it may not be cheap, there is no limit on bandwidth increase over time [3]. However, with the advances in networking, new application domains are emerging and, as illustrated in Figure 1.1, many of them no longer require just high bandwidth, but very low latency as well, posing a new challenge for communication networks [4, 5]. Consequently, while previously just a desirable feature, low latency has become a hard requirement for many services replacing the previous "need for speed" with the "need for latency."

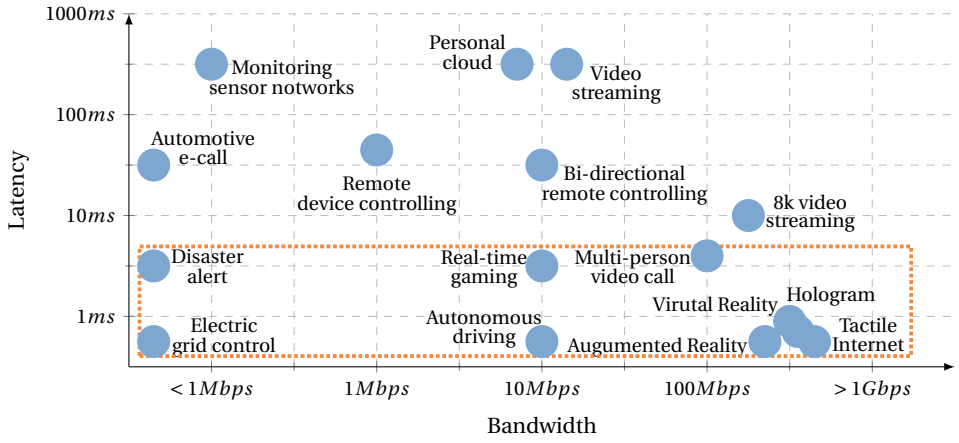
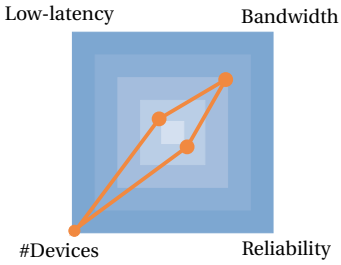


Figure 1.1: Bandwidth vs. latency. Network demands of emerging technologies. The orange line represents new, emerging low-latency services. Based on [6–9].

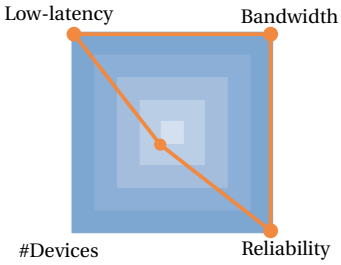
For example, the objective of one of these low-latency application domains, called the Tactile Internet, is to transmit a sense of touch over the Internet enabling end-users in different locations to interact in a shared, virtual or physical environment as if they were in the same room [4, 5, 10–15]. While still in its infancy, Tactile Internet has applications in industry, robotics, telepresence, virtual reality, augmented reality, healthcare, road traffic, serious gaming, education, and culture [14]. As an example, imagine a doctor in Delft using a remote robot to perform real-time surgery on a patient in a different city. Because surgeons have to see and feel what they are doing, multiple modalities – audio, video, and data involving the sense of touch – need to be sent together and synchronized to improve the surgeons' perceptual experience. However, in contrast to standard audio and video streaming services, surgeons' actions controlling the robot must be transmitted in the opposite direction closing a control loop with very stringent latency requirements [12]. Therefore, this type of traffic, in addition to a high bandwidth requirement (due to multiple sent and synced modalities), is also extremely sensitive to latency, requiring end-to-end latency as low as 1 ms (Figure 1.2). Violating these requirements can produce unwanted effects. Cybersickness (which is similar to motion sickness [4, 14, 16]) or control loop instability may occur, reducing the medical service quality and potentially injuring the patient.

Another application domain requiring low-latency is smart grids [17, 18]: modernized power grids that, based on the information collected from them, adjust the production and distribution of electricity. Therefore, in contrast to traditional power grids, in which the devices are monitored manually onsite, smart grids monitor, measure, adjust, and control these devices' power usage remotely, over a network. Consequently, previously unimaginable capabilities, such as self-healing features, i.e., the ability to automatically detect and respond to grid problems and ensure quick recovery after disturbances, will become a reality [19]. However, while not bandwidth-intense, such services require an up-to-date collection and exchange of information, results, and decisions, requiring high reliability and latencies as low as a few milliseconds (Figure 1.2) [18, 20, 21].

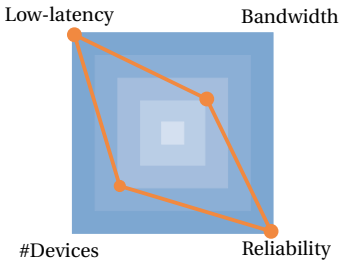
Moreover, even existing markets, such as competitive online gaming, would have a considerable benefit from consistent low-latency responsiveness, especially as video games move to the cloud streaming model [22–24] and are combined with augmented and virtual reality technologies [25]. For instance, even for current "traditional" online games, in a recent survey in the UK, 44% of the participants named "the internet lagging" as the most infuriating aspect of online gaming. Furthermore, a lower, consistent latency could increase the adaptation and growth of new emerging gaming domains, such as multiplayer virtual/augmented online reality games. For example, to solve one of these games' main bottlenecks – the need for more computational power – service providers could use centralized cloud services equipped with dedicated GPU cores. Consequently, the cloud would render the graphics and stream it back as a compressed video to the end-users. However, such a solution would also introduce additional latency to the already latency-critical control loop of augmented/virtual reality rendering (less than 10 ms round-trip time [26], Figure 1.2), simply due to the physical distance between the user and the cloud. Similar to the Tactile Internet scenario, violating these requirements would lead to a poor user experience, low adaptation rate of such games, and potential cybersickness.



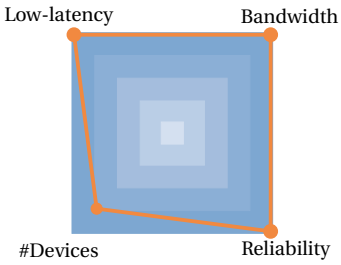
(a) Video streaming (Traditional services)



(b) Remote surgery (Tactile Internet)



(c) Precise Load Control (Smart grid).



(d) Cloud VR Gaming (Real-time gaming)

Figure 1.2: Requirements of different services. Darker squares represent that the specific service has more stringent requirements for the particular metric.

1.1. HIGHER BANDWIDTH ≠ LOWER LATENCY

Simply defined, latency is the time it takes for a packet to travel from a source to a destination host. While it might seem that merely increasing the bandwidth will lower the latency, this is not the case. By doing this, the real problem, i.e., the source of the excess latency, is usually either just temporarily hidden or not targeted at all. To better understand the relation between bandwidth and latency network latency's main components need to be understood. A packet typically encounters four types of latency between each two network nodes [3, 27]:

- Propagation latency: a function of the physical distance between two nodes and propagation speed of a link.
- Transmission latency: a function of the packet's size and data rate of a link.
- Processing latency: the time required to inspect the packet header and determine its destination.
- Queuing latency: the amount of time the packet is waiting in the queue until it can be transmitted.

The first component, the propagation latency, is dictated only by the physical distance and material through which the signal travels. As such, it does not depend on the

available bandwidth or any other traffic processed in the network. Moreover, in wired networks (considered in this thesis), the propagation speed is usually within a small constant factor of the speed of light. Therefore, this latency component is constant on a given path, cannot be reduced, and represents a lower bound on the end-to-end latency every packet experiences on a given path [3]. Moreover, in most network deployments, the propagation latency usually only represents a small portion of the total end-to-end latency experienced by packets, leaving little room for improvement.

Consequently, to support the emerging low-latency services, one must consider the other three latency components. For a packet of some constant size, the first of these, the transmission latency, depends only on each transmitting link's bandwidth. Therefore, to reduce it, a network operator can choose to increase the bandwidth on either of the links or simply reduce the number of traversed links. However, similar to propagation delay, on a given path transmission latency is constant (given the same packet size), and on speeds reaching Gbps, for typical network packets (e.g., $1500B$), represents only a tiny portion of the total end-to-end latency, e.g., only a few nanoseconds on a $100Gbps$ link per each network node. Therefore, while network operators can reduce this component through careful planning, the performance gain is limited.

The third component, the processing latency, represents the time network nodes need to process a packet. For example, each router must examine the packet's header to determine the outgoing port before processing the packet further along the path. This action and many others (e.g., reducing the TTL, changing the MAC addresses), are applied to each incoming packet. However, as much of this logic is performed in hardware, these actions are usually fast, constant, and depend solely on the deployed hardware's speed. Consequently, they represent a constant factor to the overall end-to-end latency on a given path. Therefore, similarly to the transmission latency, performance gains, such as deploying faster hardware or reducing the number of network nodes the packet traverses, are limited.

It is essential to notice that all of the above-mentioned components (i.e., propagation, transmission, processing latency) represent a constant small contribution to the overall end-to-end latency. Therefore, they are predictable and more straightforward to account for in networks compared to variable delay. In contrast, the last component, the queuing latency, depends mainly on the amount of traffic and the way that traffic is handled in a network. Since network traffic can be very bursty while the output links have a fixed bandwidth, packets might arrive faster than the output links can handle, causing congestion. To reduce packet loss, each network node is equipped with buffers absorbing these short-term fluctuations in network traffic. However, packets waiting to be processed further along the path will experience an increased variable delay depending on the current queue size, called the queueing latency. As such, these delays may vary significantly, and controlling and reducing them is crucial and, therefore, the main topic of this thesis. An obvious solution to this problem is to increase the bandwidth of these output links or to add more links to the congested routes. However, such a solution is resource inefficient and would result in low network utilization. Furthermore, increasing the bandwidth will only give more room to the bandwidth-demanding services (e.g., downloads, torrents) that seek to claim as many resources as they can, just delaying the inevitable congestion to a later moment in time.

To conclude, two critical network parameters, latency, and bandwidth, usually work together to dictate all network traffic performance and needs to be managed. However, while high-bandwidth links are desirable, they do not guarantee a stable end-to-end performance. Networks could be congested at any intermediate node at some point in time due to, for example, high demand, hardware failures, and concentrated network attacks. Consequently, to enable bounded end-to-end latency guarantees for low-latency applications, controlling the queuing latency has to be an explicit design criterion at all development and deployment stages.

1.2. MANAGEMENT OF QUEUEING DELAY

As mentioned in the previous section, one of the most significant factors contributing to queuing delay is congestion, which occurs when a network node is trying to process more data than a link can handle. Congestion control, i.e., mechanisms deployed to minimize the occurrence and the effects of congestion, has been a well-researched topic since Van Jacobson proposed it for the first time in 1988 [28]. Moreover, from application-level and TCP/QUIC congestion control algorithms [27–61] to network-based active queue management [62–77], many different mechanisms have been proposed over the years. However, at the same time, many of them were considered too complicated, risky, and had poor interoperability with existing mechanisms to be deployed in production networks. Consequently, the adaptation of many proposed techniques was slow or non-existent. Instead, network operators often choose the easy route of overprovisioning, leading to very low utilized, energy-inefficient networks.

Currently, most networks rely only on older proven congestion control algorithms deployed in the end-hosts to detect congestion informing the end-hosts to modify their sending rates accordingly. However, for many of the previously mentioned low-latency services, such as remote surgery or virtual-reality online gaming, this approach is not feasible. One cannot increase/decrease the rate at the traffic source. Moreover, even for services where one could, most of these control algorithms would only kick in *after* congestion has occurred, and the queues were filled, needing at least one round-trip time (RTT) to react to the encountered congestion. Thus, potentially critical packets would already be outdated at their reception, degrading the expected performance with sometimes very severe consequences (e.g., injury to the patient in remote surgery).

Software-defined networking (SDN [78, 79], Figure 1.3), as a new paradigm in networking, offers an alternative. In SDN, network nodes, previously responsible for many tasks (e.g., routing table build-up using routing protocols such as OFPF), were simplified to be only responsible for packet forwarding. Hence, SDN network nodes usually only contain a set of match-action tables that associate a particular set of header fields (e.g., MAC addresses, IP addresses) to a set of forwarding actions (e.g., output a packet to a specific port). Moreover, all SDN network nodes are connected to a software-based controller, a separate entity, that determines the exact table entries, i.e., this mapping between the match fields (e.g., header fields, metadata) and the forwarding actions. When the network nodes receive a packet they do not know how to process (i.e., no match is found in the forwarding tables), they contact the controller, and the controller subsequently executes a routing protocol. Finally, upon determining the forwarding rules for the packet, the controller updates the necessary forwarding tables in the network nodes.

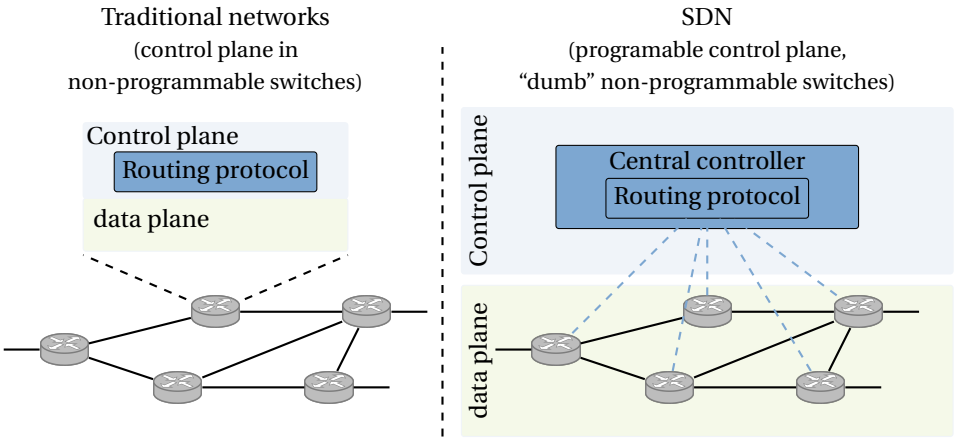


Figure 1.3: Traditional networks vs. SDN. SDN, in contrast to traditional networking approaches, separates the control plane and the data plane: central controllers make routing decisions while switches only forward the traffic.

All subsequent packets belonging to the same flow (i.e., packets having the same values for all the match fields in the forwarding tables) are processed using these installed rules and without the need to contact the controller again.

It is crucial to notice that the executed routing algorithm is just another software application that needs to be installed in the controller. This significantly simplifies the development and deployment of new custom protocols that previously needed to be supported in each switch's ASIC. Moreover, with SDN the central controller has a global view of the network and all its traffic (in contrast to network nodes in traditional networks, Figure 1.3). Hence, it can combine the information from of all the switches to determine the current network state (e.g., current flows, available links, connections between nodes, bandwidth available on each link) and, subsequently, adapt to the traffic and changing network conditions, by, for example, rerouting specific flows from congested routes and/or potentially offering special treatment for particular flows. Taking advantage of this, many different SDN frameworks trying to improve the Quality of Sevice (QoS) have been developed over the years, offering many possibilities for advanced network reconfiguration and queue management. We can divide these frameworks into two distinct groups: QoS routing frameworks and resource reservation frameworks.

QoS routing frameworks. Many frameworks use some form of QoS routing to find the path that satisfies different QoS requirements [80–86]. However, SDN frameworks from this group depend on very precise monitoring [87, 88]. First, monitoring intervals directly influence the gathered data's usefulness, as well as the number of monitoring packets sent. Even newer monitoring approaches, such as the streaming telemetry, in which the switches push incremental data updates to the central controller, can generate too much overhead, depending on the number of subscribed events and still depend on the latency between switches and the controller [89]. Second, even after congestion is detected, a non-zero time is needed for the controller to recompute the path and recon-

figure the forwarding table entries before switching the flow to a better path (if it exists) or throttling another flow (by for example rerouting it to a route that has less resources). At the same time, until rerouted, all packets are processed by the congested node causing an increase in the total end-to-end delay.

Another option to ensure that the QoS requirements of each flow are satisfied would be to know in advance, i.e., at the moment a new flow is initiated, its bandwidth demands throughout its duration. By using this knowledge, the central controller could make sure, while configuring the routes, that the total bandwidth is not exceeded on each link. However, a flow's bandwidth demands are usually very variable and unpredictable, i.e., it depends on the current applications' dynamic and user behavior.

Resource reservation frameworks. Other standard methods to provide QoS in SDN are (1) to use priority queuing or (2) to implement virtual slicing of the available bandwidth on all the nodes on the path, reserving parts of it to different services, effectively isolating them [90–96]. On the one hand, while initially minimizing queuing delay for the higher prioritized flows, priority queuing can lead to starvation of flows and does not prevent congestion, especially between flows having the same priority, i.e., belonging to the same service. In fact, high-priority flows will starve when congestion forces low-priority flows to occupy all available queue space.

On the other hand, as mentioned above, the maximum, or even the average or current bandwidth requirement is usually variable and not known beforehand, providing a severe flaw in these mechanisms. Moreover, even if we could determine the required bandwidth, it can be in the order of a few Gbps (e.g., Tactile internet services, augmented, and virtual reality [5–7, 97], Figure 1.1). Therefore, reserving the maximum required bandwidth for every flow would not be scalable. However, reserving less than the maximum required bandwidth does not ensure the strict per-flow QoS many low latency applications need. Besides, networks often will not need to simultaneously support all of these services with very stringent requirements for all end-hosts. Thus, provisioning the networks to meet all services' peak requirements would lead to inefficient, overcompensated, and far too expensive networks. Instead, networks should be flexible and programmable, adapting to individual end-users and services' needs and delivering precisely the needed performance.

One way to improve network flexibility could be to use a central controller to adjust the network resources assigned to a service, i.e., increasing or decreasing the resources assigned to a service depending on its current needs. However, such a solution would still depend on precise monitoring to determine the current service needs. Moreover, as mentioned earlier, pure SDN solutions will always depend on the latency between switches and the controller. Consequently, the gathered information, as well as the controller's calculated best response, might already be outdated before the rules are pushed to the switches. Therefore, all deployed algorithms will inherently be slow and potentially lead to oscillations and inconsistencies due to the variable delay between the switches and the controller.

In conclusion, resource reservation frameworks offer a good starting point, providing sound isolation between different services. Specifically, by reserving a share of the available network resources for a service, a network operator can make sure that different services will not interfere with each other, i.e., degrade the QoS of either of them. However,

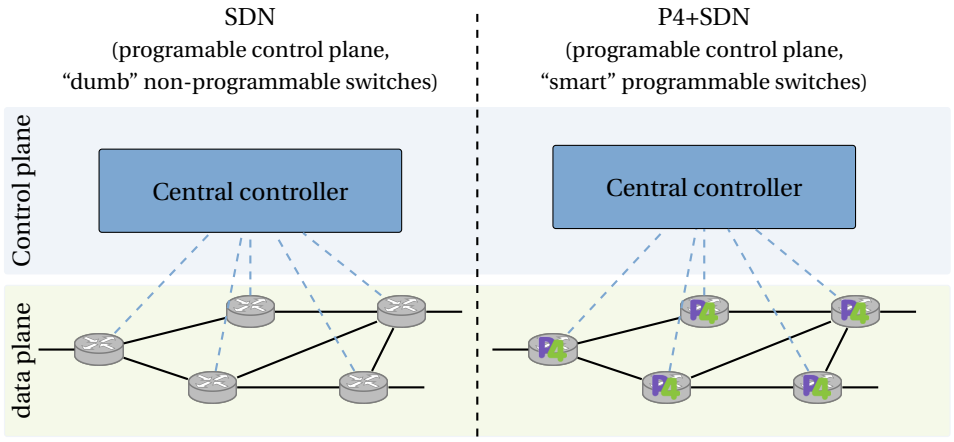


Figure 1.4: SDN vs. P4+SDN. P4 enables the data plane to be programmed as well, providing fully programmable networks. Previously “dumb” fixed-function switches are replaced by programmable switches able to execute custom algorithms providing a fast reaction loop for latency-sensitive applications.

at the same time, current SDN implementations lack a fast response to changing service and traffic requirements and are, as a consequence, unable to adjust the resources to the current service needs in real-time. Furthermore, they do not provide any mechanisms to ensure that the same service flows, while isolated from flows belonging to other services, would achieve good service quality and share the assigned resources fairly.

1.2.1. PROGRAMMABLE NETWORKS

To solve the problems identified in the previous section, i.e., low reaction speed to traffic and user changes of the central controllers, a possible solution would be to switch back to the traditional network architecture with a distributed control plane. This way, the switches themselves could react to the traffic changes providing a fast reaction loop. However, with its traditional fixed functions and lack of reconfigurability, typical network hardware limits new algorithms’ deployment. This, in turn, limits the agility needed to support the diverse QoS requirements of newer services. Furthermore, by switching to a distributed model, all the SDN architecture’s advantages, such as the centralized view of the network enabling fine-grained traffic management, would be lost.

Programmable switches, another solution offering flexibility, decouple the actual forwarding program run on the switch from its hardware. Hence, the network operators could design, implement, test, and, finally deploy, custom forwarding schemes to all the switches in its network, potentially providing different treatments to different applications. Moreover, using programmable switches, the network operators would no longer depend on the vendors to support their specific algorithm, improving the interoperability between switches belonging to different vendors, and avoiding vendor lock-in. However, programmable switches were for years significantly less performant than legacy fixed-function switches and hence, not a viable option. Nevertheless, with the new advancements in hardware, they have become available at a comparable price and

performance [98]. Moreover, due to the emergence of these switches, new high-level domain-specific programming languages such as P4 were designed, allowing network engineers to test and deploy new algorithms quickly without worrying about the details of the underlying hardware.

However, to be a feasible replacement for a high-performance fixed-function switch, programmable switches need to process packets at speeds reaching Tbps without any degradation. Hence, per processed packet, a switch would only have a few ns, limiting the programs and actions (e.g., loops, floating-point operations, memory accesses) that can be applied to packets (either by the compiler for a specific switch or by not being supported in the P4 language at all). Therefore, while offering many advantages and flexibility, programmable switches cannot support all imaginable forwarding schemes without any performance loss (or at all). Consequently, algorithms deployed on the switches must be designed with these restrictions and limitations in mind.

Moreover, while both programmable switches and SDN focus on increasing the network programmability, they address two fundamentally different needs in networking and are, consequently, not exclusive, but complementary. Hence, programmable switches, and programming languages such as P4, were not designed, nor are intended to replace SDN. In contrast, as illustrated in Figure 1.4, an SDN controller will often interact with P4 programmable switches in the same way it would with traditional SDN switches instructing them on how to process the received packets by filling in a set of match-action tables. However, in contrast to traditional SDN switches, which were fixed-function switches, and as such, had a predefined set of tables (as well as header fields one could match on and actions one could execute), for programmable switches the programmer himself/herself can specify the tables, their order, the header fields to match on, and define custom actions to execute upon a match allowing for custom forwarding schemes and adaptability while keeping the advantages of an SDN controller.

In addition to this flexibility and agility, P4 programmable switches offer advanced monitoring features, such as the possibility to gather custom packet meta-data (e.g., queue depth, queuing delay). By combining this meta-data with custom forwarding schemes, switches can react immediately to events such as congestion while processing a packet - an option not available on traditional fixed-function SDN switches. Hence, switches can adjust to the current application needs (e.g., providing higher or lower QoS or re-routing to avoid congestion) on the fly based on the current network state. More importantly, this process can happen without the latency needed to contact the central controller each time (a situation common in many SDN deployments). In fact, the central controller can be programmed to guide the long-term behavior of the network, instructing the switches on what to do in certain situations (e.g., if the network is congested, the link is down) while the process of determining the current state, and the forwarding rules that need to be applied can be left to the switches.

In this thesis, to keep all the SDN's advantages, but decrease the reaction time, we use a combination of the programmable switches and the SDN concept. Hence, as explained above, we design networks in which the central SDN controller, in contrast to pure SDN frameworks, would only affect the network's general long-term behavior. To quickly react to specific events, such as link failures or congestion, the controller will offload part of the control to the programmable switches (that would execute custom for-

warding schemes), reducing the reaction time significantly and, consequently, provide per-packet QoS for each of the flows. Programmable switches promise more flexibility, agility, and the possibility of gathering and reacting to important packet meta-data (e.g., queue depth and queuing delay) immediately while the switch processes the packet [99]. This way, new forwarding schemes for different traffic types can be implemented and deployed to the switches instantaneously, isolating different applications and adjusting to their current requirements on the fly. Furthermore, deploying novel, custom algorithms (e.g., providing higher or lower QoS or re-routing to avoid congestion) makes it possible to target specifics of low latency applications directly in the data plane. Doing so reduces the reaction time significantly and, consequently, provides per-packet QoS for each of these flows.

1.3. RESEARCH QUESTIONS

In the previous section, we identified programmable switches as one of the main tools enabling low reaction times; however, despite their popularity, the specific means to control the network delay and provide the requested QoS to all flows inside a network remain largely undefined. Therefore, we formulate the main research question of this thesis as follows:

What mechanisms need to be developed and deployed in a programmable network to support low-latency applications?

First, we start by carefully investigating the limits imposed by programmable switches and techniques to overcome those limits by presenting best practices for designing data plane algorithms. Next, in light of these best practices, we develop combined control and data plane techniques to support the newer low-latency application domains alongside the traditional bandwidth-oriented applications. Finally, we show that our algorithms can enable networks to support various services with very diverse QoS requirements simultaneously.

In order to refine our research question, we identify three sub-questions that need to be addressed by any approach that seeks to satisfy QoS requirements for all flows belonging to a network:

1. What techniques can be used to overcome the challenges associated with programmable hardware when designing and deploying network algorithms?

2. How can we allocate network resources to different applications in real-time by taking into account their individual requirements?

3. How can we provide a fair resource distribution between different flows belonging to the same service, thus guaranteeing the same performance for all service users?

1.4. OUTLINE

In this thesis, we systematically investigate the above-mentioned sub-questions and provide corresponding solutions (Figure 1.5). In particular, we address the first sub-question

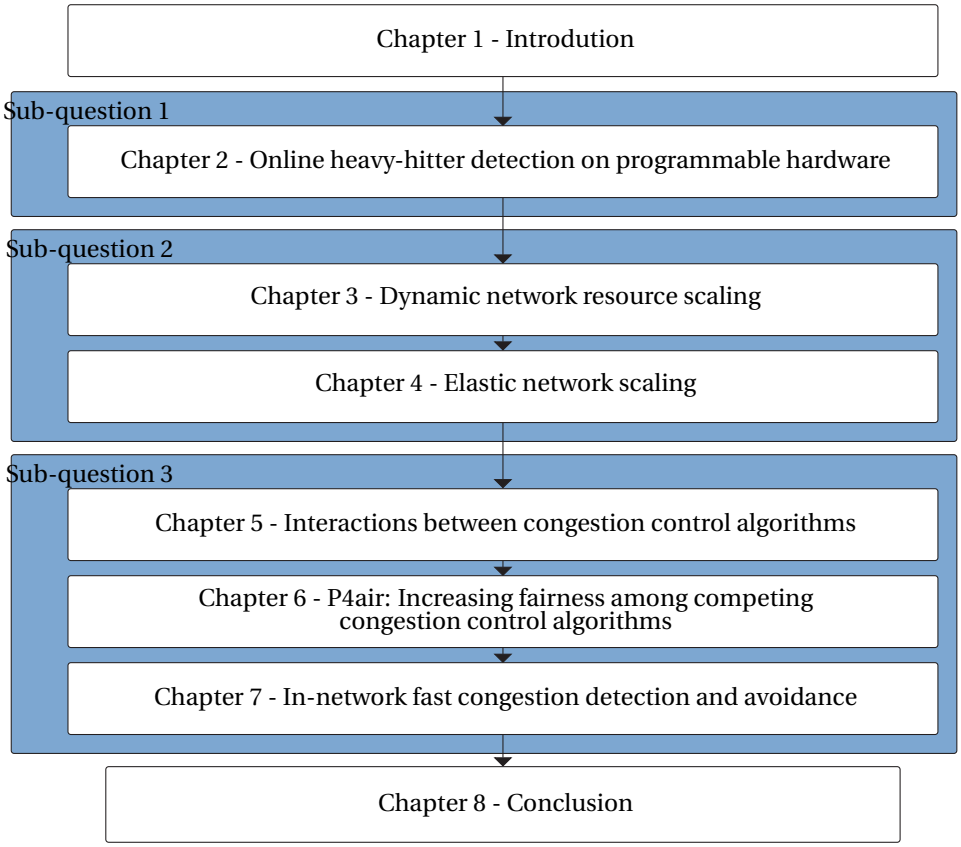


Figure 1.5: Dissertation outline, connection between main challenges (in blue) and chapters.

in Chapter 2 by developing a set of techniques to overcome programmable hardware limitations. Next, in Chapters 3-4, we address the second sub-question by exploiting the concept of network slicing. Finally, we take on the third sub-question in Chapters 5-7 by providing a different solution for two distinct transport choices: (1) transport protocols using a feedback loop (e.g., TCP/QUIC) and (2) transport protocols without a feedback loop (e.g., UDP).

In Chapter 2, we first identify the constraints of programmable hardware, imposed to ensure that a switch's performance will not degrade by deploying different programs—i.e., to ensure that a switch can always run at line rate. Next, using the example of a heavy hitter detection application (an application that determines if a flow has sent more than a given percentage of the last sliding window of N packets), we illustrate techniques to overcome the constraints of programmable hardware while minimizing the resource utilization of the switch.

Chapter 3 shows that many low-latency applications, such as remote teleoperation, have varying dynamics that often stay far below their highest value. We leverage this fact by designing a system in which, at any time, the current dynamics govern the amount

of network resources allocated to a flow and, consequently, the QoS experienced. In particular, we design a data plane scaling solution based on an application's input. Our solution modifies the switches' configurations on the fly by re-routing the flow and re-allocating the bandwidth assigned to the flow to meet the current application's needs.

In Chapter 4, we extend our previous scaling approach by designing a general elastic network slicing system. Our solution allows operators to provision a virtual network per service on top of a single (shared) physical infrastructure, with the ability to automatically provision or release assigned network resources based on the current traffic demands of the slice. We show that, by being deployed in the data plane, our system significantly reduces the reaction time and, as a consequence, can on-the-fly adjust to the changing traffic patterns. Finally, we show that due to a faster reaction, the assigned resources are more adequately matched to the current traffic needs, leading to less user traffic degradation and a higher QoS.

Chapter 5 discusses the interactions among different TCP/QUIC flows sharing the (potentially virtual) network introduced in the Chapter 4. We demonstrate that flows using different congestion control algorithms, or having different round-trip times (RTTs), may overpower each other; this results in unfair resource distribution. A subset of the flows usually claims most of the capacity. Consequently, we show that merely isolating different services (as in Chapter 4), without taking their flows' specifics into account, is not enough to guarantee good performance for all service users.

To solve the above-mentioned problems, in Chapter 6, we make use of programmable switches to enforce fairness from within the network itself, instead of from the congestion control algorithms running at the endpoints. Our solution continuously monitors the properties of all flows that pass through a switch and groups them based on the congestion control algorithms' behavior. Furthermore, it applies appropriate custom measures to suppress the aggressive flows and boost each group's smaller flows.

Finally, in Chapter 7, we focus on strict low-latency applications not using TCP/QUIC congestion control mechanisms and seek to ensure that packets are not dropped or delayed at any node in the network. Therefore, we deploy a custom data plane congestion control and avoidance solution in the forwarding nodes instead of at the source or via a central controller. To do so, we enable programmable switches to (1) track processing and queuing delays of latency-critical flows and (2) react immediately in the data plane to congestion by re-routing the affected flows. Furthermore, we show that our solution ensures per-packet QoS for applications, such as remote surgery, that cannot reduce their sending rate on demand.

2

ONLINE HEAVY-HITTER DETECTION ON PROGRAMMABLE HARDWARE

The previous chapter introduced programmable networks, showing that they enable more flexibility and agility by allowing operators to deploy custom algorithms directly on the network nodes. However, to run at line-rate, i.e., to process packets without any drop in throughput, the switches can assign only a limited number of processing cycles to each processed packet. Consequently, algorithms deployed on programmable network switches must adhere to stringent memory access rates, and limitations on the types of actions make many existing algorithms unusable.

This chapter illustrates techniques that can be used to overcome the above-mentioned restrictions; we demonstrate these techniques by developing an application to detect heavy hitters, i.e., large-volume flows that consume a considerable amount of network resources. In particular, we introduce (1) Modulo sketching, a novel counting scheme that reuses counters and limits the impact of smaller flows beyond early processing stages. We also describe (2) Sequential Zeroing, a new approach extending interval-based schemes to sliding window measurements. To the best of our knowledge, this is the first heavy-hitter detection solution that provides per-packet granularity at line-rate performance in programmable networks.

This chapter is based on a published conference paper: B. Turkovic, J. Oostenbrink, F.A. Kuipers, I. Keslassy, A. Orda, *Sequential Zeroing: Online Heavy-Hitter Detection on Programmable Hardware*, 2020 IFIP Networking Conference (Networking), 422-430 (2020) [100]

2.1. INTRODUCTION

This chapter is about providing the ability to detect, online, and at high line-rate, whether each packet going through a programmable switch is part of a heavy-hitter flow, which is a flow that has exceeded a threshold number of packets in the sliding window of the last N packets. For network operators, such an ability is crucial to enable fine-grained Denial-of-Service (DoS) mitigation, traffic anomaly detection, flow-size-aware routing Quality-of-Service (QoS) management, and load balancing [101–104].

Scope. We consider the problem of enabling per-packet heavy-hitter detection in the data plane of programmable switches, which use network programming languages such as P4 [105–108]. P4 defines registers, *i.e.,* stateful memory blocks that the switch can read from, modify, and/or write to, while packets are being processed. When deployed in a network, a P4 program works in coordination with a control plane, which configures the run-time match-action tables and action variables. P4-programmable hardware can be classified as having either (1) *local memory*, as in Intel Tofino [98], where packets are processed through a pipeline of several hardware stages. Each stage has its own separate memory and processing resources. To maintain a high processing throughput, typically only one read-modify-write action is allowed per register array; or (2) *shared memory*, as in Netronome [109], where concurrent memory accesses to the same register array are allowed. Since memory accesses consume most of the processing cycles in programmable hardware and may lead to race conditions, they should be minimized. In this chapter, we formulate our algorithms so they could be implemented in both hardware models.

2.1.1. MOTIVATION

Common heavy-hitter detection algorithms (*e.g.,* Space-Saving [110], Memento [101]), CSS [103], and WCSS [103]) which are optimized for low memory consumption in software implementations, violate programmable hardware constraints, as they are not organized in consecutive simple stages, require too many memory accesses per processed packet and/or use actions not supported by programmable hardware. Simple sketching schemes, like Count-Min (CM) [111], have no mechanism to count over the sliding window of the last N packets, and in fact do not even provide a clear implementation for counting over periodic intervals, as they do not contain a mechanism to simultaneously reset the whole data structure online [112]. Even existing data plane solutions that were developed for P4, such as HashPipe [102] and PRECISION [104], come with several limitations: most significantly, (1) they also have no mechanism to count over sliding windows; and (2) even when counting over periodic intervals, they cannot compute online a count estimate for each packet, unless they recirculate each packet through the pipeline twice, thereby halving the line rate (see Section 2.5.1). In addition, (3) they also do not provide a simultaneous memory flushing implementation for counting over periodic intervals; and (4) they intrinsically need flow identifiers for each counter, imposing a significant memory overhead.

2.1.2. MAIN CONTRIBUTIONS

We present a body of solutions for heavy-hitter detection on programmable network hardware, in which we aim at minimizing the false-positive and false-negative rates.

In Section 2.2, we start by considering the easier problem of detecting heavy hitters over a fixed *interval* of N packets. To do so, we introduce *Modulo Sketching*, a new sketching approach. Its most salient feature is that it relies on *conditional sketching*, a sketching approach that uses several consecutive stages of counter arrays and attempts to filter out the non-heavy-hitter packets by stopping them at the first stages. As a result, heavy-hitter packets are nearly the only ones to reach the last stages, thus (1) reducing potential collisions between different flows, which in turn reduces the need for a large memory size; and also (2) reducing memory access rates, since non-heavy-hitter packets almost never access later stages. Therefore, this approach is particularly adapted to the common pipeline structure of programmable switches. It stands in contrast to previous P4-based algorithms like HashPipe and PRECISION, which need to go through all stages in order to evaluate the size of a packet's flow (see Section 2.5.1 for details).

Next, in Section 2.3, we consider the case of heavy hitters over *sliding windows*, which is our main goal. We attempt to leverage our interval-based Modulo sketch by generalizing it to sliding windows. Unfortunately, unlike intervals, sliding windows also need deletions, which introduce additional memory accesses. In addition, implementing a perfect sliding window would consume a large portion of the limited switch memory, as it would require us to remember the full packet order. We thus suggest two different approaches to efficiently approximate a sliding window: (1) *Sequential Window*, which relies on control plane intervention, and (2) *Zeroing Window*, an data plane counter zeroing technique. We finally combine both to yield the *Sequential Zeroing* algorithm in the data plane.

Finally, in Section 2.4, we evaluate the performance of our new algorithms and demonstrate how they outperform existing approaches. We run our evaluations both through simulations and through experiments on a Netronome SmartNIC. In particular, we illustrate on CAIDA traces how our final schemes can achieve negligible false-negative rates and low false-positive rates while providing an estimation for each packet at line rate using the data plane, even when assuming a small memory consumption of 55kB.

2.2. INTERVAL MEASUREMENT

As a first step towards our goal of determining heavy hitters over the last N packets, we look at the easier problem of detecting heavy hitters over a fixed *interval* of N packets. In other words, our initial goal is to determine for each incoming packet whether its flow has exceeded a threshold number H of packets within this interval.

We would like to obtain a heavy-hitter detection scheme that satisfies two major criteria: (1) it should not consume too much memory and (2) it should have a low memory access rate. In addition, as commonly considered in the literature, we assume that false negatives (failures to detect heavy-hitter flows) carry a higher penalty than false positives [113, 114].

Conditional sketching. To satisfy these criteria, we introduce the concept of *conditional sketching*, a sketching approach that relies on several consecutive pipelined stages of

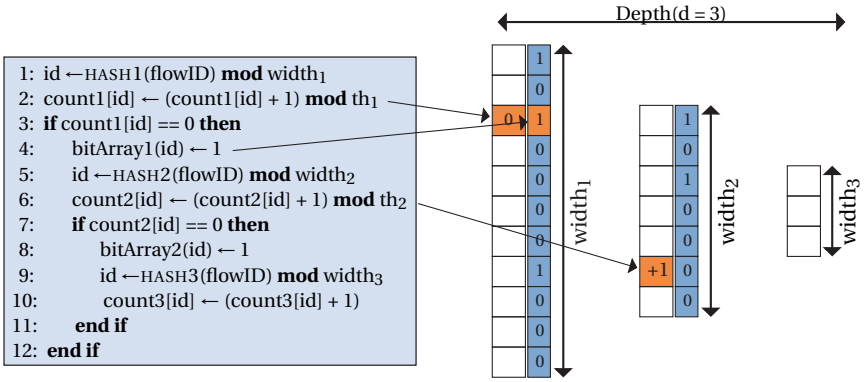


Figure 2.1: *Modulo* sketch with $d = 3$ stages. The flowID of an incoming packet first hashes into a first-stage (orange) counter and increments it. If the counter value is $th_1 - 1$ and it is incremented to $th_1 \equiv 0 \mod th_1$, we reset the counter, set its associated bit to 1, hash the flowID into a second-stage (orange) counter, and increment that counter as well. If the result is below the threshold th_2, we stop here. As the counter's associated bit is 0, there is no need to continue to the third stage, and we can evaluate flowID as *not* a heavy hitter.

counter arrays. Since we want to reduce the overall memory access rate, our key idea is to stop non-heavy-hitter packets in early stages. Thus the overall number of operations is significantly reduced, as heavy-hitter flows are nearly the only flows to reach the last stages.

2.2.1. MODULO SKETCH

We introduce the *Modulo sketch* algorithm to efficiently implement the concept of conditional sketching. As mentioned, each packet goes through several stages to update its counters. However, following conditional sketching, most packets will stop updating counters at the early stages of the stage pipeline. The packets may then continue over several additional stages without any counter update simply to estimate the packet's flow size.

Intuitively, Modulo sketch works like a clock, as it zeroes the counter of the *seconds* (first stage) when incrementing the counter of the *minutes* (second stage). It proceeds in the same way by resetting the *minutes* (second stage) when incrementing the *hours* (third stage), and so on. Thus, Modulo manages to increase its efficiency by both (1) stopping most counter updates in early stages and (2) reusing the counters.

Architecture. As Figure 2.1 illustrates, the Modulo sketch consists of d successive stages of counters, where d is the depth of the sketch. Each stage i holds $width_i$ counters of c_i bits each. We define the threshold th_i of each stage $1 \le i \le d-1$ such that

$$H > \prod_{i=1}^{d-1} th_i, \tag{2.1}$$

where H is the heavy-hitter threshold that was defined above.

In addition, to enable the conditional sketching, we introduce a bit array at each stage but the last, *i.e.,* each counter in stage $1 \le i \le d-1$ is provided an additional initially-null bit that determines whether the packet should continue to the next stage.

Algorithm. As shown in Figure 2.1, upon a packet's arrival, its flowID is first hashed onto a counter in the first stage. It then increments this counter. Next, for any stage $1 \leq i \leq d-1$ but the last, the first time that the resulting counter value reaches $th_i \equiv 0 \mod th_i$, we set its associated bit to 1, indicating that the threshold has been reached at least once, and therefore that all subsequent packets hashing to this counter should continue to the next stage $i+1$ in order to estimate their flow size. Therefore each packet goes through all stages with a set bit, until it reaches a stage where it hashes to a counter with a null bit, in which case it can stop. Since $H > \prod_{i=1}^{d-1} th_i$ (Equation (2.1)), any packet that stops before the last stage is considered as a non-heavy-hitter. More generally, the flow size of a packet that sees a counter value v_i at each stage i can be estimated as $v_1 + th_1 \cdot (v_2 + th_2 \cdot (\cdots + th_{d-1} \cdot v_d))$, which needs to be compared to H.

Threshold details. To fully utilize all the bits of all the counters, we define the threshold th_i of each stage $1 \leq i \leq d-1$ to be $th_i = 2^{c_i}$. We also allocate enough bits to the last-stage counters so that they never overflow even if a single flow uses N packets, i.e., $c_d = \left\lceil \log_2 \left(\frac{N+1}{\prod_{i=1}^{d-1} th_i} \right) \right\rceil$ bits per counter.

Properties. The main advantage of Modulo is its *reduced memory consumption* at high scales. If N packets are added to the sketch, at most N/th_1 packets reach the second stage to update it. More generally, at most $N/\prod_{j=1}^{i-1} th_j$ packets will reach stage i. Therefore we have an *exponentially decreasing* load further down the stages, yielding a particularly scalable architecture. For instance, when doubling the window size N and the heavy-hitter threshold H, Modulo would only need to double a single threshold. Again, we can increase the width of the first stages at the expense of smaller widths for the late stages.

Note that the Modulo sketch presents the drawback of allowing a small number of false negatives. Specifically, the packet that resets a counter is the one that increments the next-stage counter. For instance, a first flow F_1 may increment a first-stage counter to 63, but then the packet of another flow F_2 may arrive, reach the threshold of 64, reset this counter, and increment its hashed second-stage counter, thus in a sense *stealing* the entire counter value of 64.

2.3. SLIDING WINDOW MEASUREMENT

The goal of this chapter is to compute online heavy hitters over *sliding windows*. Since in the previous section, we considered the problem over *intervals* as a first step, we would now like to provide ways to generalize interval-based sketching schemes to sliding windows.

However, sliding windows involve both additions and deletions at each packet arrival, and therefore cause many challenges to overcome: (1) we want to delete the last packet from the counting structure without keeping in memory the list of packets, and therefore without remembering what the last packet is, as this would significantly increase the memory consumption and the number of memory accesses; (2) the logic of a conditional sketch-based structure like Modulo breaks down with deletions: *e.g.*, we mentioned above the example of a flow F_1 contributing 63 packets out of a counter threshold of 64, and another flow F_2 contributing the last packet and consequently in-

crementing its hashed second-stage counter. If we want to delete an F_1 packet from the structure, we would not know how to update the second-stage counters; (3) finally, the counter increments (due to packet arrivals) and decrements (due to packet deletions) are not allowed to occur in two different counters of the same stage, since there is a bound of one memory access per stage, and therefore there needs to be some scheduling of the memory accesses.

First, in Section 2.3.1 we generalize interval-based approaches to support the sliding window concept. Next, in Section 2.3.2 we propose a sliding window approach to reset the counting sketches directly in the data plane, while the packets are processed. Finally, we combine the advantages of the two previously described approaches to create a sliding window solution that can maintain accuracy over time without any intervention from the control plane.

2.3.1. SEQUENTIAL WINDOW

Architecture. As Figure 2.2 illustrates, given some integer parameter k, our *Sequential window* scheme periodically implements an interval-based sketching scheme such as Modulo or Count-Min of depth d every sub-interval of $\lceil \frac{N}{k} \rceil$ packets. Therefore, each window of N packets can be covered by at most $k+1$ consecutive interval sketches.

Algorithm. At the first incoming packet, we start by counting in the first counting sketch corresponding to the first sub-interval. Then, every $\lceil \frac{N}{k} \rceil$ packets, we keep advancing to the next counting sketch. When we are done with the last sketch, we can reset the first one using the control plane, since it counts packets that were received over N packets ago. Every $\lceil \frac{N}{k} \rceil$ packets, we then keep resetting the next sketch in the sequence to start a fresh sketch for the next sub-interval. Finally, to estimate a count for a given packet, we simply sum the estimates provided by all the sub-intervals (shown in green in Figure 2.2).

Threshold details. Since we scaled down the N-packet intervals to $\lceil \frac{N}{k} \rceil$-packet sub-intervals, we want to similarly scale down the thresholds, and therefore update Equation (2.1) by using a locally-scaled-down heavy-hitter threshold $\lfloor H/k \rfloor$.

Properties. The Sequential Window approach has several advantages. First, by avoiding any deletions and including enough sub-intervals to cover the entire sliding window of N packets, the Sequential Window scheme does not introduce additional false negatives.

Second, the value k helps to trade off the performance against the total number of stages, thus targeting different hardware platforms as well as different window sizes. For instance, increasing k will decrease the number of packets processed by each sub-interval, and therefore reduce the number of unique flows and the number of hash collisions in each stage, thus increasing accuracy. Also it will decrease the number of packets that are counted outside the window, and therefore further decrease the number of false positives. On the other hand, it will also need more stages in the implementation.

Third, the Sequential Window approach is easily implementable on any type of programmable hardware. As illustrated in Figure 2.2, it only requires up to $d \cdot (k+1)$ memory accesses per packet, i.e., $d \cdot (k+1) - 1$ reads and 1 read-modify-write. Thus, on hardware with shared memory, e.g. Netronome, the sketch can easily be tuned by the choice of k and d to avoid a drop in throughput. Moreover, for other types of hardware, it does not violate the one-access-per-stage rule.

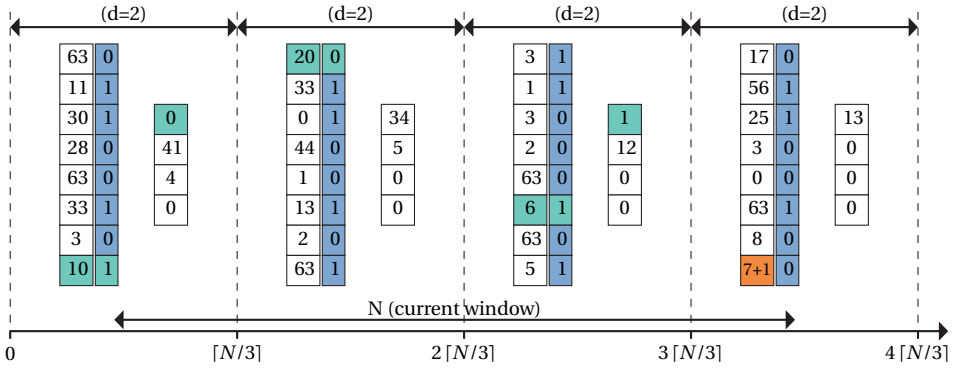


Figure 2.2: *Sequential Window* algorithm with Modulo sketch, using $k = 3$. The Sequential Window architecture comprises $k+1 = 4$ sub-intervals, each implementing a Modulo sketch with $d = 2$ stages for $\lceil N/k \rceil$ packets, thus covering the entire window of N packets. An incoming packet would only update its count in the last sketch (in orange), while reading and summing its count estimate from all 4 sketches (in green and red).

However, there are several disadvantages to this approach. First, the total number of consumed stages increases by a factor of $k+1$ compared to any interval-based sketch: $(k+1) \cdot d$ stages in total. Second, by counting over a larger window than the actual sliding window, we introduce false positives. Third, and most significantly, all the counters in the oldest sub-interval need to be reset at the same time *using the control plane*. Thus, this approach is not entirely done in the data plane. This control plane resetting may be an issue on a fast link with a small window. For instance, Intel Tofino switches can process packets at 6.5 $Tbps$. Assuming a window of $N = 2^{16}$ minimally-sized packets of 64B and $k+1 = 8$, then every 81ns the hundreds or thousands of counters of a sub-interval would need to be reset, which at best increases the processing resources from the control plane used by the algorithm and at worst is simply impossible, depending on the hardware platform.

2.3.2. ZEROING WINDOW

Overview. We now look for an alternative way of transforming an interval-based sketch like Modulo or Count-Min into a sliding-window-based sketch. A significant challenge is that we need to delete old packets that are not in the sliding window anymore, but on the other hand we do not want to allocate memory space to remember old packets. Instead, our first key idea is *to loop through all counters and zero them once every N packets*, thus ensuring that packets that have left the sliding window do not influence our counts anymore. In addition, our second key idea is to make sure that we can do it in the *data plane*, and do not require the massive intervention of the control plane anymore.

Initial algorithm. Our *Zeroing Window* algorithm is relatively straightforward. Consider a given interval-based sketching algorithm like Modulo or Count-Min. Then, for each stage i of width $width_i$ in the sketching algorithm, Zeroing Window defines a zeroing period $m_i = \lfloor N/width_i \rfloor$, and essentially resets the next counter at every m_i^{th} packet that is added to the sketch. Specifically, it resets counter j at packets $j \cdot m_i \mod N$. For

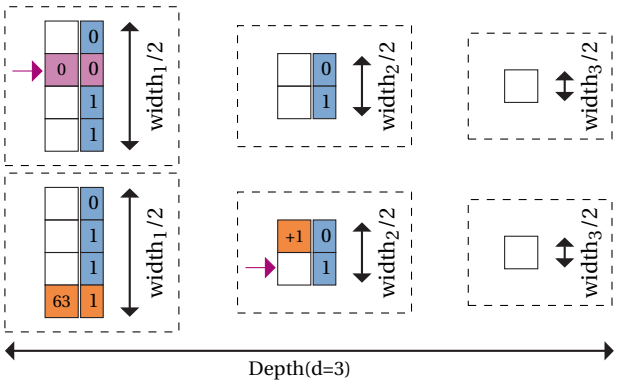


Figure 2.3: Zeroing Window for a Modulo sketch of depth $d = 3$ stages. Each stage is subdivided into 2 equal-size sub-stages (in dashed rectangles), and we are only allowed one memory access per sub-stage. In addition, assume we want to reset the counters with blue arrows. The incoming packet checks its hashed (orange) counter in the first stage, which is equal to the first-stage threshold $th_1 - 1 = 63$, and then resets it and increments a (orange) second-stage counter. As a result, we can reset the (purple) first-stage counter in the top sub-stage, since there is no memory access to this sub-stage. However, we cannot reset the second counter with an arrow, and therefore will wait for the next time that this sub-stage is not accessed by a packet.

instance, if $N = 2^{16} = 65,536$ and $width_i = 1,000$, it resets the first counter at packet 65 of the window, the second counter at packet 130, and so on, until the last counter at packet 65,000. It then resets the first counter again at packet $N + 65$, etc.

Data plane implementation. The above algorithm is simple, but it violates our rule that each stage should be accessed at most once per packet, since it may want to increment a counter as well as reset another one within the same stage. Therefore, as Figure 2.3 illustrates, we suggest a data plane implementation of the Zeroing Window algorithm. We split each stage into two equal-sized and independent sub-stages. Then, we want to apply the same zeroing scheme to each sub-stage. Assuming that the hashing functions are uniformly distributed, each sub-stage is only accessed at most half the time by the inserted packets. Therefore, whenever a counter needs to be reset, it can simply wait for the next time its sub-stage is free. In the worst case this waiting time is unbounded, and in the case where all packets are independent it is a geometrically distributed variable of expected value below 2. In practice, we never encountered any issue with this waiting time.

Properties. The main benefit of the Zeroing Window scheme is that it manages to operate in the data plane. However, because it resets arbitrary counters in the sketches, its disadvantage is that it also significantly increases the false negative rate, which we consider as more costly than the false positive rate. Therefore, we also consider several ways of reducing these false negatives for different sketching algorithms, at the expense of increasing the false positives:

Zeroing the Count-Min sketch. When removing values from the CM-sketch, or as in our case resetting some of the counters to 0, one should apply the median instead of the minimum [111]. For instance, if the $d = 3$ counts are 100, 110 and 120, and 120 is zeroed, the median estimate of 100 is more accurate than the minimum estimate of 0.

Zeroing the Modulo sketch. We now do not stop at the first time that a bit is set to 0

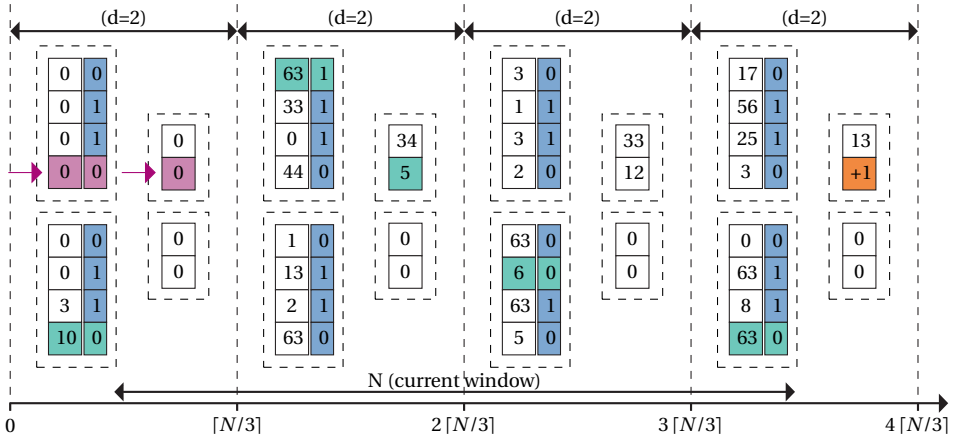


Figure 2.4: *Sequential Zeroing* scheme, which applies the Zeroing Window technique to the last sub-interval of the Sequential Window. We use the example of Figure 2.2 with $k+1=4$ sub-intervals, each implementing a Modulo sketch of depth $d=2$. Again, each stage is subdivided into 2 equal-size sub-stages (in dashed rectangles). Assume we want to reset the last (blue) counters of both of the top sub-stages in the left sub-interval that contains the oldest packets. We can indeed do so without violating memory constraints.

(Modulo sketch), but estimate the flow size using the counts over all stages.

2.3.3. SEQUENTIAL ZEROING: ZEROING THE SEQUENTIAL WINDOW.

We finally introduce a last scheme, denoted *Sequential Zeroing*, which combines the Zeroing approach with the previously-described Sequential window. As Figure 2.4(b) illustrates, Sequential Zeroing removes outdated flow counts from the last sub-interval of Sequential Window by applying the Zeroing Window algorithm to this sub-interval. Namely, it can do it automatically in the data plane, without any intervention from the control plane, by splitting its stages into sub-stages and applying the scheduling described above.

2.4. EVALUATION

2.4.1. EXPERIMENT SETUP

We conducted our evaluations, first using simulations in Python, and then experiments with a Netronome Agilio CX SmartNIC [109].

Hashing. The 5-tuple flow identifier (flowID) consists of the source IP, destination IP, layer 4 protocol, source port, and destination port. All our implementations used the CRC16 hash function. Different hash functions were created by appending seed values to the flow identifiers.

Traces. We classified heavy hitters as flows whose packet counts were above a threshold $H = \lfloor N/1000 \rfloor$, initially considering an interval size of $N = 2^{16}$ packets. Packets were obtained from (1) 40 different traces collected from an ISP backbone link at the Equinix data-center in Chicago in January 2016, made available by CAIDA [115], and (2) 10 different traces collected from university campus data centers (UNI1 and UNI2 dataset), made available by [116]. We observed similar results using both datasets, even though

they have significantly different flow-size distributions (the university traces have more heavy hitters), and therefore only present the CAIDA. In the hardware evaluation, all the traces were replayed at the rate of N packets per second.

Metrics. We evaluated all schemes on the percentages of false negatives (percentage of heavy hitter packets that are not reported) and false positives (percentage of non-heavy hitter packets that are reported). As previously mentioned, following literature, we assume that the penalty of false negatives is significantly higher than that of false positives. We also measured the distribution of the absolute count estimation error.

Comparison baselines. For interval-based evaluations, all of our solutions were compared against the following baseline solutions: (1) *Count-Min (CM)* sketch [111], (2) *HashPipe* [102], (3) *HashPipeMod* which implements HashPipe using 2B flowID fingerprints rather than 13B flowIDs (see Appendix), (4) *HeavyKeeper* [117], and (5) PRECISION [104]. For the sliding-window evaluations, since we are not aware of any other P4 solution that implements sliding windows, we considered a set of solutions that combine all interval-based solutions with a periodic resetting of all the counting stages every N packets. In all evaluations, we fixed a given allocated total memory for a fair comparison.

2.4.2. INTERVAL MEASUREMENTS

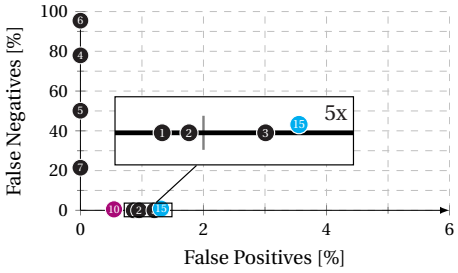
False-positive vs. false-negative. Figure 2.5a shows the false-positive vs. false-negative rates of all schemes given 15kB of memory and an interval of $N = 2^{16}$ packets. HashPipe and our modified HashPipeMod with flowID fingerprints exhibit many false negatives, which we try to avoid (note that they would also run at half the line-rate). Count-Min (CM) performs better, especially with $d = 2$ stages. Our Modulo scheme performs best, especially with $d = 2$.

Memory vs. performance. Figure 2.5a and 2.5b show the performance of our solutions for two different memory quotas. As expected, assigning more memory improves the accuracy for all solutions, as the width of the counter arrays can be increased, which reduces the number of hash collisions. We can see how for small memory allocations, other approaches start to break down, while the Modulo sketch manages to achieve acceptable heavy hitter detection. The main reason is a more efficient memory usage: the total number of counters used in our scheme is much higher than in the other approaches, since our counters do not need to count up to N and therefore are smaller. This memory efficiency makes our solution uniquely suitable for programmable hardware.

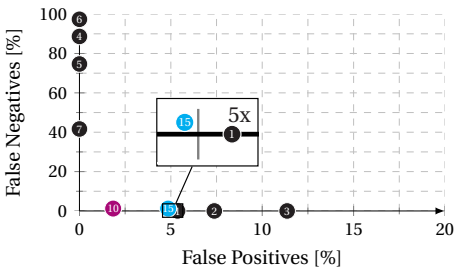
Tuning of Modulo. Choosing a higher value of th_1 that filters more flows and prevents them from reaching the last stages of the pipeline improves accuracy (Figure 2.5c). Also, most counters should be placed in the first stages, and only a few of them in the last stages (Figure 2.5d). This way, due to larger widths in early stages, the probability of collisions in the first stages is also lowered and the number of false positives reduced. However, this comes at a cost: the probability of a collision between two flows reaching the last stages is increased, leading to an increased probability of high count over-estimation (Figure 2.5f).

Interval sizes. In contrast to other schemes, Figure 2.5e shows that Modulo performs very well with longer interval sizes, suffering only from a small decrease in accuracy (0.12

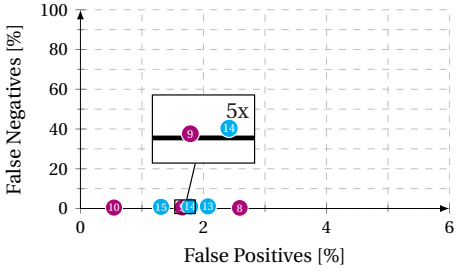
Related work: Count-Min: ❶ d=2 ❷ d=3 ❸ d=4 ❹ HashPipe(d=6)
❺ HashPipeMod(d=6) ❻ PRECISION(d=2)
❼ HeavyKeeper(b=1.08, d=2)
Modulo(d=2): ❽ widths={1,0.1}, th_1=16 ❾ widths={1,0.1}, th_1=32
❿ widths={1,0.1}, th_1=64 ⓫ widths={1,0.5}, th_1=64
⓬ widths={1,0.05}, th_1=64
Modulo(d=3): ⓭ widths={1,0.1,0.01}, th={8,8} ⓮ widths={1,0.1,0.01}, th={16,4}
⓯ widths={1,0.1,0.01}, th={32,2}



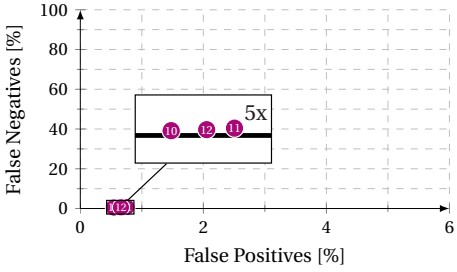
(a) Memory 15kB.



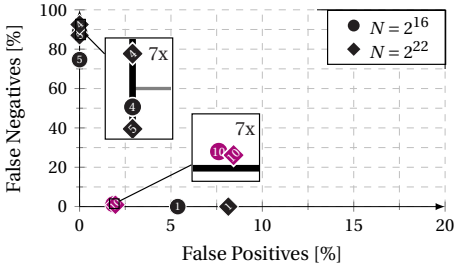
(b) Reducing memory to 5kB.



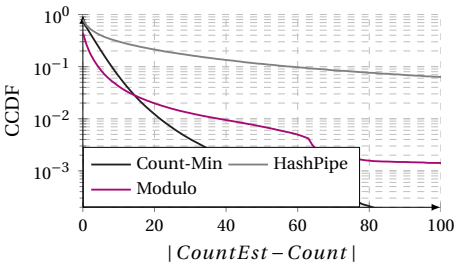
(c) Impact of thresholds. Memory 15kB.



(d) Impact of widths. Memory 15kB.



(e) Impact of interval size. Memory 5kB.

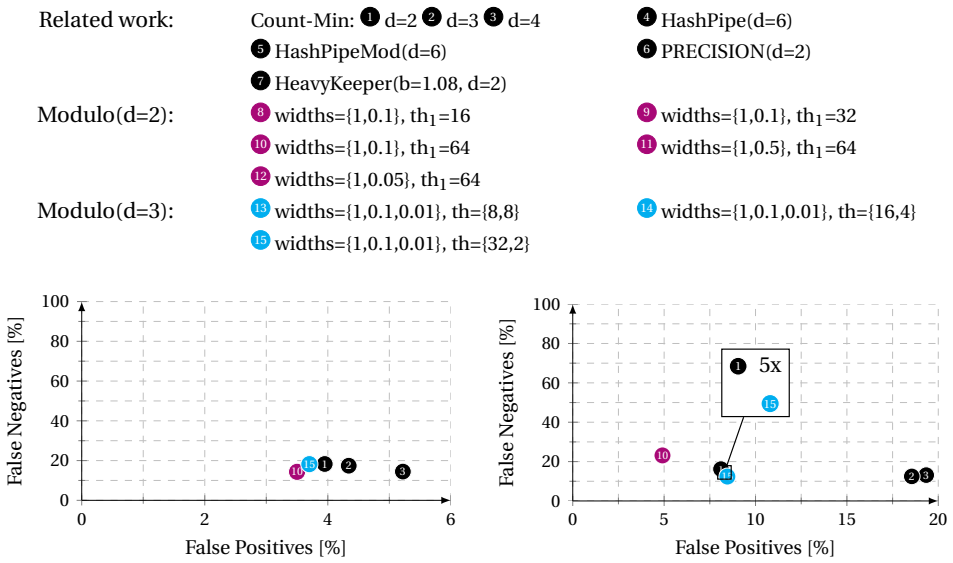


(f) CCDF of count estimation error. Memory 15kB.

Figure 2.5: *Interval-based* simulation of all schemes using 40 different CAIDA traces.

percentage points of false positives) given the same memory consumption ($5kB$).

Count estimation error. Figure 2.5f shows the complementary cumulative distribution



(a) Memory 15kB.

(b) Memory 5kB.

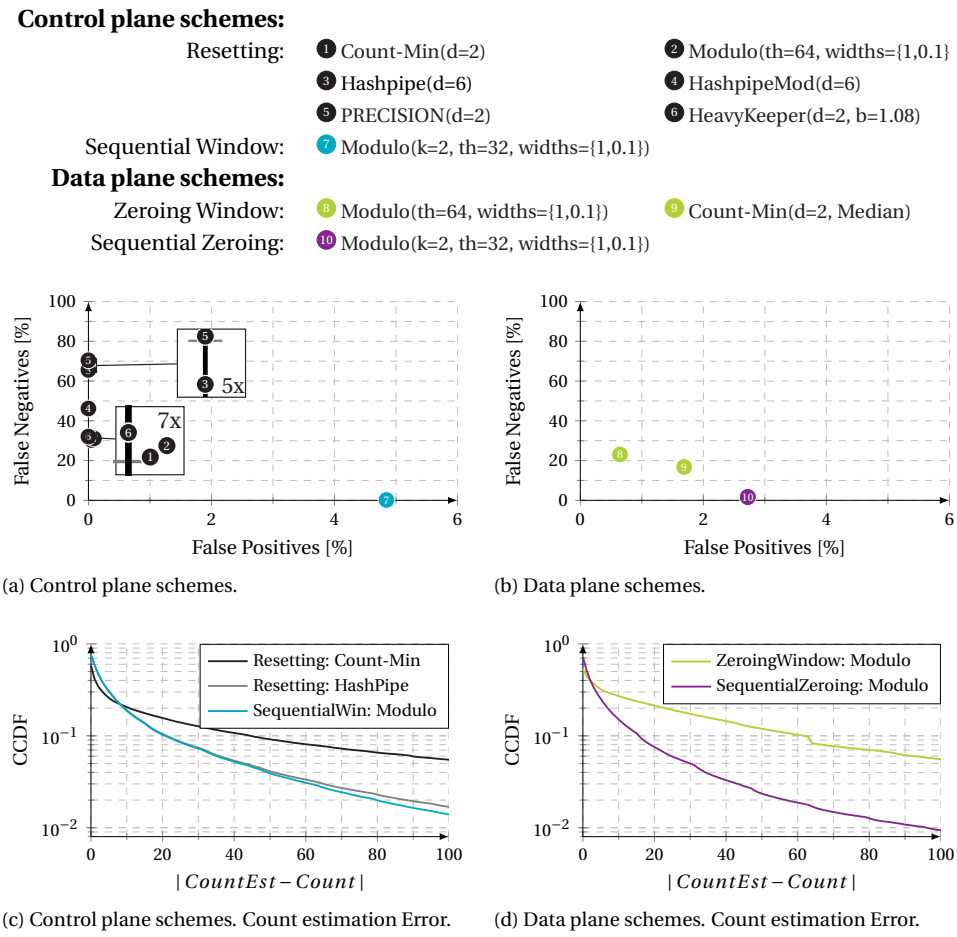
Figure 2.6: Netronome SmartNIC hardware experiments for *interval-based* schemes using 40 different CAIDA traces.

function (CCDF) of the absolute count estimation error, i.e., the probability of exceeding a given value. HashPipe is clearly outperformed by all other approaches, which is also reflected by its large number of false negatives (see Figure 2.5b and 2.5a). Count-Min has a higher probability than Modulo of being mistaken in the flow count estimation, but a slightly smaller probability of being significantly mistaken (by more than 17).

Hardware experiments. Figures 2.6a and 2.6b shows run experiments on Netronome SmartNIC, using the same settings as the simulations of Figures 2.5a and 2.5b, respectively. We assume that the control plane takes time resetting intervals, and runs a full reset of all counters each time in a somewhat naive way. As expected, we find that this indeed impacts the performance of all schemes. For example, on Netronome SmartNICs, the speed of the control plane is the main limiting factor. Intuitively, resetting the interval counts is not immediate, i.e., while an RPC call is initiated every second for every array to correspond to N packets, these actions are not executed instantaneously, resulting in an increased number of false negatives. However, other schemes, such as Hashpipe and PRECISION, cannot provide an online count estimate (see Section 2.5.1). Similarly, as programmable hardware does not support floating point operations, nor loops to implement fixed point math, HeavyKeeper cannot be implemented.

2.4.3. SLIDING WINDOW MEASUREMENTS

Control plane solutions. Figure 2.7a and Figure 2.7c show that our Sequential Window outperforms all resetting solutions that rely on an interval-based scheme and reset it periodically. Simply periodically resetting interval-based schemes yields signifi-



(a) Control plane schemes.

(b) Data plane schemes.

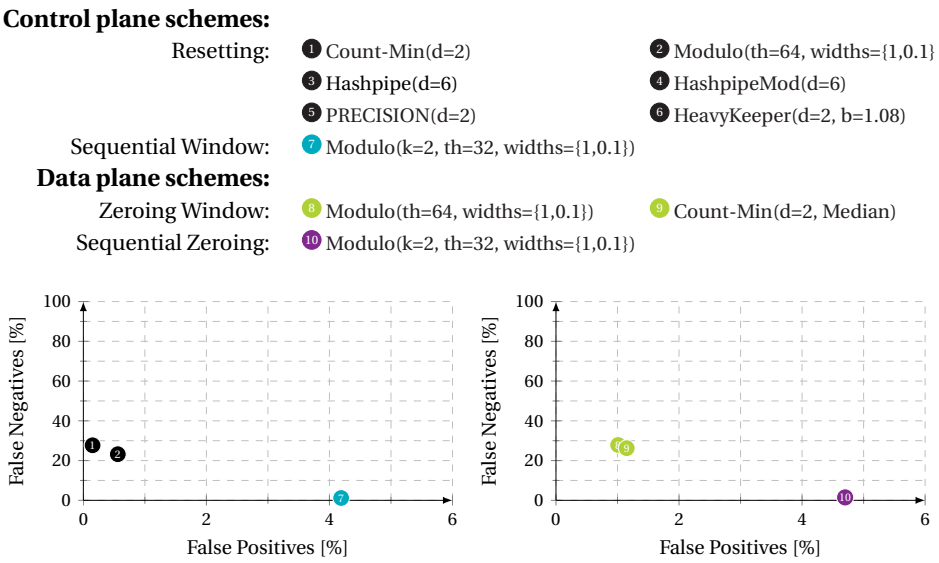
(c) Control plane schemes. Count estimation Error.

(d) Data plane schemes. Count estimation Error.

Figure 2.7: *Sliding window* simulation using 40 different CAIDA traces. Memory 55kB.

cant false negative rates, because at the start of each interval these schemes do not take into account packets that appeared previously. In contrast, the *Sequential Window* approach that implements several sub-intervals of interval-based schemes yields significantly lower false negative rates, yet the false positive rates are slightly higher (4.19% with just $55kB$).

Data plane solutions. Figure 2.7b shows that our Zeroing algorithm, which periodically resets each counter, in combination with Modulo attempts to reach some compromise between false positives and negatives, but still yields a non-negligible rate of false negatives. However, in doing so it always outperforms a solution that resets all counts using the control plane, by almost halving the percentage of false negatives. Moreover, Sequential Zeroing, which combines our two window approaches, Sequential window and Zeroing Window, by applying the Zeroing Window approach to the last sub-interval of Sequential Window, outperforms all other schemes, by dramatically lowering the per-



(a) Control plane schemes.

(b) Data plane schemes.

Figure 2.8: Netronome SmartNIC experiments for *sliding window* schemes using 40 different CAIDA traces. Memory 55kB.

centage of false negatives (from over 20% to 1.67%) while keeping a reasonable false positive percentage. Packets from the extra sub-interval are gradually removed, and the count estimation error, as a result, is reduced (Figure 2.7d).

Count estimation error. Looking at the CCDF of the count estimation error (Figure 2.7c and Figure 2.7d), we find that the Sequential Zeroing schemes perform best again, even though they do not need control plane intervention.

Hardware experiments. Figures 2.8a and 2.8b show the results of our experiments on a Netronome SmartNIC, using the same settings as the simulations of Figures 2.7a and 2.7b, respectively. Again, our new schemes significantly outperform the others: the Sequential window is able to maintain high accuracy while our Sequential Zeroing approach displays a negligible rate of false negatives and only a slight increase in false positives, compared to other simulations. These slight differences may be due to race conditions in hardware. In contrast, similarly to the interval based measurements, approaches, such as the HashPipe and PRECISION cannot provide an online count estimate (see Section 2.5.1) and, since programmable hardware does not support floating point operations, nor loops to implement fixed point math, HeavyKeeper cannot be implemented.

2.5. RELATED WORK

Algorithms relating to heavy-hitter detection can be divided into three groups: (1) *sampling* algorithms, (2) *sketch-based* algorithms, and (3) *counting* algorithms.

Sampling algorithms. Sampling algorithms, such as NetFlow [118], Sflow [119], Sample&Hold [120], are currently widely deployed and used by network operators. In these

algorithms, nodes usually maintain current flow statistics that are periodically sent to a remote point for further analysis. However, they do not determine for *each* packet whether it belongs to a heavy hitter, which is needed for fine-grained control.

Sketch-based algorithms. Sketch-based algorithms such as ours use specialized data structures called sketches that hash and count all packets in the switch hardware. In exchange for some count overestimation or underestimation, this approach can achieve a considerably lower memory usage, which makes it especially suitable for programmable hardware. Unfortunately, existing sketch-based algorithms (e.g., Cold Filter [121], UnivMon [122], Count-Min Sketch [111], Count Sketch [123], Probabilistic lossy counting [124], CountMax [125], Elastic sketch [126], HeavyKeeper [117]) were not designed with P4-programmable switches in mind, and often cannot be directly implemented without modifications or loss of accuracy. For example, to estimate a count of an item, Cold Filter calculates a minimum of d hashed counters in each stage, violating the constraint of one memory access per register array present on modern programmable hardware. Moreover, other meta-algorithms, such as Elastic sketch that relies on the Count-Min sketch, are orthogonal to our approach and could benefit from using our sketches with higher accuracy.

Counting algorithms. Counting algorithms (HashPipe [102], PRECISION [104], Space-Saving Algorithm [110], CSS [103]) maintain a data structure consisting only of heavy-hitter flows and corresponding counts. The Space-Saving algorithm requires either maintaining a sorted list or finding an item with the minimum counter value. Unfortunately, both are either not supported by existing programmable hardware or exceed the available processing budget. CSS uses TinyTable [127], which also violates the available processing budget. Hashpipe [102] is explained in the Section 2.5.1 and PRECISION [104] is similar. Both were designed for P4, but cannot operate at line-rate.

Sliding window approaches. (WCSS [103], SWAMP [128, 129], Memento [101]) remove the oldest entries from the counting data structure so that only information about the last N processed packets is present at the switch. SWAMP [128, 129] maintains an additional array with flow identifiers from the last N packets. Every time a new packet arrives the oldest entry from the array is removed and replaced with a new flow identifier. However, depending on the selected window size, memory consumption is very high. Ben-Basat et al. present two different solutions in [101, 103] optimized for memory consumption with constant query time. However, their use of TinyTable [127] is unsuitable for programmable network hardware.

2.5.1. HASHPIPE

HashPipe [102] consists of d consecutive stages, each with its own counter array and its own hash function. In addition to flow counts, HashPipe also stores the corresponding flowIDs (see Figure 2.9).

Upon a packet's arrival, its flowID is hashed to produce an index and compared to the flow identifier $flowID_1$ currently stored at that index in the first stage. If the identifiers do not match, the $flowID_1$ and count are *evicted* from the first stage and replaced by the new flowID and count 1. If the identifiers do match, the count is simply increased by 1. When a flow identifier $flowID_i$ (and count) is evicted from stage i, HashPipe will try to store it in the next stage $i+1$ by following the same process until the last stage is

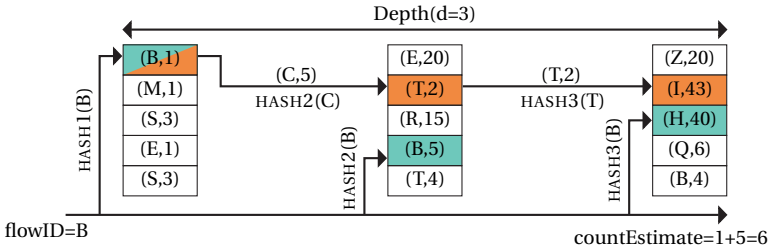


Figure 2.9: HashPipe algorithm. Based on [102].

reached.

In addition, to compute the count estimate, counts of all matching pairs from each stage $(flowID, *)$ need to be summed. Figure 2.9 illustrates this: flow insertion uses the red counts, while the count estimation uses the blue counts. As each array can only be accessed once, each packet would need to go through the pipeline twice to get an estimate for each packet, halving the throughput. Since PRECISION is similar to HashPipe, it suffers from the same throughput reduction.

For this chapter, we implemented two versions of HashPipe: (1) the original algorithm storing the full flow identifier (13B for a 5-tuple); and (2) *HashPipeMod*, a modification of the original algorithm that we introduced for a fairer comparison. HashPipeMod stores a fingerprint of the flow identifier (2B) instead of the full flow identifier. Thus, HashPipeMod has lower memory consumption than HashPipe, at the cost of introducing false positives (since a non-heavy-hitter flow may obtain the same fingerprint as a heavy-hitter flow).

2.6. CONCLUSION

In this chapter, we introduced the first heavy-hitter detection algorithm for programmable switches that provides per-packet granularity at line-rate performance.

To do so, we first introduced the *conditional sketching* technique that filters most small flows in early stages, and illustrated it by developing an interval-based sketching algorithm called *Modulo* sketch. Next, we addressed the problem of enabling such conditional sketching to work over *sliding windows*. Specifically, we started with the *Sequential Window* algorithm that is based on sub-intervals and needs the control plane. We then presented the *Zeroing Window* technique that periodically resets each counter in any interval-based sketch and which works fully in the data plane. Last, we combined both techniques to obtain the data plane *Sequential Zeroing* scheme. In our evaluations, we showed how our techniques significantly improve the accuracy of our estimation when compared to several baseline algorithms inspired by the literature, and implemented our schemes on a Netronome SmartNIC.

In this chapter, beyond bringing sliding-window heavy hitter detection to the data plane, we introduced techniques, such as (1) zeroing through ping-ponging the memories, (2) sequential windows, and (3) counter reuse through modulo, that we believe can also benefit data plane applications in general. Hence, by introducing these techniques, we have addressed the first challenge, as described in Sec. 1.3 of this thesis, and by build-

ing upon them in the following chapters, we are able to design more complex algorithms for programmable data planes.

3

DYNAMIC NETWORK RESOURCE SCALING

In the previous chapter, we discussed techniques that can be used to overcome certain hardware-related limitations when deploying algorithms in the data plane. In this chapter, we focus on data plane algorithms that enable low-latency applications such as the Tactile Internet. To do so, we first consider the network requirements of typical tactile applications (such as remote tele-operation); we show that these requirements, such as latency and bandwidth, fluctuate over time rather than remaining static. Consequently, statically assigning network resources to support these services at their peak is wasteful and would lead to low utilization. However, allocating fewer resources than an application needs at its peak would violate its requirements, degrading the user experience in potentially critical moments.

To optimize the resource utilization, we leverage the application's dynamic behavior to design a system in which the current perceived dynamics govern the number of network resources allocated to the flow—and, consequently, the quality of service (QoS) experienced by each user. In particular, we design a data plane scaling solution that, based on an application's current requirements, modifies the switches' configurations on the fly by rerouting the flow and re-allocating the bandwidth assigned to it, ensuring that the current application's needs will be met.

This chapter is based on a published conference paper: K. Polachan, B. Turkovic, T.V. Prabhakar, C. Singh, F.A. Kuipers, *Dynamic Network Slicing for the Tactile Internet*, 2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS), 129-140, (2020) [130]

3.1. INTRODUCTION

As introduced in Chapter 1, the Tactile Internet represents a low-latency application domain that enables users at different physical locations to interact with each other as if they were in the same room. For example, one subset of Tactile Internet applications involves human operators controlling remote robots, called teleoperators, over a network. In this scenario, the communication network transports the kinematic (position/velocity) commands from the operator-side to the teleoperator-side and feeds back audio, video and haptic data in the reverse direction. As a result, human operators can see and feel what the result of their actions is. This allows them to behave and react in the same way they would in a non-remote scenario and effectively transports their unique set of skills (e.g., surgeon's skills, Figure 3.1) over a network to remote locations.

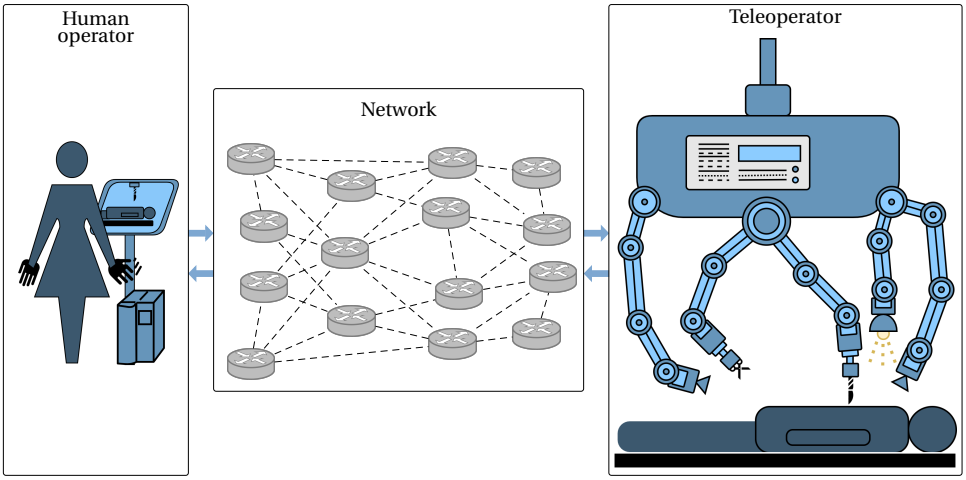
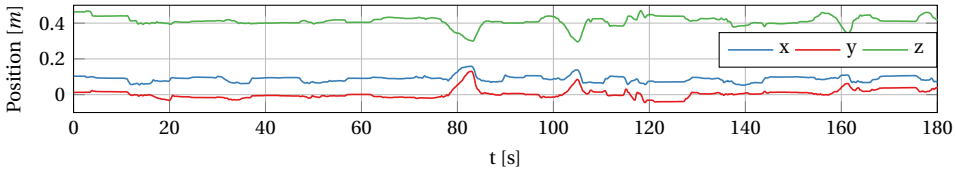


Figure 3.1: Tactile Internet application in which a human operator is controlling a remote robot, called teleoperator, over a network.

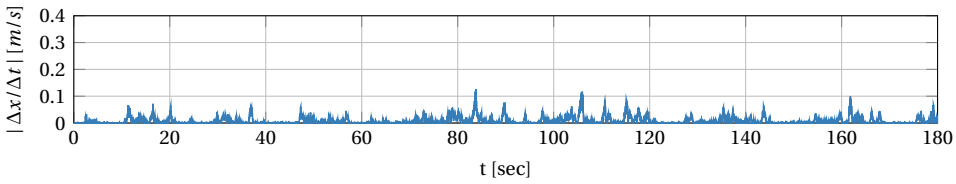
3.1.1. MOTIVATION

To guarantee a transparent experience, especially at higher operator's dynamics (e.g., higher hand speeds), Tactile Internet applications can have very stringent network requirements, such as extremely low-latency (order of a few ms), low jitter, high reliability and high bandwidth (order of $Gbps$) [131, 132]. At the same time, for many Tactile Internet applications, operator dynamics widely fluctuate and, for most of the time, stay away from their peak value [130]. Consequently, at lower dynamics the network requirements of these flows can be relaxed as well, allowing for higher latencies and less bandwidth. For instance, consider Figure 3.2 showing the left-hand movements of a surgeon performing a suturing operation using a da Vinci Surgical System [133]. The operator's dynamics (i.e., the hand speed of the surgeon) along the x axis vary throughout the procedure, but stay below their peak value (≈ 0.13m/s) most of the time.

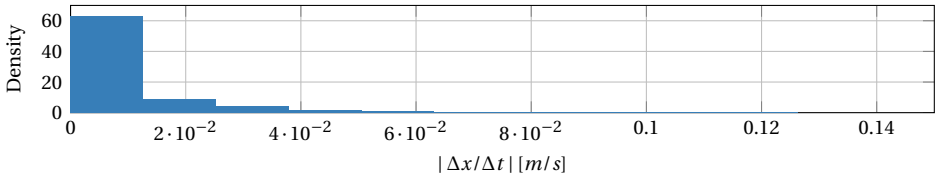
If the network provider were to statically allocate network resources to such an application based on the peak value the application might require at some time, performance



(a) (x, y, z)-positions tracking the left-hand movements of a surgeon performing the surgical task called suturing using a da Vinci Surgical System.



(b) Dynamics in x-position.



(c) Histogram of the dynamics in x-position.

Figure 3.2: Position tracking, dynamics and the histogram of the dynamics of a Left-hand movements of a surgeon performing a suturing operation using a da Vinci Surgical System.

would be guaranteed during the lifetime of the flow. However, due to the dynamic nature of these applications, allocated resources would be underutilized most of the time (Figure 3.2). However, if the network operator would reserve less and at any moment throughout the lifetime of the application, the available network resources would not able to match the application demands, significant end-to-end latencies and/or packet drops could occur. As a result, remote teleoperation systems could become unstable or result in severe operator-side cybersickness [134], an effect which occurs when the feedback signals are noticeably delayed, resulting in physical and physiological effects in the operator that prevent her/him from an extended use of the teleoperation system [135, 136]. Furthermore, for critical applications, such as telesurgery, in which a surgeon operates on a remote patient, such side-effects could have significant consequences, potentially resulting in injuries or even the death of the patient. Hence, trials that were performed with teleoperation systems usually involved a guaranteed network bandwidth reserved solely for this purpose (e.g., Lindbergh Operation [137, 138]) and/or networks that had a stable performance (e.g., low jitter, stable available bandwidth [138]). Therefore, although teleoperation systems have been around for decades,

the fact that the network operator must support the application at its peak as well as their criticality, restrict their large-scale deployment over public networks, especially in situations in which human operators need to perform control actions that demand higher operator dynamics (e.g., complex and/or critical surgeries) and/or cover large distances [134, 139].

3.1.2. CONTRIBUTIONS

As a way to provide high utilization in the network while maintaining a high quality of service for the above-mentioned applications, this chapter introduces a dynamic resource allocation scheme in which the operator's dynamics govern the amount of network resources allocated to the application's *network slice*, i.e., a part of the network allocated to a specific application and tailored to its requirements.

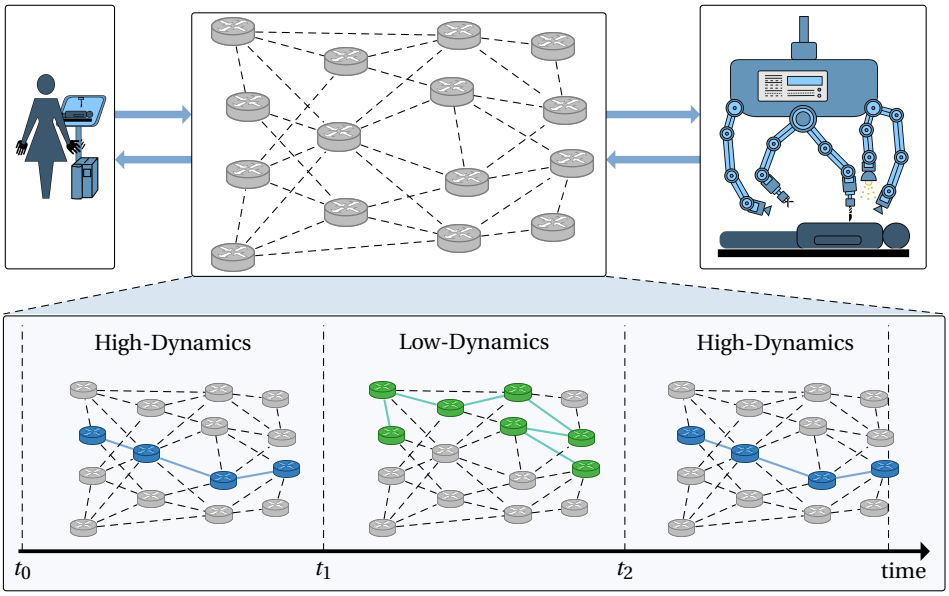


Figure 3.3: Dynamic network slicing. A tactile application uses a network slice provisioned for high dynamics (shown in blue). At some point, t_1, the network dynamics change, and the previously used blue slice is destroyed. Next, at some point t_2 the application dynamics increase, and the blue slice is provisioned, while the green slice is destroyed.

In particular, our solution routes each Tactile Internet flow through a set of network slices (each tailored to different dynamics) that are created and destroyed on the fly. Hence, at any moment, the amount of network resources reserved for an application matches its current dynamics and its corresponding requirements. Figure 3.3 illustrates this solution for a Tactile application that uses two slices: one provisioned for high dynamics (blue slice) and one provisioned for low dynamics (green slice). First, the Tactile application uses a network slice provisioned for high dynamics (shown in blue). At some point, t_1, the network dynamics change, and the previously used blue slice is destroyed. At the same time, the green slice is created, and the application starts utilizing it. To

determine the set of network slices and the corresponding network requirements that a Tactile application can use, we leverage the clustering algorithm introduced in [130]. This algorithm clusters the operator's dynamics based on historical data and maps each cluster to a resource vector (e.g., maximum latency, minimum bandwidth). The complete system overview is explained in more detail in Section 3.2.

Moreover, to reduce the time needed to destroy and create a new slice, we use a Software-Defined Networking (SDN) controller to pre-compute the paths for the identified slices and P4-programmable switches for real-time resource allocation and switching of slices. This way, by offloading these latency sensitive tasks to the switches, we significantly reduce the switching time (which could otherwise adversely impact Tactile applications in potentially critical moments, explained further in Section 3.2.2). Finally, in Section 3.3, we show that our approach leads to a more efficient network resource utilization but also to savings in aggregate switch memory while keeping the slice switching latency in control.

3.2. SYSTEM OVERVIEW

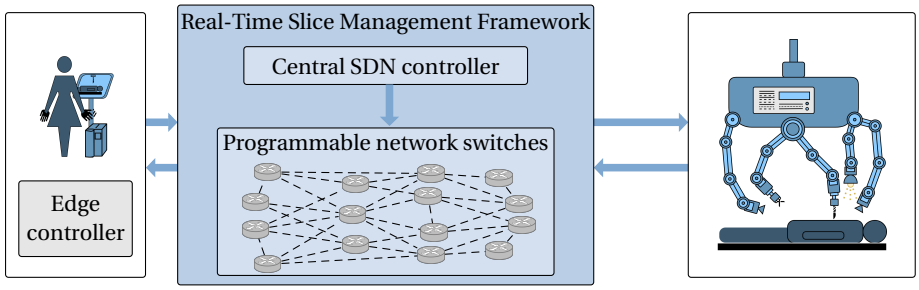
Figure 3.4a shows the main building blocks of a solution to enable dynamic resource scaling: (1) the edge controller and (2) a real-time slice management framework. The edge controller and all the associated processes (see Figure 3.4b and Figure 3.4c, shown in gray) were developed as part of a collaboration project and are not part of this thesis. More details about them can be found in [130]. In contrast, blocks (and processes) shown in blue were developed as part of this thesis. Moreover, these blocks can be used independently of the above-mentioned edge controller as long as there is a component at the applications side that performs comparable actions, as explained in Section 3.2.1.

3.2.1. EDGE CONTROLLER

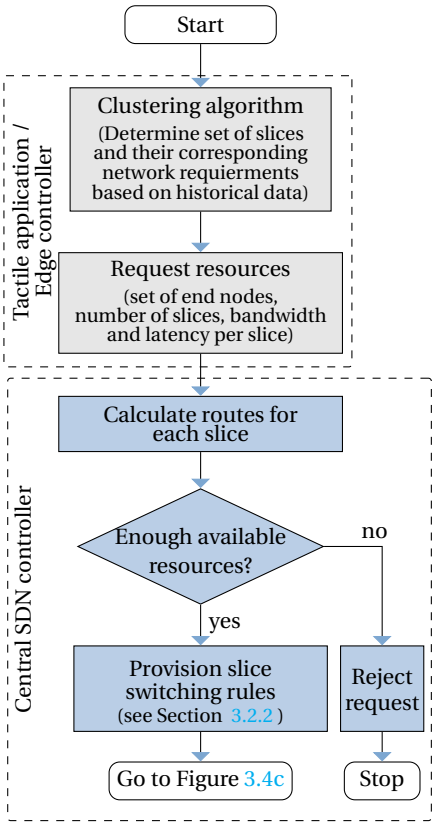
The main purpose of the edge controller is to determine the requirements of the processed flow and inform the network of their changes. To do so, it performs two main tasks:

- Before a new tactile flow starts, the edge controller residing at the operator side requests a set of network slices of specific bandwidth and latency from the network (in particular, from the central SDN controller, see Figure 3.4b). To do so, in this chapter, we used the custom unsupervised clustering algorithm that does so based on the available historical trials of the Tactile operation in question, and the dynamics-to-slice mapping described in [130]. However, any algorithm that would produce a similar set of slices to be forwarded to the central SDN controller can be used.

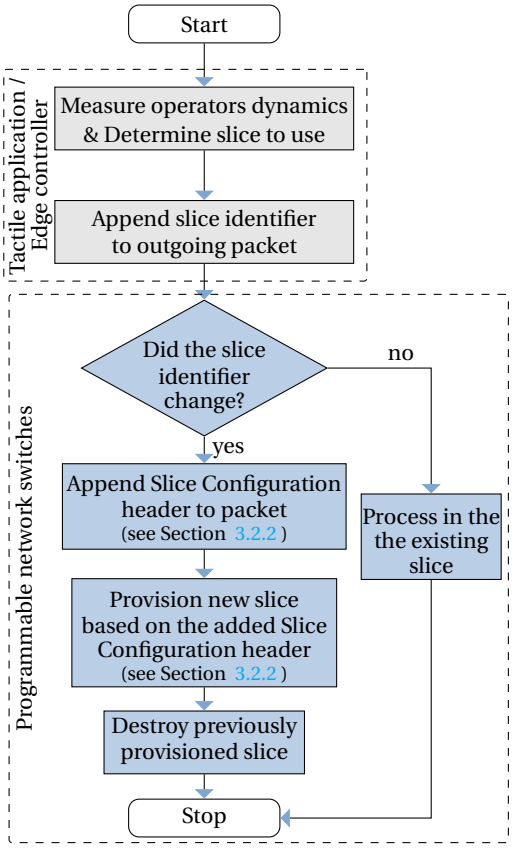
- Through the duration of the Tactile flow, the edge controller monitors the operators' dynamics and for each generated packet, decides which slice to use. Finally, it informs the network by appending this information to the outgoing IP packets by, for example, modifying the Type of Service (ToS) value. In this chapter, we use the system described in [130]. However, as mentioned above, any algorithm that tags the outgoing IP packets based on the application's requirements can be used.



(a) Main building blocks.



(b) Actions taken before the start of the Tactile flow.



(c) Actions taken per each packet throught the runtime of the Tactile flow.

Figure 3.4: Main building blocks and flowchart showing the main processes and interactions between them that enable dynamic network resource scaling. Blocks shown in gray were developed as part of a collaboration project and do not represent the authors contributions in this thesis. Blocks shown in blue are part of this thesis.

3.2.2. REAL-TIME SLICE MANAGEMENT FRAMEWORK

The second block, the real-time slice management framework enables on-demand provisioning of network resources, i.e., the resources are made available on the fly only when needed, similarly to computing resources in cloud environments. This way, network utilization is constantly high, as no resources are over-provisioned. At the same time, the QoS is guaranteed during the whole lifetime of the Tactile application, i.e., the network is adapting its behavior to match the current Tactile application's needs. The scaling process is done at the expense of any other non-Tactile flows, which get the remaining resources in the network. To enable this on-the-fly network resource management, our slice management framework consists of two main components:

- *Central SDN controller* (control plane) that has a global view of the whole network, as well as the currently present traffic. For every new Tactile flow, the central controller finds the appropriate routes that satisfy the end-to-end slice requirements according to the current global network state.

- *Slice configuration protocol* (data plane), deployed using the network programming language P4 [140], that creates/destroys the slices on-the-fly in the data plane using the information from the edge controller and the pre-computed inputs from the SDN controller. It provides a fast update loop, enabling the switches to react quickly to the changes in the application dynamics without the need to contact the SDN controller. Thus, the slices are created/destroyed at run-time, with negligible latency overhead.

Every time a new Tactile flow is initiated, the edge controller forwards the slice specifications (e.g., bandwidth, latency) it wishes to use to the central SDN controller (Figure 3.4). With its up-to-date overview of the current network state, the SDN controller uses these slice specifications to calculate the routes that satisfy the QoS (e.g., latency and bandwidth) requested for each slice. Finally, these pre-calculated routes are forwarded and stored in the first switch on the path (i.e., the edge switch to which the Tactile Edge Device is connected) and ready to be used by the second component of our solution: the *slice configuration protocol* (blue processes shown in Figure 3.4b).

The slice configuration protocol enables the creation/destruction of network slices on the fly, directly in the data plane, using just the routes stored in the edge switches. Whenever the Tactile dynamics change, i.e., the edge controller appends a different slice identifier to the outgoing packet, the assigned network slice (with a specific latency and bandwidth constraints) is destroyed, and the resources are freed to be used by other services. At the same time, a new slice, corresponding to the new dynamics, is created (blue processes shown in Figure 3.4c). To ensure that both creation/destruction actions are executed in real-time, our protocol *enables the data-packets to program the data plane as they pass through the switches.*

For example, to create the slice, the first switch appends a special *slice configuration header*, containing, among other things, the pre-calculated route to the original Tactile packet that was received from the edge controller (as explained further in Section 3.2.2). By reading this special header, every switch in the path configures/updates its forwarding and bandwidth reservation rules to correspond to the currently requested slice (as

explained further in Section 3.2.2). Thus, as the switches process the first Tactile packet, a new slice is configured and ready to be used (blue processes shown in Figure 3.4c). All the subsequent Tactile packets follow this first packet and are routed using the newly created rules, which prevents any temporary inconsistencies. In addition, every time a new slice is created, a similar packet is sent to destroy the slice that was previously used. Every switch that processes this packet deletes the configured rules and frees the allocated bandwidth. This way, the number of installed rules is minimized and the bandwidth released to be used by other services. Every time the edge controller detects a new change in dynamics, the whole process repeats.

During the lifetime of a Tactile application, at any moment, only one slice per application is configured/used in the switches. Thus, although the rules (processing as well as bandwidth reservations) for multiple slices are calculated by the controller, only one set of rules (corresponding to the currently used slice) is active. As a consequence, resources assigned to slices that are not currently used are free and have no impact on other traffic present in the network.

When calculating the routes, the SDN controller makes sure that, if the resources of other slices are to be requested (due to a change in dynamics), they would be available instantaneously. To do so, the controller keeps track of the amount of bandwidth allocated on every link to all slices of Tactile flows. Hence, new Tactile flows are only admitted through switches that will have enough resources available. As a consequence, during the entire duration of any Tactile flow, resource availability is always guaranteed. However, this approach also limits the maximum number of Tactile flows that can be present in the network at the same time. Depending on the exact Tactile application, this requirement can be relaxed by providing a trade-off between the maximum number of flows and the probability of a QoS degradation.

SLICE CONFIGURATION PROTOCOL

To switch a Tactile flow f, between two Tactile endpoints, from slice A (with a pre-calculated route r_A) to slice B (with a pre-calculated route r_B) two actions need to be performed: (i) creation of a new slice B, i.e. updating the forwarding rules and allocating bandwidth for flow f on all switches on path B, and (ii) deleting the old slice A, i.e. deleting the rules and freeing the allocated bandwidth for flow f on all switches on path A. To ensure that (i) and (ii) are executed in real-time, we designed a new slice configuration protocol that enables the data-packets to create/destroy a network slice as they pass through the switches.

To perform the above-mentioned actions, our slice configuration protocol uses two different messages (shown in Figure 3.5): (1) "Slice setup" message to create a new slice and (2) "Slice delete" message to delete the previously used slice. The field *Ports array* represents the pre-calculated route r_B (r_A) as a sequence of output ports from all the switches on path B (A). The size of this field depends on the number of switches used on the path (specified by the field *Header Length*). *Slice ID* corresponds to the bandwidth constraint needed on the path. Based on this parameter, the switches will reserve the needed amount of bandwidth as the packet passes through them.

In Section 3.2.2, we describe how the pre-calculated routes (by the central SDN controller) are used in the edge switches, while in Section 3.2.2 we describe how we use

our protocol to change routing entries on the intermediate switches (i.e., in the network core).

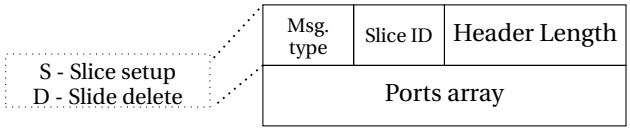


Figure 3.5: Slice Configuration Protocol header.

PROCESSING ON THE NETWORK EDGE ROUTER

When a first packet belonging to a Tactile flow is received at the network edge, the router, based on the flow identifier (source and destination IP addresses, protocol field, and source and destination ports) and the ToS (Type of Service) field, inserts a new "slice setup" header between the Ethernet and the IP headers. For example, as shown in Figure 3.6, when a packet with flow identifier 35 is received, the router checks the table containing the headers for all the potential slices this flow can use (shown on the left in Figure 3.6, for memory overhead calculation see Section 3.2.3).

Since the ToS field of the packet is set to 1, the slice with ID 1 is chosen and a header containing the values $1-4$ is inserted between the Ethernet and IP headers. The header field 1 represents the $Slice\ ID$ (Figure 3.5) and corresponds to a certain predefined amount of bandwidth that will be allocated at all the switches for this flow. Similarly, the next header (value 2) represents the length of the route r_B and indicates that two intermediate switches are present between the Tactile Edge Devices. All other values represent port numbers used at the intermediate switches (port 3 at the first intermediate switch and port 4 at the second intermediate switch). In addition, the Ethernet type is changed to a specific value (0xBB) to indicate the presence of the new header.

To delete the previously used slice, the same procedure is used. An additional packet, containing "Slice delete" header is sent on the route r_A (determined by the flow identifier and the previously used ToS field).

PROCESSING IN THE NETWORK CORE

When a packet with the *Slice Configuration Protocol* header is received by the next switch in the path, additional bandwidth is reserved, and the forwarding table updated (Figure 3.7). The first intermediate switch, after processing the "Slice setup" header, inserts a new entry in the forwarding table (shown on the left bottom corner in Figure 3.7). All the subsequent packets, with flow identifier equal to 35 and belonging to slice 1 (indicated by ToS equal to 1), are processed by this rule and output to port 3. In addition, the resources reserved for slice 1 are increased by 10 units of bandwidth (shown on the right bottom corner in Figure 3.7), the used port number field (with the value 3) removed, and the header length reduced by 1 to represent the number of switches left to configure. Similarly, when this packet is received by the second intermediate switch, a new rule (to output packets to port 4) is inserted and the bandwidth allocation table updated.

All subsequent packets with the flow identifier 35, are processed by these newly installed rules (e.g., output to port 3 on the first switch and port 4 on the second switch),

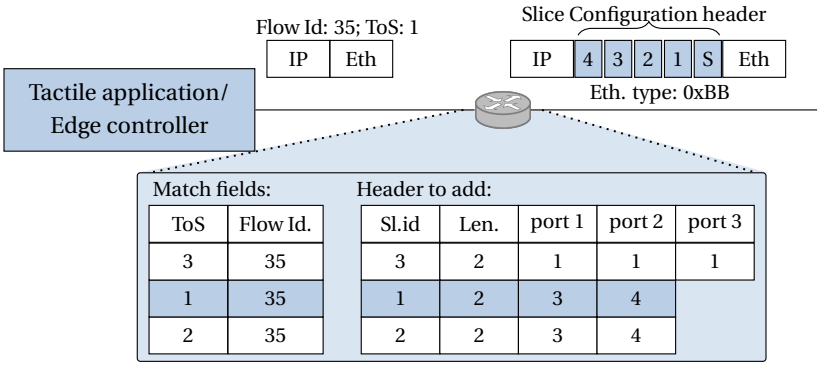


Figure 3.6: Processing on network edge. "Slice setup" header is inserted between the Ethernet and IP headers.

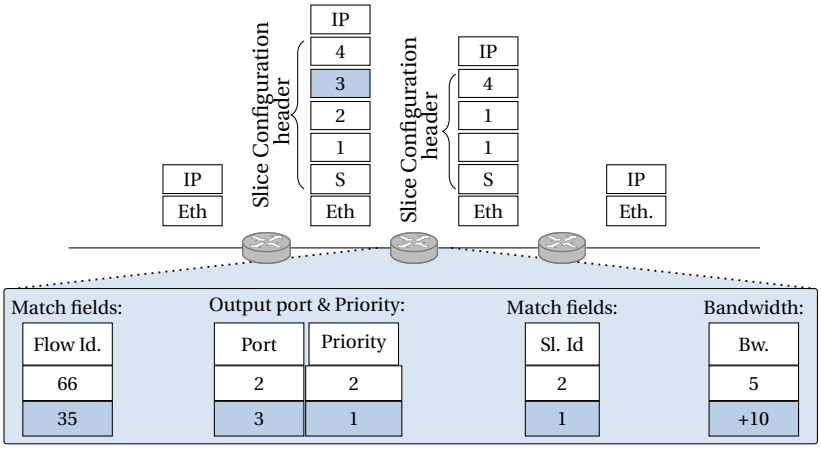


Figure 3.7: Processing of a "slice setup" message in the network core. New rules are added as the packet passes through the switch.

preventing any temporary inconsistencies. When a slice needs to be deleted, a similar process occurs on every switch in the network. The only difference is that, when a switch detects a "slice delete" header, network resources would be released (or scaled down) and processing rules removed.

3.2.3. NETWORK SWITCHING OVERHEAD

Delay Overhead. The contribution to the "switching overhead" by the network is represented by the additional transmission delay from all the switches due to the increase in packet size of the first packet processed by the new slice. In the case of a slice setup message, this overhead is equal to (3.1). Here S_{hdr} is the size of the Slice Configuration Protocol header (except the *Ports Array* field), n is the total number of switches in the path, R_x is the speed of the output link of switch x and S_{port} is the size in bits used to represent one port (usually $8bits$).

$$t_{network} = \sum_{1 \le x \le n} \frac{S_{hdr} + (n-x) \cdot S_{port}}{R_x} \tag{3.1}$$

Memory Overhead. To support such a system, an additional table containing all the possible headers that can be added needs to be maintained for every Tactile flow at the edge switch. This overhead for one Tactile flow can be calculated using (3.2). Here n_{slice} is the number of slices, S_{flowID} is the size of the chosen flow identifier, usually the 5-tuple (source IP, destination IP, source port, destination port, transport protocol). However, the memory consumption is reduced on all the other switches in the path (compared to a solution where rules are reconfigured in all the switches), as the number of rules needed in the core switches to process a Tactile flow is reduced to 1 from n_{slice}.

$$M_{network} = \sum_{1 \le x \le n_{slice}} \left(S_{port}\,(n+1) + S_{flowID} + 1 \right) \tag{3.2}$$

3.3. EVALUATION

3.3.1. EXPERIMENT SETUP

To evaluate our slice switching protocol, we emulated the USNET topology (Figure 3.8), using the Mininet emulator with the P4 software switch (behavioral model [141]). Multiple Tactile flows were generated between two Tactile Edge Devices TE_1 and TE_2 and delay, jitter, and throughput were measured in both directions. After receiving a packet, TE_2 bounced it back to TE_1 using the same slice. Tactile flows were routed through 4 slices with RTTs equal to $117.4ms$, $29.5ms$, $13.4ms$, and $5.7ms$ (determined by the clustering algorithm specified in [130]). Therefore, for each Tactile request, a set of four slices was calculated by the SDN controller (example set shown in Figure 3.8). To simulate the traffic belonging to other services, additional TCP traffic was generated using iperf between different switches in the network. All measurements were repeated 30 times.

Traces. Each Tactile trace was 98 seconds long and contained data from the da Vinci Surgical System database. For each packet in this trace, the ToS header was set using the clustering algorithm described in [130] and indicated the slice the packet should be processed in. Packet lengths were fixed to $140B$ (including all the headers) and sent at a rate that depended on the latency of the current slice (one packet each $RTT/2$). Thus packets belonging to slices with stricter latency requirements (and higher dynamics) were also sent at a faster rate, creating sudden bursts of Tactile traffic in the network.

Comparison baselines. Our approach (P4 + Slicing) was compared to (1) an approach that uses an SDN controller to compute and install a new slice (both route and bandwidth reservation) each time a switch occurs (SDN + Slicing), and (2) an approach that does not use slicing, but provides QoS guarantees (No Slicing) by reserving either the maximum or average bandwidth needed by the flow.

3.3.2. SWITCHING DELAY

To demonstrate the advantages of our slice configuration protocol, we evaluated the time needed to switch between two different slices. This switching delay was measured as the difference in the delay between the first (with the additional Slice Configuration

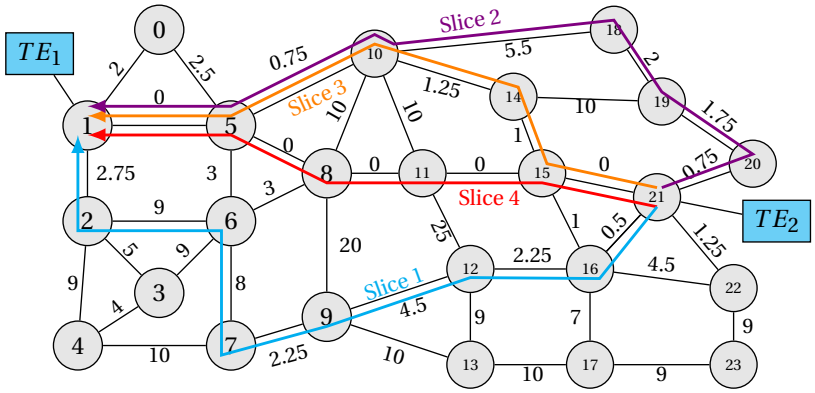


Figure 3.8: USNET topology. Link delays are in *ms*. TE blocks represent Tactile end-hosts.

header) and the second packet processed by the slice. Compared to the solution that uses a centralized controller to reroute packets (SDN + Slicing), our solution was able to reduce the switching delay significantly (0.34*ms* on average compared to 72.68*ms* on average). The main reason for the significant increase in performance is the fact that the (SDN + Slicing) solution forwards the first packet to the controller introducing a significant delay penalty, while our solution enables the packets to program the data plane.

Table 3.1: Scenario without any additional TCP traffic. Maximum and average RTT values measured for each slice for the USNET topology. (metrics calculated for 30 different runs). Values are in *ms*.

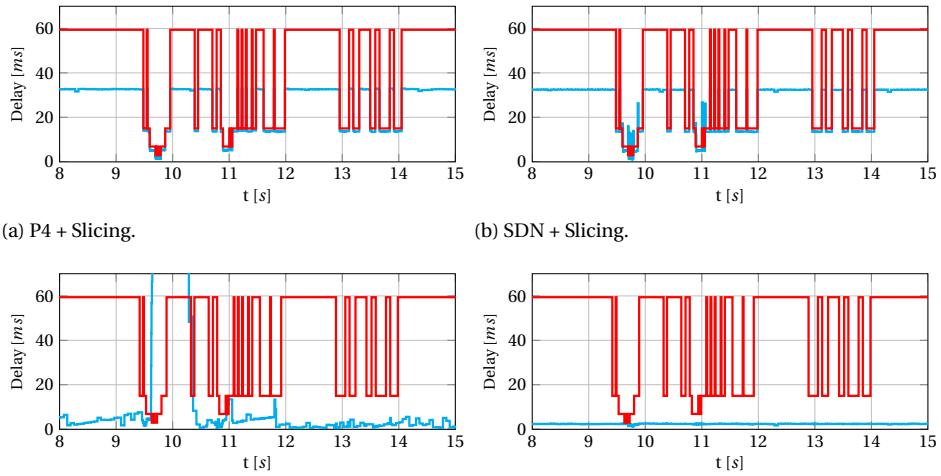
		P4 + Slicing		SDN + Slicing		No Slicing		No Slicing	
	Slice RTT constraint	Avgerage RTT	Maximum RTT	Avgerage RTT	Maximum RTT	Avgerage RTT	Maximum RTT	Avgerage RTT	Maximum RTT
1	117.48	65.37	67.00	32.50	44.13	-	-	-	-
2	29.51	27.46	29.28	29.28	36.13	-	-	-	-
3	13.48	10.71	12.77	5.77	24.75	-	-	-	-
4	5.75	3.43	5.74	5.86	21.46	4.53	18.38	-	-

Table 3.2: Scenario with additional TCP traffic. Maximum and average RTT values measured for each slice for the USNET topology. (metrics calculated for 30 different runs). Values are in *ms*.

		P4 + Slicing		SDN + Slicing		No Slicing (Max. Reserved)		No Slicing (Avg. Reserved)	
	Slice RTT constraint	Avgerage RTT	Maximum RTT	Avgerage RTT	Maximum RTT	Avgerage RTT	Maximum RTT	Avgerage RTT	Maximum RTT
1	117.48	63.07	66.24	63.10	67.64	-	-	-	-
2	29.51	27.65	29.39	39.28	91.92	-	-	-	-
3	13.48	9.92	12.67	57.31	998.74	-	-	-	-
4	5.75	4.11	5.21	67.64	781.76	2.051	15.93	138.88	1539.11

3.3.3. PERFORMANCE GUARANTEES

Figure 3.9 and Tables 3.1 and 3.2 show that, with our proposed (P4+Slicing) solution, the network can guarantee the performance at any moment during the duration of the



(a) P4 + Slicing.

(b) SDN + Slicing.

(c) No Slicing (average bandwidth reserved).

(d) No Slicing (maximum bandwidth reserved).

Figure 3.9: Influence of external traffic: Observed one-way delay ($RTT/2$) for 7 seconds of the Tactile flow when 4 slices are used for the USNET topology shown in Figure 3.8. Blue line represents the $RTT/2$ measured between two Tactile end-host, while the red line represents the $RTT/2$ constraint of the current used slice determined based on the current application dynamics.

Tactile flow. The blue line, representing the one-way delay between the two Tactile end-hosts (TE_1 and TE_2) was under the red line, representing the delay constraint of the system corresponding to the current perceived dynamics (i.e., the maximum allowed delay each packet can experience), during the whole flow duration. When switching from a high to low RTT slice, packet reordering sometimes occurred at the Tactile end-host, due to the difference in the slice's RTTs. However, this will not degrade the system's performance, as the later-arriving, outdated packets can simply be discarded.

In contrast, in the scenario (SDN+Slicing), the first packets processed by the slice experienced a significant increase in delay (Figure 3.9). This effect was more significant for low-latency slices (Slices 3 and 4), in which the delay constraints were violated for almost every packet. Due to a higher packet rate, packets arrived at the switches faster than they were processed by the controller, resulting in more packets being forwarded to it (as the switches are stateless and hence unaware that a packet was already forwarded to the controller until the new route is configured), thereby flooding it. This results in significant packet reordering at the Tactile end-host.

In case (No Slicing) was used and resources corresponding to the maximum possible dynamics (Slice 4) were reserved, the delay observed by the Tactile flows was the lowest possible (except for the first packet that is forwarded to the SDN controller and used to setup the route). However, when we decreased the reserved bandwidth to match the average amount of resources used by the other two solutions (*P4+Slicing*, *SDN+Slicing*), *No Slicing* behaved the worst among all the analyzed solutions, by having the highest average, as well as maximum RTT (see Table 3.1). Especially in moments in which high QoS

was required (Slices 3 & 4), and the packet rate was high, Tactile packets were queued due to insufficient resources thereby violating the RTT constraints (Figure 3.10).

3.3.4. BANDWIDTH UTILIZATION

In our proposed (P4+Slicing) solution, scaling of the reserved bandwidth happens when the packet (having a different ToS set, indicating to switch slices) was processed at each switch. Thus, the amount of the bandwidth reserved for Tactile traffic corresponded to the current packet rate (see Figure 3.10). Similarly, in case (SDN+Slicing) was used, the SDN controller matched the reserved bandwidth to the one required by the current slice. However, in case of (SDN+Slicing) the reconfiguration delay in addition would depend on the delay between the switches and the controller, potentially decreasing the performance in cases when the controller is not run on the same machine as the switches. In case (No Slicing) is used, the resource utilization is either constantly low (max. reserved) resulting in significant over-provisioning, or insufficient to account for the high dynamics (avg. reserved) resulting in noticeable degradation to the end-users.

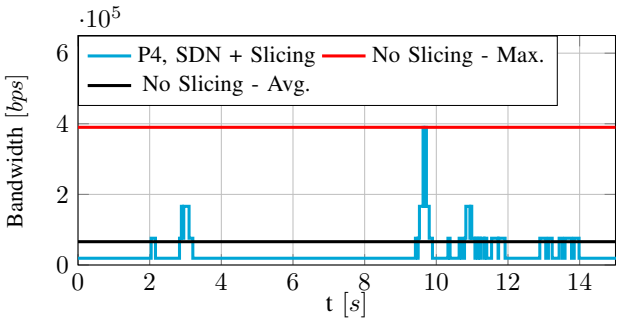


Figure 3.10: Reserved bandwidth. (P4+Slicing) and (SDN+Slicing) are reserving the same amount of bandwidth.

To conclude, our (P4+Slicing) solution is able to satisfy the latency constraints of the Tactile flow during the whole duration of the flow. In addition to guaranteeing the performance of a Tactile flow, by dynamically routing the traffic through multiple paths and by scaling the resources, it also minimizes the resources that need to be assigned by the network provider and maximizes the utilization of the resources assigned to it.

3.4. RELATED WORK

Traditional networks were static and offered very limited possibilities for Quality of Service (QoS) provisioning. When assigning resources, network providers were left with two choices: to either over-provision, i.e. reserve too many resources, but keep the average utilization low, or to under-provision, i.e. increase the utilization, but reserve insufficient resources to support the application at its peak load, potentially degrading the end-user experience.

By splitting the control plane from the data plane, SDN enabled more flexible, fine-grained QoS provisioning [142] and it facilitated the concept of network slicing, where,

on top of a common physical infrastructure, different virtual networks, tailored to different traffic needs, can be created, enabling the coexistence of diverse services [143–145]. Several slicing frameworks, such as FlowVisor [146] and FlowN [147], have been proposed over the years. However, they focus on creating isolation between the slice tenants and do not address the specific and very strict per packet QoS requirements of a Tactile application.

Additionally, many SDN frameworks enabling resource reservations were proposed over the years [90–95]. However, the time-varying resource requirements of flows were not taken into account. Moreover, even if dynamic rerouting and resource scaling would be added to these SDN frameworks, they would still violate the constraints (e.g., end-to-end latency) of a Tactile application. To add new or to modify existing rules for the network, SDN controllers need to be informed first, resulting in a significant re-configuration latency penalty. Furthermore, variable latency between the controller and the switches can lead to inconsistencies in the switch tables. If all the switches are not updated at exactly the same time, packets can get dropped (due to non-existing rules) or processed by outdated rules potentially violating the service-level agreement between the network provider and the slice tenant [148]. To solve the aforementioned problem, programmable switches, along with domain-specific programming languages, such as P4, can be used [140]. They offer the possibility to respond quickly to traffic changes directly from the data plane, while the data-packets are being processed [27].

Specific to Tactile flows, [149–151] list the benefits of network slicing to guarantee lower latency, higher reliability and security. However, they consider the network slices to be static, i.e., the lifetime of these slices extends over the full duration of a Tactile flow. While the authors of [152, 153] discuss dynamic-aware routing of Tactile flows, exploiting the burstiness in the packet arrival rates, it is limited to latency and bandwidth optimization in radio access networks alone. It also does not take into account the varying dynamics of the Tactile operator. Several works examined how to use time series methods to cluster/classify human hand motion in a general context (see [154–156]). However, a framework to adopt these methods in the context of Tactile and dynamic network slicing, i.e., how to use these algorithms to design slice specifications for given Tactile quality requirements, is still missing.

In the context of wireless embedded systems, several works have attempted to guarantee communication quality to ensure the stable control of remote devices. Some of these works also address how to dynamically change the communication path between a controller and a remote device without compromising on stability (see [157] and references therein), a concept that bears some similarity to the concept of dynamic network slicing discussed in this paper. These works, however, focus only on embedded wireless systems working on a limited number of hops. In particular, they are not tried and tested on IP network components such as switches, nor do they account for the peculiarity of historical data to design slices.

3.5. CONCLUSION

In this chapter, we used the fact that the dynamics of a Tactile Internet application vary over time and are usually under their peak value to dynamically, at runtime, scale the network resources assigned to this application. To do so, we designed a framework

that combines the advantages of a typical SDN architecture, such as the centralized control and the possibilities of advanced traffic engineering, with the flexibility offered by the programmable network switches. Next, we showed that, by offloading latency-sensitive tasks to the programmable switches, our solution can provide hard QoS guarantees needed for tactile applications while maintaining high slice utilization and minimizing the used network resources.

While our solution was developed with Tactile Internet applications in mind, it is generic and can be used for any applications whose requirements vary in time as long as there is an edge controller that a priori informs the network controller of all the possible slices that can be used as well as which slice to use at what time.

4

ELASTIC NETWORK SLICING

In the previous chapter, we introduced the network slicing concept by designing a data plane protocol that adapts to the current application's requirements. It does so by allocating or de-allocating network resources assigned to the flow and (optionally) by rerouting traffic. However, this solution can only scale the network resources vertically (on one network path) and cannot deal with traffic demands that exceed the link limit and network state. Furthermore, with the rise of programmable data planes, the state of the network slice and functions can change at rates reaching Tbps, making the traditional controller-driven state transfer solutions not feasible. Consequently, a data plane component, able to react to short-term events, is required.

In this chapter, we extend the previously designed framework to support horizontal slicing to solve the issues mentioned above. This form of scaling redistributes the traffic over multiple paths (with its network functions and state). To do so, we extend the link configuration module described in Chapter 3 with two additional data plane components: (1) load monitoring, able to detect the appropriate scaling moments in the data plane; and (2) state management, able to maintain a consistent network state.

This chapter is based on a published conference paper: B. Turkovic, S. Nijhus, F.A. Kuipers, *Elastic Network Slicing*, NetSoft 2021-IEEE International Conference on Network Softwarization (2021) [158]

4.1. INTRODUCTION

Since their introduction, Software-Defined Networking (SDN) and Network Functions Virtualization (NFV) have enhanced network flexibility, reconfigurability, and agility [142, 159]. Moreover, when combined, they support offering Quality-of-Service (QoS) through the concept of network slicing [144, 160].

Network slicing assumes that virtual networks, each tailored to different service needs, are created on top of a shared physical infrastructure. Each of these virtual networks consists of virtual nodes and virtual links. As can be seen in Figure 4.1, every virtual link represents a path with reserved network resources (e.g., bandwidth) in the physical network. Moreover, those links connect virtual nodes, representing Network Functions (NFs), that provide a specific network functionality. Stateful NFs, such as firewalls or heavy-hitter detectors, require knowledge of the previously processed packets to function correctly, while stateless NFs, such as routing, do not [145].

As traffic volumes are unpredictable and generally change over time, a static slicing solution either over-provisions resources or does not guarantee a certain QoS [130]. A slicing solution that supports elasticity, i.e., the ability to automatically scale the assigned network resources to match the current traffic volumes, would solve this problem. We discern two ways of scaling: (1) vertical scaling, in which slice resources are scaled up/down at the NF(s) and reserved bandwidth increased/decreased on the virtual link(s) (Figure 4.2, targeted in Chapter 3), and (2) horizontal scaling, in which a new NF is deployed/removed and a portion of the traffic redirected by creating/removing virtual links connected to the slice, to reduce the load on the existing NF (Figure 4.3). This way, service providers only pay for the resources they use (pay-per-use model), end-users get their requested level of QoS level, and network providers can support multiple services simultaneously using fewer resources.

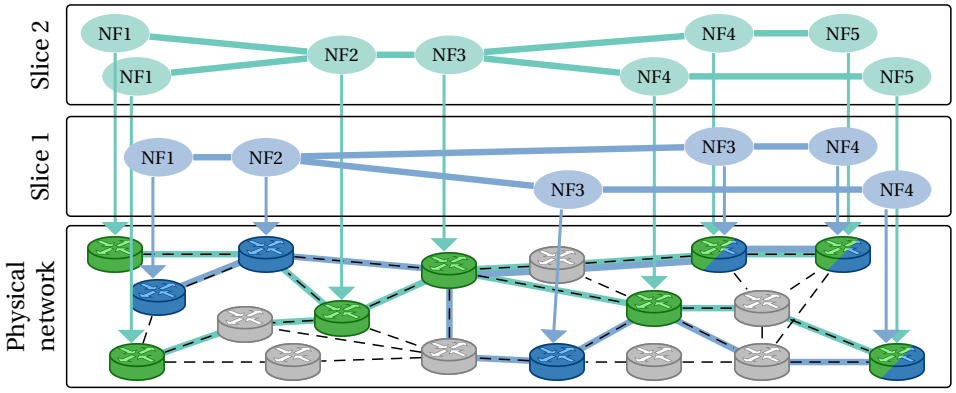


Figure 4.1: Network slicing. Several virtual nodes/links may run atop of a physical node/link.

4.1.1. SCOPE & MOTIVATION.

In this chapter, we consider elastic slicing (both vertically and horizontally) in the context of P4-programmable data planes [140]. On the one hand, NFs primarily responsible

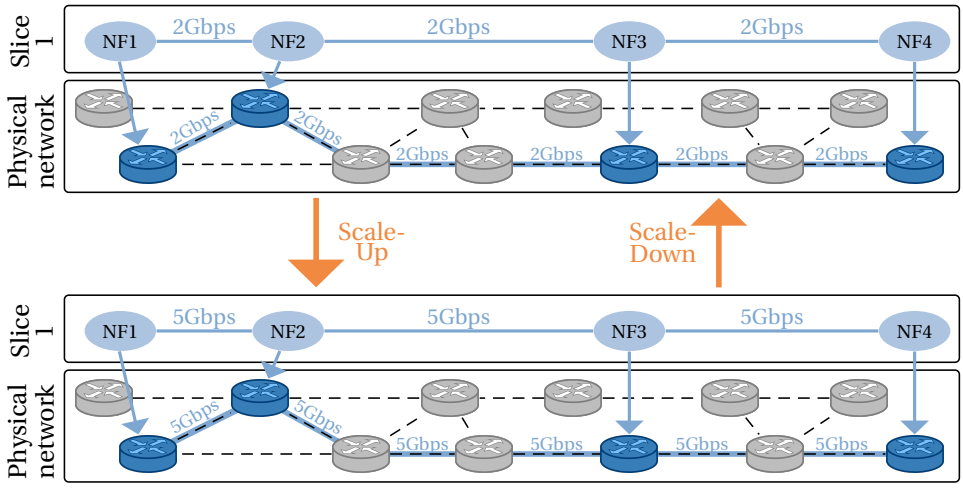


Figure 4.2: Vertical scaling. Whenever a new flow joins or the volume of an existing one increases, resource reservations on all the virtual links are increased (Scaling-Up). Similarly, if the traffic demands decrease resource reservations on all the links are decreased (Scaling-Down).

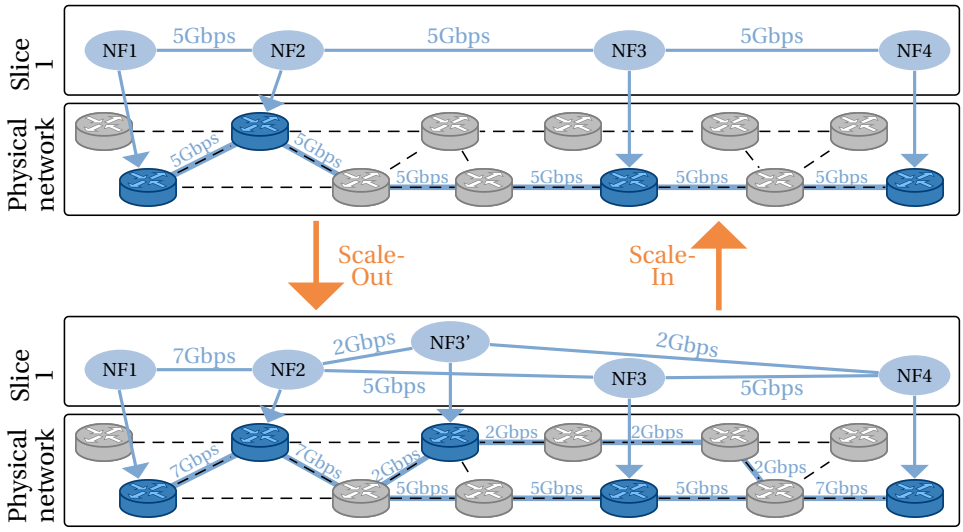


Figure 4.3: Horizontal scaling. When NF3 is no longer able to process all the incoming traffic without any QoS degradation, and/or bandwidth assigned on virtual links NF2-NF3 and NF3-NF2 cannot be scaled up, a new NF3 instance NF3' is spawned and two new virtual links (NF3'-NF2 and NF3'-NF4) created (Scale-Out). Similarly, if enough resources to process all the incoming traffic are present on one of the paths, two virtual nodes are merged into one (Scale-In).

for packet forwarding (e.g., firewalls, NAT, monitoring) might benefit from hardware acceleration by being offloaded to (P4) programmable switches. On the other hand, programmable hardware can process packets at Tbps speeds [98]. Hence, NF-state may

change very frequently, making controller-driven scaling and traditional NFV frameworks (focused on migrating software-maintained states) too time-consuming. Moreover, even if controllers could keep up, migrating an NF, which potentially maintains hundreds of state variables per flow [161], would overload the controller, thereby prolonging the scaling time and leading to state inconsistencies. Yet, up-to-date state information is crucial for the correct functioning of many NFs.

Fortunately, programmable switches come with monitoring features that enable the data plane to report the exact QoS the packets experienced while being processed [27, 99]. And, in contrast to centralized approaches, programmable switches allow us to offload time-sensitive actions from the central controller to the data plane. When combined with the advanced monitoring features, we can quickly detect and react to changing traffic conditions [130].

4.1.2. CONTRIBUTIONS & OUTLINE.

We present an elastic network-slicing framework for P4-programmable network devices that economizes on slice resource utilization, while maintaining state consistency and at low scaling time.

We split our framework into two components: (1) a central controller and (2) a data plane component. With its global view of the network, the central controller is responsible for long-term network management, such as the (de)allocation of NFs and route calculation. The data plane component is deployed directly on the switches and only has a local network view but fast reaction time and accurate traffic information. Therefore, it is responsible for reacting to time-sensitive operations, such as load monitoring (Section 4.3.1), state transfer, and virtual link configuration (Section 4.3.2).

In Section 4.4, we evaluate our framework, both through emulation as well as via experiments on programmable hardware, by comparing it to traditional (controller-driven) approaches. Our experiments show that only by having an "intelligent" data plane, on-time scaling can be achieved.

4.2. ELASTICITY FRAMEWORK

To support elasticity, we propose a hierarchical framework (Figure 4.4) consisting of:

1. A central controller (CC) that, with its global overview of the available network resources and existing traffic flows, determines the network's long-term behavior. It is responsible for initializing new slices, finding the most appropriate locations to place NFs during scaling, and guiding the data plane component.

2. The data plane component (DPC) that, based on the central controller's input and measured traffic conditions, performs all latency-sensitive tasks. It is responsible for load monitoring, state transfer, and flow rerouting.

To support the information exchange between different data plane components running at different switches, we implemented a custom *slice management (SM) protocol*, shown in Figure 4.4. During each scaling process (horizontal or vertical), the DPCs exchange information and update the slice (e.g., reroute flows, transfer state, adjust bandwidth) using the SM header.

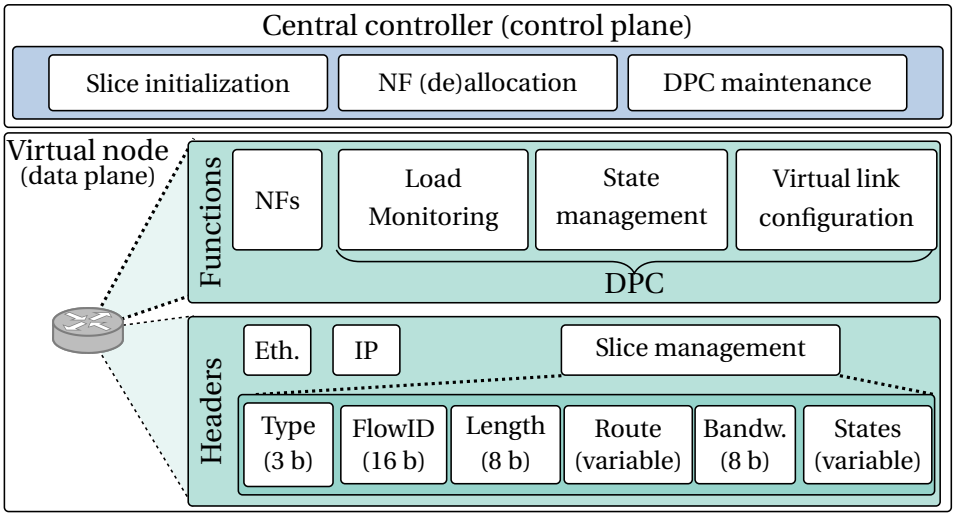


Figure 4.4: Hierarchical design of the slicing framework.

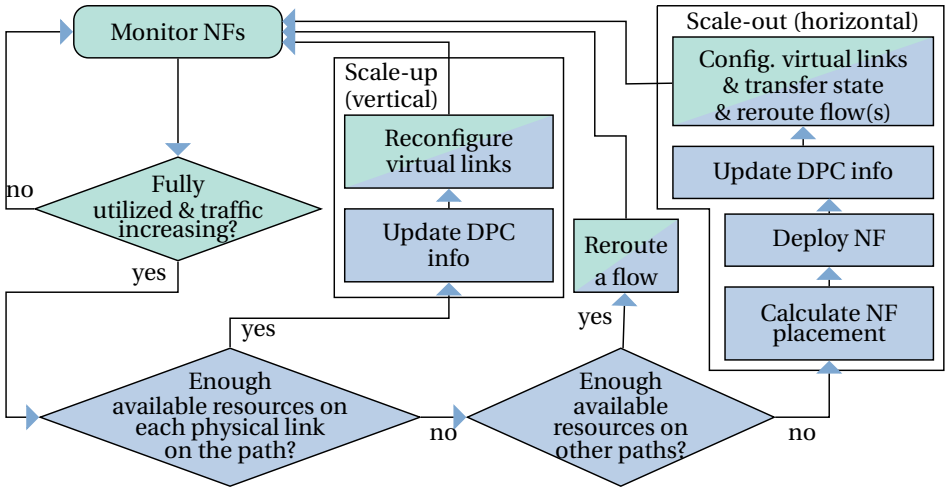


Figure 4.5: Flow chart illustrating the process of scaling-out and -up.

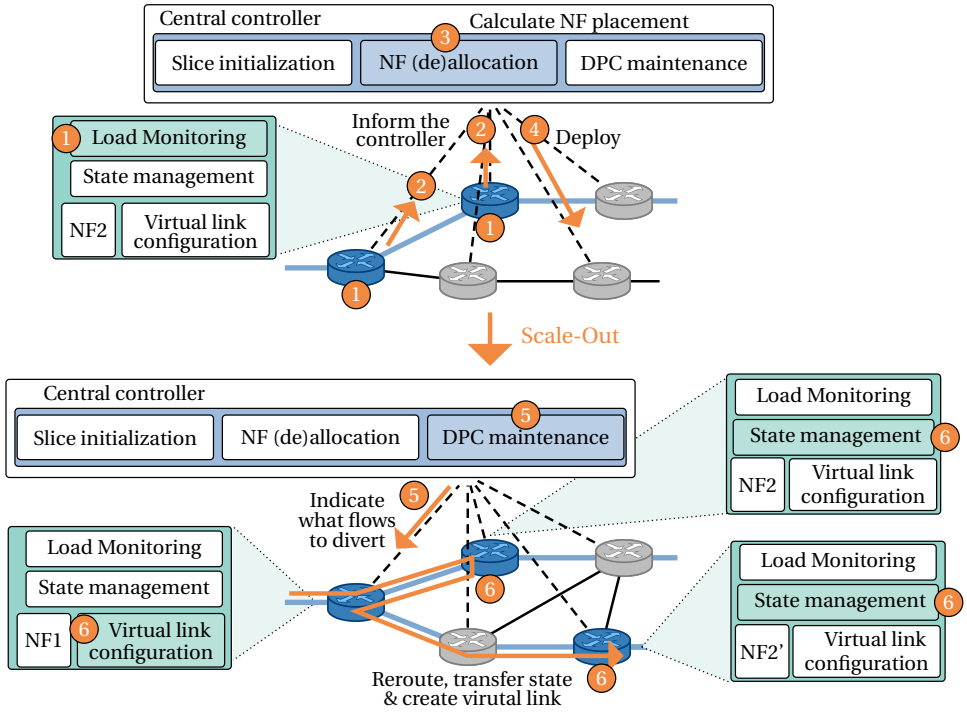


Figure 4.6: Interactions between components during scaling-out. DPCs continuously track the slice's utilization (step 1, Section 4.3.1). If they detect overload, they notify the CC (step 2). The CC subsequently determines the switch where NF2' will be deployed on and the route to connect the node to the rest of the slice (step 3). Next, the CC deploys the new NF (step 4) and, upon its completion, informs NF1 to divert a portion of its traffic to the newly deployed NF2' (step 5). When a packet of this flow is received, the state from NF2 is transferred to NF2', and a virtual link is created using the Slice Management Protocol (step 6, Section 4.3.2).

Figure 4.5 illustrates the processes to react to overload. DPC processes are shown in green and involve all latency-sensitive tasks. For example, a DPC continuously monitors the slice and quickly detects traffic changes. In contrast to a controller-driven approach, it makes this decision on a per-packet basis, informing the CC only when scaling is needed, and does not depend on a monitoring interval or the control plane's speed. If detected, the CC (whose processes are shown in blue) first attempts the easier of the two: vertical scaling. If insufficient resources are available at that path, the CC tries to reroute a flow to a path that would have enough resources or, ultimately, scales-out. Since scaling-out is the most involved of these three situations, Figure 4.6 explains the involved interactions in more detail. It is essential to notice that, while the CC indicates which flows to reroute (green-blue blocks in Figure 4.5), the process of rerouting and state-transfer is completely offloaded to the data plane (Figure 4.6). Therefore, route/state inconsistencies, due to one switch receiving a controller update (e.g., a new route) before others or after the state has changed, are avoided.

During under-load detection, the processes are similar (but reversed). As before, upon detection, DPCs inform the CC, which either scales vertically (scaling-down) or

horizontally (scaling-in). However, in contrast to the previous example, during under-load, if enough resources are available on one path so that another one can be merged into it, the framework will initiate the scaling-in (horizontally) independently of the possibility to scale-down (vertically, see Section 4.3.1).

4.3. DATA PLANE COMPONENT

How much bandwidth to assign? We decided to use an auto-tuned value ΔBw, which represents the step in which we increase/decrease reserved bandwidth. If ΔBw is set low, the reserved bandwidth is increased/decreased in tiny steps. While this increases resource efficiency, it also leads to instability and frequent scaling processes for dynamic traffic. If ΔBw is high, resource efficiency reduces, and scaling will occur infrequently. Our solution initially assigns a low value to ΔBw to preserve the slice resources. After each scaling event, ΔBw is increased by a factor k, and a timer is started. If another scaling event occurs within this timer, the same process is repeated. This process continues until an interval is encountered in which no scaling occurred, which causes ΔBw to be reset to its initial value. This way, we start scaling conservatively, only assigning small chunks of bandwidth to the slice. However, if we detect a high increase, we quickly build-up ΔBw to reduce the number of scaling events.

Since our framework relies on offloading latency-sensitive tasks to the DPC, we will explain two main tasks: (1) load monitoring and (2) virtual link configuration, state transfer, and flow rerouting (as well as all involved modules) in more detail.

4.3.1. LOAD MONITORING

Since the scaling procedure is relatively time-consuming and requires many network updates, frequent scaling would lead to network instability and sub-optimal resource utilization. To infer the best times for horizontal scaling, for each P4 NF, we deployed two two-rate three-color meters: *growth meter* to detect that the slice is close to its full capacity, and *decline meter*, to detect that the slice is underutilized.

Growth meter. Scaling-out (and up) is initialized whenever the slice is no longer able to process all the traffic without any degradation. Consequently, the growth meter rate threshold needs to be configured low enough to allow the CC to deploy the new NF without any degradation to any of the flows currently processed in the slice. In this chapter, we set them to ΔBW and $2\Delta BW$ less than the reserved bandwidth BW_r. Additionally, to avoid unnecessary NF allocations, too frequent scaling, and instabilities due to traffic fluctuations, the switches ensure that the traffic is increasing continuously.

To do so, we instructed the switches to track three additional metrics: *the growth rate* (R_g), *growth counter* (C_g), and *decline counter* (C_d). The growth rate tracks the number of yellow packets processed in the last N_g packet interval (Figure 4.7). C_g uses R_g to track the overall trend of the slice utilization and is calculated as follows: if the number of yellow packets (the growth rate R_g) is increasing between two subsequent intervals of N packets, or the number of red packets is greater than zero, the C_g is increased by one. However, if R_g is decreasing, C_g is reset to 0. This way, if the slice utilization is continuously increasing (for at least M intervals), scaling will occur after at most $N \cdot M$ packets since the first yellow/red packet (Figure 4.7). Moreover, to detect under-load,

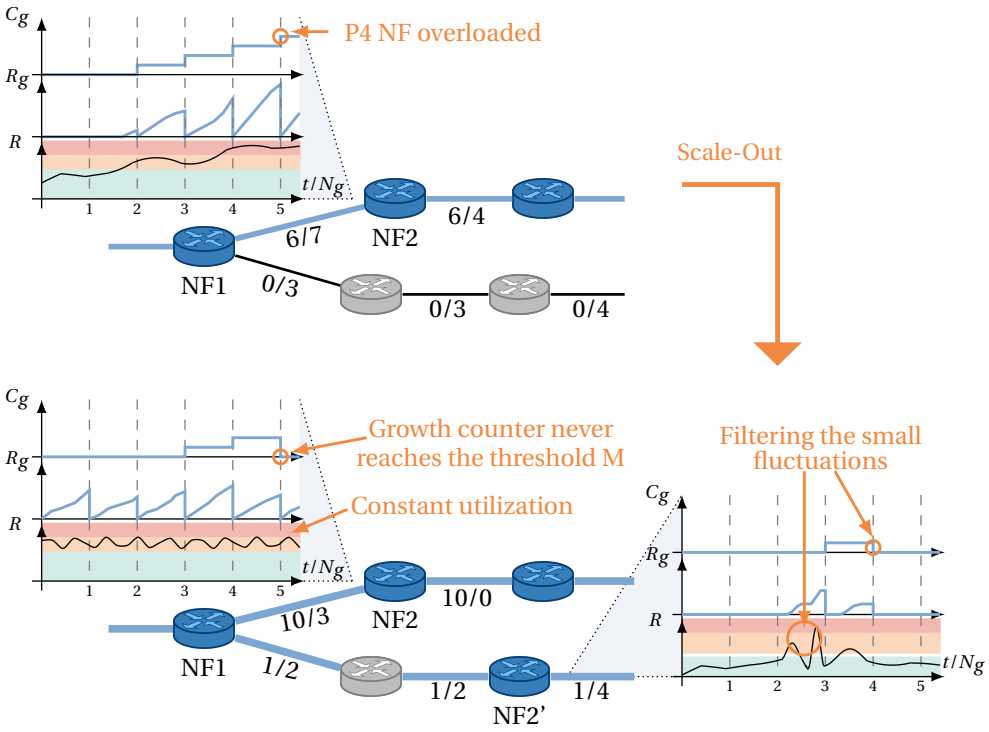


Figure 4.7: Growth meter. If the traffic volumes increase and more and more packets are colored yellow or red, the growth rate R_g and, consequently, the growth counter C_g increase until C_g reaches the threshold M. At this moment, scaling is initialized. In contrast, small fluctuations in the traffic volumes or large temporary peaks are filtered out. The first number above the link represents the amount of reserved bandwidth, while the second number represents the total amount of bandwidth available on the link (i.e., not assigned to any other slices).

each switch tracks C_d, increasing it by one if all the packets in an N-interval are green. Otherwise, it resets it to 0. Consequently, if C_d reaches M, meaning that for the last $N \cdot M$ packets, the slice had excess $2\Delta BW$ bandwidth, the switch will initiate the scaling-down process (by ΔBW).

Merging meter. To be able to scale-in, an NF should process less traffic than the maximum available on the other NFs implementing the same functionality. As illustrated in Figure 4.8, the CC configures the rate thresholds by calculating the maximum traffic volume that any of the other NFs can take over. To avoid too frequent scaling and instabilities, the switches track an additional metric, *the merging counter* C_m. This metric tracks the number of all-green intervals (in the same way C_d did), and, whenever it reaches M, scaling-in is initialized. Finally, the CC readjusts the thresholds (Figure 4.8).

Processing at the CC. To filter the requests belonging to the same event (e.g., scaling-up detected at multiple switches), we implemented a back-off mechanism that, every time a scaling request is received, checks if other requests were received in at least the last $N \cdot M \cdot MSS/BW_{reserved}$ seconds.

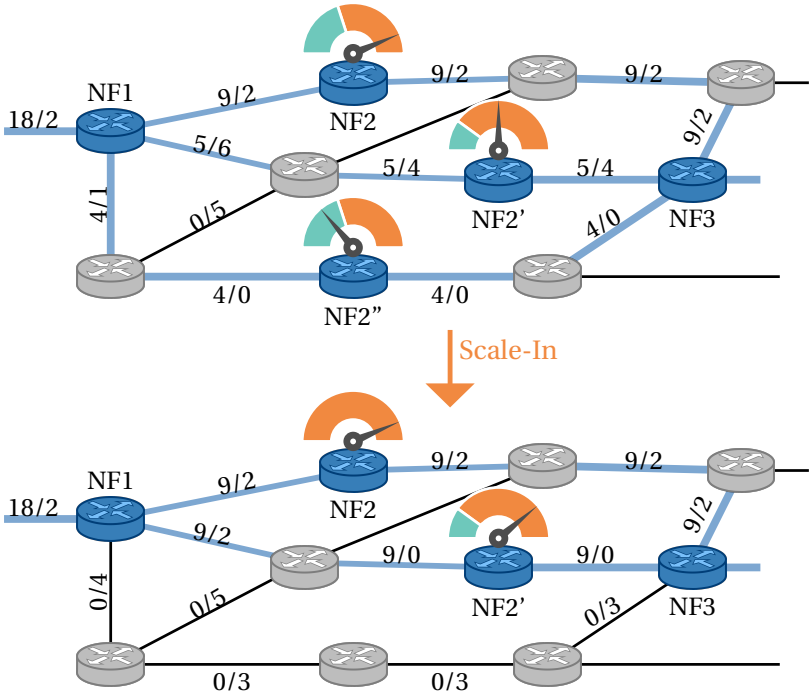
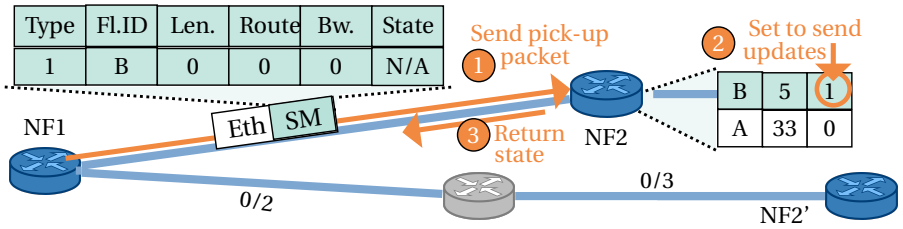


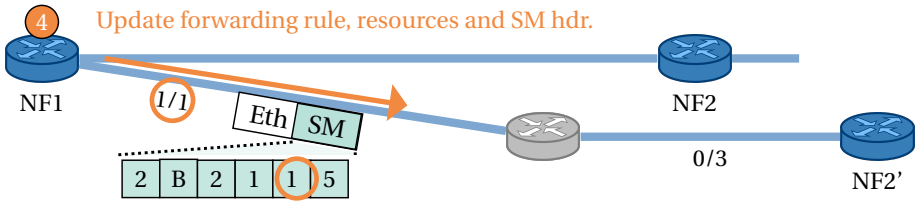
Figure 4.8: To merge NF2" into NF2' or NF2, either of them needs to have enough resources to take over the traffic processed on NF2" (i.e., 4). DPC maintenance calculates the minimum resources available on the virtual links connecting NF2 and NF2' to NF1 and NF3. Finally, it sets the rate thresholds on NF2" to the maximum of these two: 4. Likewise, the thresholds on NF2 and NF2' are configured to 4 and 2, respectively. As all the packets processed on NF2" are green, i.e., NF2" is processing less traffic than the amount of free resources available on NF2', scaling-in is initiated and the thresholds recalculated to 0 and 2 for NF2 and NF2', respectively. The first number above the link represents the amount of reserved bandwidth, while the second number represents the total amount of bandwidth available on the link (i.e., not assigned to any other slices).

4.3.2. VIRTUAL LINK CONFIGURATION, STATE TRANSFER AND FLOW REROUTING

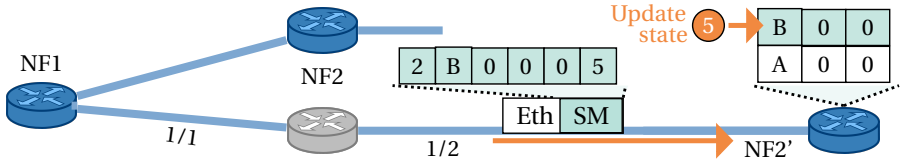
To avoid the state and route inconsistencies associated with a controller-driven approach, our framework reroutes the flows, transfers their state, and updates the resource allocations in the data plane (while processing data-packets), as illustrated in Figure 4.9. To fill in the Slice Management header, it relies on the central controller's updates (in particular, the DPC maintenance module) indicating, among other things, what flow to divert to the new NF. For example, if the controller decided to reroute B (Figure 4.9), it informs NF1 of this. NF1, after receiving the first packet of flow B forwards it to the original NF (NF2) to pick up its state (Figure 4.9a). While the original packet is processed further along the path (thus avoiding unnecessary delay), a small copy is sent back to NF1, and, afterwards, NF2'. Each switch on the new virtual link updates its forwarding rules (e.g., output port stored in a register array) and resource allocation, as this packet passes through (Figure 4.9b). Since it is not possible to update the bandwidth allocations in P4, we decided to generate a digest to a local digest listener, while processing this header.



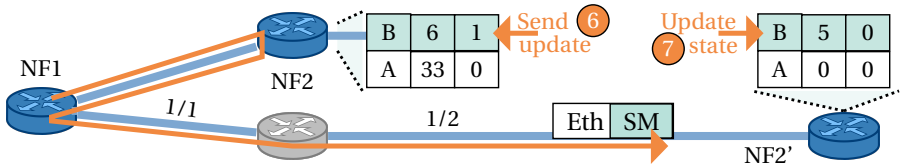
(a) After receiving flow B's packet, NF1 appends the SM header (shown in green) and forwards it to NF2 (step 1). Upon reception, NF2 creates a copy, updates the SM header (e.g., state information, source route to NF1), sets a bit register associated with B to 1, indicating that B's state is migrated (step 2), and returns the packet to NF1 (step 3).



(b) NF1 updates its forwarding rules for flow B, adjusts the allocated bandwidth for the slice to ΔBW (1, circled values, step 4). Next, it updates the SM header (e.g., source route to reach NF2') and forwards it to NF2'. All switches in the path repeat this process, updating their forwarding rule to the value from the source route and allocating the resources.



(c) Upon reception, NF2' updates its state.



(d) If a packet of flow B is received at NF2, an update is sent to NF2'.

Figure 4.9: Virtual link configuration & state management module.

Consequently, this local listener issues the command to update the reservations. When this packet reaches NF1, all subsequent packets (of flow B) are diverted to NF2, and the new virtual link is established. Finally, the same process is used to create the other virtual link (connecting NF2' to the rest of the slice). Moreover, to account for packets that were already forwarded towards NF2 (before NF1's rule was updated), we instruct NF2 to send small updates to NF2', in the same way the first packet was sent, but with a different type (to prevent unnecessary forwarding rules and bandwidth allocation updates (Figure 4.9d)).

Scaling-in. To avoid the dependency on the incoming packets, we decided to change the migration procedure during scaling-in by transporting one flow's state per received packet destined for NF2'. Thus, in contrast to the example shown in Figure 4.9, we use a sequential index to read all the active states of the states table and append it to the SM header. Two situations can occur. First, the data packet can belong to a flow whose state is still present at NF2. In this case, a small copy to transfer the state of the flow indicated by the sequential index is sent to NF2, while the original packet is forwarded further along the path (after updating its state). Second, the packet can belong to a flow whose state was already transferred to NF2. In this case, in addition to the state information of the flow indicated by the sequential index, we also forward the data-packet back to NF2. Upon reception, NF2 updates the state from the SM header as well as the state belonging to the data-packet. Further, while processing these transfer packets, switch NF1 adjusts its forwarding rule for the flow in the SM packet to point to NF2 (instead of NF2'). However, this rule is only put into affect after the state table was transferred (i.e., $table\_size$ packets were sent to NF2').

Difference with SwingState. In our state migration, we leverage the idea from SwingState [162] to transport the packets in the data plane. However, in contrast to SwingState, we avoid the dependency on the incoming traffic patterns, which could lead to a very long scaling-in process and, consequently, overhead for infrequent flows. Moreover, we combine the state transfer with rerouting and significantly reduce the number of updates that need to be sent. To do so, we assume that during horizontal scaling, NFs implementing the same function execute the same P4 program (i.e., state tables have the same size), and that the same hash function is used for index calculation (assumption not needed for SwingState). In scenarios in which this assumption does not hold, the controller can provide a table index mapping to the switch transferring the state, ensuring consistency.

4.3.3. OVERHEAD AND LIMITATIONS

Collisions. Indices to P4 register arrays storing the state are calculated by hashing the packet's header fields. Hence, the probability of hash collisions increases with the number of concurrent unique flows in a slice. When flows collide, P4 NFs merge their state. Hence, our framework, which relies on each NF's state management, does not distinguish between colliding flows either and will treat them as the same flow.

Memory overhead. Per state array, DPC uses two bit arrays to store active flows and transferred flows. Moreover, is uses two registers containing source routes (towards the next and previous NF), and eleven counters for load monitoring (e.g., number of red/yellow/green packets in the current and previous intervals, C_d, C_g, C_m)

Latency and packet overhead. Every time the DPC appends the SM header, it increases the packet's transmission delay. Moreover, our solution generates additional packets (e.g., state transfers, state updates). However, due to the small size of the SM header (a few bytes), this overhead is not significant and lower than other data plane approaches (see Section 4.4).

Packet reordering. Like other data plane scaling approaches (e.g., SwingState) packet reordering can occur if packets are present at the outdated link during rerouting. While we make sure to reroute these packets, we cannot guarantee that all packets will arrive at their correct virtual node in correct order. Consequently, NFs that depend on exact packet order might have their state overwritten by an update packet. To maintain state consistency, a sequence number can be appended to each packet at the first NF, and the state only updated if the sequence number is higher than the last received one.

Hybrid scenario (only some P4-programmable switches). To use our approach, switches acting as NFs must be P4-programmable. For non-virtual nodes, traditional SDN switches can be used. The only difference would be that to adjust bandwidth reservations a packet would be sent to the CC resulting in a potential latency increase. Moreover, to avoid potential inconsistencies during rerouting, the controller would, while deploying a new NF, update the rules on SDN switches connecting this new NF to the rest of the slice.

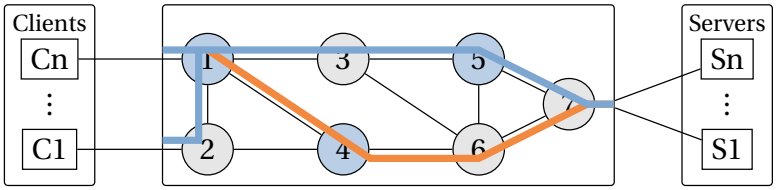
Recirculations. During the scaling-in process, in some cases we need to read multiple indexes from the same state register array (i.e., a state belonging to the original flow and the state belonging to the flow indicated by the sequential index). For switches with a limited number of memory accesses per register array, we implement these actions using recirculations.

P4 NF deployment. All currently available programmable hardware requires a firmware reload when a new P4 program is deployed. Since this is never instantaneous, it can lead to some downtime, state loss, and service interruptions for all NFs deployed on the switch and all flows processed by it. Fortunately, various data plane reconfiguration approaches facilitate uninterrupted reconfigurability of the data plane [163–165], and should be used to enable dynamic NF placement. In this chapter, for simplicity, the P4-program contains all the NFs, and we just update a register indicating if the NF is active or not.

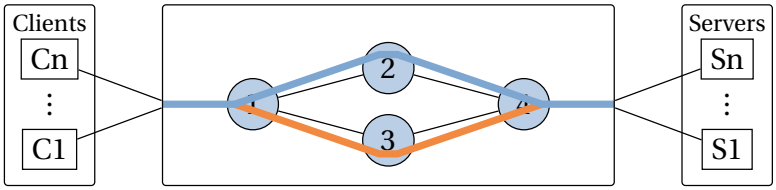
4.4. EVALUATION

Experiment setup. To evaluate our solution, we used two topologies, shown in Figure 4.10. The first one (Figure 4.10a), was emulated using Mininet with the P4 software-switch (behavioural model [141]), while the second one (Figure 4.10b) used an Intel Tofino switch [98]. We observed similar results with both our implementations. A notable difference was a more unpredictable latency in Mininet, presumably due to emulation. We will therefore focus mostly on the hardware measurements.

Traffic scenarios. We considered two traffic patterns: (1) baseline scenario (Figure 4.11a), used to test the scaling processes; and (2) state transfer stress scenario (Figure 4.11a), to test the state transfer process while scaling-in, especially when some flows send packets rarely. In all scenarios, the number of TCP/UDP flows (generated using iperf3) was



(a) Simulation topology (7 nodes, 10 links).



(b) Experiment topology (Intel switch, 4 nodes, 4 links).

Figure 4.10: Topologies. Blue lines represent the initial slice, while red lines represent the path added during horizontal scaling.

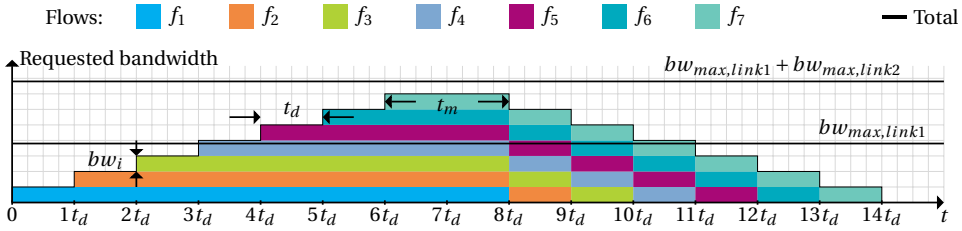
varied, as well as t_d, t_m, n, k. Moreover, delay t_c was added to each control plane request/replay to account for the slower control plane. Each experiment was run five times. Since we mainly focused on the data plane's performance, the controller did not track the flows' bandwidths, but always rerouted the last flow that was added to the path.

Comparison baselines. Our monitoring approach has been compared to an approach that uses an SDN controller to re-configure a slice (both route and bandwidth reservations) by periodically monitoring the queuing delay/utilization of the slice (controller-driven approach). We varied the controller's monitoring delay between 1, 3, and 5 seconds, and the number of successive intervals in which an increase/decrease in queuing delay/utilization happens from 1 to 3. For the state transfer, we compared our approach to (1) SwingState [162] and (2) a controller-driven polling approach (i.e., the controller, while rerouting the flow, also polls for state information). Since we assume that both the NFs are identical (including the hashing algorithms), for SwingState, we only sent the state information in the update packets (in contrast to the full copy of the original packet). This way our solution would not have an unfair advantage.

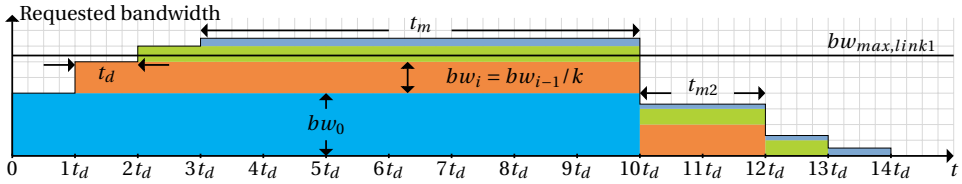
Performance metrics. We evaluated all our schemes on (1) the average and maximum round-trip times, (2) the average and maximum jitter, (3) overhead caused by the state-transfer process (in bytes and packets), (4) percentage of encountered corrupt states during transfers (both temporary and at the end of the scaling process), and (5) duration of the scaling process.

4.4.1. OVERALL PERFORMANCE

Control plane. In all our experiments, we observed that the control plane was limiting the factor in our framework. For example, the switch generates the digest notification (i.e., small notifications to the control plane indicating the need for, for example, scaling and/or rerouting) much faster than control software can process them and adjust the



(a) Baseline scenario.



(b) State transfer stress scenario.

Figure 4.11: Traffic scenarios.

parameters (e.g., meter rates, deciding which flow to reroute).

Tuning the monitoring module ($N, M, \Delta Bw$). Choosing a lower N or M decreases the detection time since fewer packets need to be processed by the NF to detect overload (Figure 4.12a). However, we observed that a very low N (e.g., 64) increases the number of intervals needed for detection (e.g., for $M = 5$, our framework needed 15 intervals on average for $N = 64$ compared to 5 intervals on average for $N = 256$). Moreover, when combined with a very high M, the probability of resetting the counter C_g, and missing the scaling event, increases. Similarly, during scaling-in/scaling-down, low values of N and M decrease the detection time (Figure 4.12a). However, at the same time, we observed that the probability of scaling-in/scaling-down too quickly (or immediately after the scaling-up/scaling-out process) increases, leading to instability.

Further, we evaluated the influence of different ΔBW. A higher ΔBW decreases the number of generated scaling events, but due to the reduced granularity increases the excess bandwidth (i.e., $Bw_{res} - Bw_{req}$, Figure 4.12c). Additionally, since the threshold needed to trigger scaling-down is set at $Bw_{res} - 2\Delta Bw$, the excess bandwidth per-link is usually under $2\Delta Bw$ (Figure 4.12c). The only difference to this rule occurs when, due to the adaptive nature in which we assign the bandwidth (e.g., if two subsequent scaling-up events are registered, we double the ΔBw), the bandwidth is scaled higher than $Bw_{req} + 2\Delta Bw$ (Figure 4.12c for $bw = 19$ and $\Delta Bw = 10$). However, if no new flow is generated, this increase is only temporary and is always followed by a scaling-down event. Furthermore, if the increased number of scaling events is combined with a very high M and/or N, by very steep bandwidth increases, the time needed to reach the needed bandwidth can be very long (and might never be reached during the duration of the scenario). Hence, we chose $N = 128$ and $M = 5$ and $\Delta Bw = 10$ as the values that provided a good trade-off between fast and stable load detection for the remainder of

N, M	Scaling-out ($bw = 15Mbps$) 5	10	25	Scaling-in ($bw_1 = 19Mbps$) 5	10	25
64	0.98	1.23	2.60	0.90	1.27	3.24
128	1.27	1.91	5.70	1.28	2.90	4.73
256	1.47	2.75	6.59	2.95	3.98	8.07
512	2.42	4.98	12.66	4.34	6.73	14.41
1024	4.49	9.61	24.97	6.86	12.32	27.68

(a) Detection speed (in $1000 \cdot pkts$) of f_2 during scaling for different values of N and M. $\Delta Bw = 10Mbps$, $bw_1 = bw_2 = 14Mbps, t_d = 10$.

bw_2, t_m	1	2	3
10	4	5	5
7	3	4	5
5	3	4	5
3	2	3	5
1	0	0	0

(b) Number of times (out of 5) scaling was detected for f_2.$M = 5, N = 256, bw_1 = 25, \Delta Bw = 2.5, t_d = 15, n_f = 2$.

bw, ΔBw	Detection speed scaling-up 10	5	2.5	Detection speed scaling-down 10	5	2.5	Max. excess bandwidth 10	5	2.5	Num. of scaling operations 10	5	2.5
11	0.99	0.99	1.24	1.43	1.43	1.43	18	8	3	4.0	8.3	10.0
13	1.31	1.39	1.31	1.61	1.44	1.19	14	9	1.5	4.0	9.0	12.0
15	1.38	1.47	1.30	1.29	1.38	1.12	20	10	5	6.0	10.0	12.0
17	1.26	1.01	1.01	1.34	1.17	1.17	16	6	3.5	6.0	8.0	14.0
19	1.44	1.34	1.35	1.46	1.37	1.12	22	7	2	7.0	10.0	15.0

(c) Baseline scenario ($N = 256, M = 5, t_d = t_m = 10s, n_f = 2$). Detection speed of the second flow (in $1000 \cdot pkts$), excess bandwidth ($Bw_{res} - Bw_{req}$) and the total number of scaling operations for different values of bw and ΔBw.

bw, t_c	Our approach (dataplane reroute) 0	0.1	1	SwingState + control plane reroute 0	0.1	1
1	0.003	0.003	0.003	0.09	0.09	0.18
4	0.003	0.003	0.003	0.09	0.09	0.20
16	0.003	0.003	0.003	0.18	0.27	1.6
64	0.003	0.003	0.003	0.54	1.63	6.1
262	0.003	0.003	0.003	1.03	3.25	24.2

(d) Traffic volume overhead per flow during scaling-out (in $1000 \cdot pkts$) for different controller delays t_c and different bw. Values for t_d are in s. $n_f = 1$

bw, w	Our approach (dataplane reroute) 64	128	256	SwingState + control plane reroute 64	128	256
1	0.02	0.05	0.11	0.18	0.18	0.18
5	0.02	0.05	0.11	0.18	0.19	0.18
10	0.02	0.05	0.11	0.26	0.25	0.25
20	0.02	0.05	0.11	0.36	0.36	0.36
50	0.02	0.05	0.11	0.89	0.85	0.91

(e) Traffic volume overhead during scaling-in (in $1000 \cdot pkts$) for different state table sizes w (size of the register array) and different bw. $t_c = 0.01, n_f = 2$.

Figure 4.12: Evaluation of the separate modules of the framework (4-node topology, TCP). All bandwidth values are in $Mbps$.

this chapter.

Scaling-down upon competition of all flows on a path. P4 programs are executed upon packet reception. Hence, if a switch does not receive packets, the load monitoring module will not detect the last scaling-down event. Consequently, at least $2\Delta Bw$ resources will remain assigned. The only exception to this is a scaling-in event that releases all the resources assigned on a path. To avoid these situations, the central controller must detect these cases and, subsequently, release the assigned resources.

State transfer. SwingState transfers the state in the data plane, but relies on an external entity (in this case, the central controller) to reroute the traffic. Hence, until the controller reroutes the traffic, SwingState continues sending updates to the newly deployed NF. Consequently, as Figure 4.12d illustrates, the overhead of transferring one flow during scaling-in depends on the controller delay t_c and the flow's bandwidth (how

many packets are sent before the controller can react). In contrast, our approach incorporates a data plane rerouting procedure. Hence, it depends solely on how fast the network can transport the SM header to the previous NF, which will, upon reception, update its forwarding rule (i.e., stateful register storing the output port). Furthermore, during scaling-in, our approach depends on the width of the state array and the number of flows processed by the NF. For example, if we consider the scenario in Figure 4.12e, for a table size of 64, our approach sends 64 packets towards the NF to pick up the states stored in each register. However, the two flows we were transferring in this example had hashes 41 and 56. Hence, the first 40 packets were processed as usual, and no updates were created (the bit array index indicated that flows with indexes lower than 40 were not active). Next, the state for the first flow (with the index 41) was transferred. Next, for each index between 41 and 56, packets belonging to the first flow triggered an update for the first flow and were sent back to be processsed by the other NF. Finally, all packets between 56-64 triggered an update as well (since they can only belong to the flows that were already transferred). In contrast, SwingState does not have this dependency. However, it depends on the traffic pattern of the incoming packets. Hence, if an infrequent flow would need to be transferred during scaling-in, it would delay the whole process (and the controller would not be able to remove the NF).

4.4.2. DATAPLANE VS. CONTROLLER-DRIVER APPROACH

Control plane vs data plane load monitoring. In our experiments, we observed that the controller-driven approach is unreliable. When we increased the monitoring interval (from 1 second to 3 or 5 seconds), but kept the number of successive intervals in which an increase in queuing/utilization should be observed constant (2 or 3), the TCP's congestion control mechanisms at the end-hosts kicked-in, reducing the rate and, consequently, the observed queueing delay/utilization. Thus, the controller immediately detected a decrease and concluded that the congestion was merely a consequence of a short-term fluctuation and that scaling is not needed. In contrast, if we reduced the number of successive intervals to 1, we observed that the controller detected the need for scaling too early and, consequently, oscillated between a scale-in/down and scale-out/up phase. The data plane solution, due to the possibility of using smaller monitoring intervals (number of packets N) and the possibility of aggregating the statistics on the switch, detected the overload faster, and, consequently, maintained a lower average and maximum delay for all the flows (Figure 4.13).

Very low t_d. As mentioned above, our framework was limited by the latency between the switches and the central controller. Consequently, in scenarios with a lower t_d our framework had less improvement than with higher t_d (Figure 4.13). Moreover, when we set $t_d < t_c$, our framework could not scale in time.

Monitoring overhead and limitations. The CC must periodically query the switches' registers. This overhead depends on the configured monitoring interval t_{mon} and is equal to $n_{vnf} \cdot t_{mon}$. Moreover, during scaling procedures, the CC could not process all the tasks within the given monitoring interval, resulting in delays. In contrast, by offloading monitoring to the data plane, our solution only contacted the controller in case overload was detected. This resulted in a significant reduction of this overhead to a few digest notifications per switch for each scaling-event.

	Max. delay [ms]			Avg. delay [ms]		
Slicing approach	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$
Our approach	171.0	127.4	58.06	69.8	31.6	38.0
SwingState + Polling	185.5	186.8	186.6	105.7	63.6	54.9
Controller-Driven + Polling	226.3	189.7	186.8	107.1	62.1	53.3
No Slicing	186.8	186.8	186.8	135.9	127.2	125.5

(a) Baseline scenario. Observed maximum and average delay at the end-hosts.

	Reserved resources [Gb]			Min throughput [%]		
Slicing approach	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$
Our approach	6.1	12.2	20.6	81.29	99.78	99.89
SwingState + Polling	6.8	14.1	23.4	70.25	90.75	99.85
Controller-Driven + Polling	6.3	13.6	23.6	67.7	90.75	99.83
No Slicing	4.8	10.4	17.4	35.66	59.10	57.99

(b) Baseline scenario. The amount of reserved resources and the minimum throughput per flow among all the end-hosts.

	Corrupt [%]			Max. Duration [s]		
Slicing approach	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$
Our approach	0.00	0.00	0.00	0.001	0.001	0.001
SwingState + Polling	0.00	0.00	0.00	0.1	0.1	0.1
Controller-Driven + Polling	43.0	48.1	55.2	0.1	0.1	0.1

(c) State transfer accuracy and duration.

	Overhead [$\#kB$]			Overhead [$\#pkts$]		
Slicing approach	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$
Our approach	0.28	0.28	0.29	12.2	12	12.4
SwingState + Polling	50.36	46.94	47.93	1398.8	1304.0	1331.5
Controller-Driven + Polling	-	-	-	-	-	-

(d) State transfer overhead.

Figure 4.13: Baseline scenario (4-node topology, TCP, $t_m = 30$, $n_f = 8$, $bw = 20Mbps$, $bw_{link1} = 100Mbps$, $t_c = 0.1$, $w = 64$.)

State corruption. During every reroute (scaling-in or scaling-out), the controller-driven approach could not transfer the state in time, causing all rerouted flows to have an incorrect count. Moreover, while deploying the state, the controller overwrote the present state in the switches, deleting all the state information and, hence, in contrast to data plane approaches (our framework, SwingState), it could never recover. Consequently, all packets that followed had an incorrect count ($\approx$ 50% of the packets, since 4, flows got rerouted to the second path in Figure 4.13). In contrast, our approach and SwingState maintained state consistency, with our scheme being faster (due to the offload of the rerouting procedure to the data plane).

4.4.3. STATE TRANSFER STRESS SCENARIO

To test the scaling-in functionality, we configured the controller to reroute 8 of the 11 flows to the red path. After the first flow was completed, the controller initiated the scaling-in. We observed that t_c (the delay between controller and switch) limited the speed of the controller-driven approach (Figure 4.14). Moreover, as previously, during each rerouting, the state was corrupted. In contrast, SwingState and our framework avoided inconsistencies, recovering from them using update packets. Moreover, low-

	Corrupt [%]			Max. duration [s]		
Slicing approach	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$
Our approach	0.00	0.00	0.00	0.07	0.07	0.07
SwingState	0.00	0.00	0.00	7.3	42.2	86.5
Polling	5.62	5.95	6.14	0.1	0.1	0.1

(a) State transfer accuracy and duration.

	Overhead [$\#kB$]			Overhead [$\#pkts$]		
Slicing approach	$t_d = 1$	$t_d = 5$	$t_d = 10$	$t_d = 1$	$t_d = 5$	$t_d = 10$
Our approach	55.68	55.68	55.68	45	45	45
SwingState	10.48	11.45	12.5	291.2	318.3	347.4
Polling	-	-	-	-	-	-

(b) State transfer overhead.

Figure 4.14: State transfer scenario for the 4-node topology, Overhead during scaling-in. $t_m = 120$, $n_f = 11$, $bw = 1kbps$, $k = 4$, $bw_{link1} = 1.32Gbps$, $t_c = 0$, $w = 64$.

speed flows determined the duration of this process for SwingState (Figure 4.14). In our experiments, we noticed that iperf3 (which we used to generate traffic) sent bursty traffic, especially at low speeds, and would remain idle the rest of the time. Consequently, packets originating from flows with the lowest bandwidth occurred even less frequently than initially expected, and only after the last flow (with the lowest bandwidth) terminated SwingState was able to complete the scaling-in. Furthermore, since our framework was transferring only eight flows but directed 64 packets (table width size) to the NF, many packets needed to be sent back as an update (since their state was already transferred), increasing the overhead in bytes.

4.5. RELATED WORK

Over the years, several network-slicing frameworks have been proposed. However, most of them focus on providing isolation between different slices and do not provide QoS guarantees inside a slice, cannot handle the problems associated with P4 NFs, and/or do not adapt to the time-varying requirements that slices may have. The slicing framework presented in [130] does dynamically scale the resources assigned to a flow. However, it relies on the network edge to detect when scaling needs to occur and only supports vertical scaling.

NFV frameworks. Depending on how the state is organized, stored, and accessed, the different NFV scaling solutions can be divided into local (e.g., [166–170]), remote (e.g., [171]), and distributed approaches (e.g., [172–174]). Remote approaches store all the state remotely at some centralized storage and can thus not be used with P4 NFs, since they would impose significant performance penalties per processed packet (to retrieve the state). Local approaches never migrate the state and can, therefore, not deal with scenarios in which an increase of the flow's volume causes NF overload. Additionally, they must wait for all flows present on an NF to finish before shutting it down, resulting in inefficient resource utilization. Furthermore, most NFV frameworks were not designed with P4 NFs in mind. The ones that were, such as P4NFV [175], use a controller-driven approach and will, consequently, suffer from all aforementioned issues associated with that approach.

Data plane state migration. SwingState [162] depends on the arrival pattern of the incoming packets and can therefore have a long transfer time. LODGE [176] targets distributed network applications by creating a shared network state. SNAP [177] and U-HAUL [178] move only the state of long-lived flows. Thus, these approaches are not well suited for this chapter's objective.

4.6. CONCLUSION

This chapter has presented an elastic network-slicing framework for P4-programmable network devices. The presented framework has a hierarchical design, focusing on both the control and data planes, and builds on top of the framework presented in the previous chapter, which focused on vertical scaling. Hence, similar to Chapter 2, the control plane with its global overview of the network guides the data plane behavior but offloads all time-sensitive tasks to the fast data plane, which, at time-sensitive moments, performs tasks autonomously and with limited input. However, in contrast to the previous chapter, the framework removes the application from the scaling process and introduces the overload (under-load) detection module in the programmable switches. This module is primarily responsible for tracking the utilization of the resources assigned to the slice and, consequently, detects when the slices' resources need to be readjusted. Moreover, the framework in this chapter enables horizontal scaling by introducing the rerouting and state transfer module. By doing so it allows for more flexibility in the traffic management module in the central controller, which no longer needs to make sure that traffic will not exceed the bandwidth of the link and can support flows with changeable and not predetermined maximal bandwidth.

Finally, the last two chapters addressed the second challenge, described in Sec. 1.3 of this thesis, by introducing a dynamic network slicing framework able to handle the changeable requirements of different flows and slices in real-time. Moreover, they demonstrated that offloading time-sensitive tasks to the data plane can increase slice resource efficiency while minimizing scaling time and maintaining state consistency, especially when compared to state-of-the-art controller-driven approaches.

5

INTERACTIONS BETWEEN CONGESTION CONTROL ALGORITHMS

In the previous two chapters, we focused on service isolation by introducing the concept of network slicing. However, it is also crucial to satisfy the quality of service (QoS) requirements of multiple flows present inside these slices. In the following chapters, we, therefore, address interactions between these flows. We start with flows that provide a feedback loop (such as connections which use transport protocols like TCP or QUIC); in particular, we study the interactions between flows using different congestion control algorithms, i.e., an essential component of these transport layer protocols crucial in achieving high utilization while preventing network overload.

Over the years, many different congestion control algorithms have been developed, each trying to improve over the others in specific scenarios or for specific applications. However, the interactions between flows using different algorithms and their co-existence have, to date, not been thoroughly evaluated. This chapter fills that knowledge gap. We start by dividing these algorithms into three groups (depending on the metric they use to detect congestion): loss-based, delay-based, and hybrid congestion control algorithms. Next, we reveal that resources are rarely distributed fairly through head-to-head comparisons, especially when flows sharing a link have different round-trip times or belong to different groups.

This chapter is based on a published conference paper: B.Turkovic, F.A. Kuipers, S. Uhlig, *Interactions between Congestion Control Algorithms*, 2019 Network Traffic Measurement and Analysis Conference (TMA) 2019 [179] and its extended version: B. Turkovic, F.A. Kuipers, S. Uhlig, *Fifty shades of congestion control: A performance and interactions evaluation*, arXiv preprint arXiv:1903.03852, (2019) [180]

5.1. INTRODUCTION

When a packet arrives at a switch, it is processed based on the installed forwarding rules and forwarded to an output link. However, since output links have a fixed bandwidth, a network node receiving more traffic than the output link can send further along the network will become congested. Hence, each connection's maximum rate is limited by the so-called bottleneck link on the path, i.e., the output link with the least amount of available resources to process that flow.

To detect this point, during the connection, end-hosts continuously probe for resources by increasing their sending rate. This process continues until a connection reaches the previously mentioned bottleneck bandwidth (Application-limited domain, Figure 5.1). By increasing the sending rate further, buffers in the network nodes start to fill, and queues might form (Queueing domain, Figure 5.1). Packets arrive at the bottleneck faster than they can be forwarded, causing an increased delay, while the delivery rate remains the same. Finally, when buffers are full, the network node has to drop packets (Dropping domain, Figure 5.1). Increasing buffer size will not improve the network's performance and instead will lead to bufferbloat, i.e., the formation of queues in the network devices that unnecessarily add delay to every packet passing through.

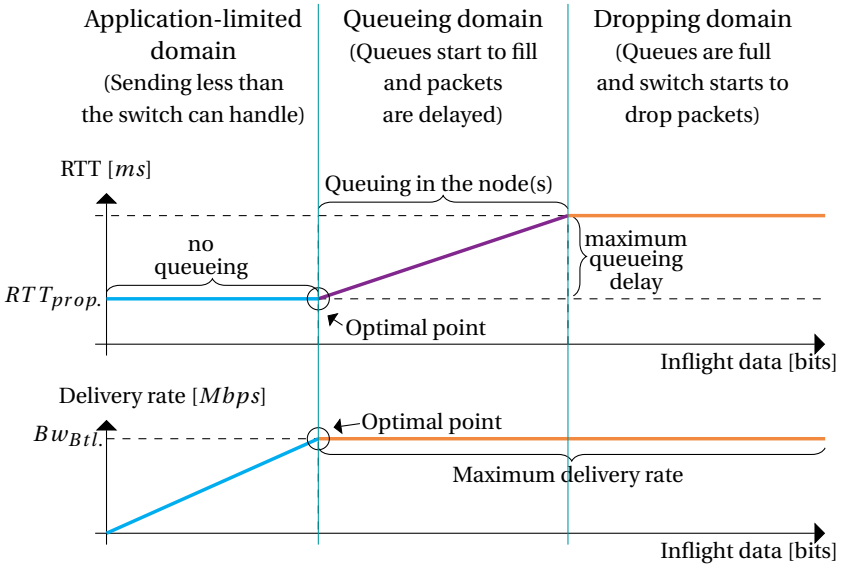


Figure 5.1: Effect of the number of packets sent on the RTT (top) and delivery rate (bottom). Based on [181, 182].

In the wake of the growing demand for higher bandwidth, higher reliability, and lower latency, novel congestion control algorithms have been developed and deployed. For example, in 2016, Google published its bottleneck bandwidth and round-trip time (BBR) congestion control algorithm, claiming it was able to operate without creating packet loss or filling buffers [181]. Around the same time, TCP LoLa [32] and TIMELY [33] were proposed, focusing on low latency and bounding of the queuing delay. Moreover,

new transport protocols such as QUIC allow the implementation of algorithms directly in user-space, facilitating the rapid development of new transport features. However, *congestion control algorithms have been typically developed in isolation*, without thoroughly investigating their behavior in the presence of other congestion control algorithms, which is the main goal of this chapter.

5.2. MAIN CONTRIBUTIONS

In this chapter, we first divide existing congestion control algorithms into three groups: loss-based, delay-based, and hybrid. Based on experiments in a testbed, we study the interactions over a bottleneck link among flows of the same group, across groups, as well as when flows have different Round-Trip Times (RTTs). We find that flows using loss-based algorithms are over-powering flows using delay-based, as well as hybrid algorithms. Moreover, when flows using loss-based algorithms fill the queues, an increase in queuing delay of all the other flows sharing the bottleneck is determined by their presence. Consequently, non-loss-based groups cannot be used in a typical network, where flows typically rely on a loss-based algorithm. In addition, we observe that convergence times can be large, which may surpass the flow duration for many applications. Finally, we discover that hybrid algorithms, such as BBR, not only favor flows with a higher RTT, but they also cannot maintain a low queuing delay as promised.

In Section 5.3, we provide an overview and classification of congestion control mechanisms. In Section 5.4, we (1) identify a set of key performance metrics to compare them, (2) describe our measurement setup, and (3) present our measurement results. Additional measurements are given in [183].

5.3. CLASSIFICATION

Since the original TCP specification (RFC 793 [29]), numerous congestion control algorithms have been developed. In this chapter, we focus mostly on algorithms designed for wired networks. The algorithms we consider can be used both by QUIC and TCP and can be divided into three main groups (see Figure 5.2): (1) loss-based algorithms that detect congestion when buffers are already full and packets are dropped, (2) delay-based algorithms that rely on RTT measurements and detect congestion by an increase in RTT, indicating buffering, and (3) hybrid algorithms that use some combination of the previous two methods.

5.3.1. LOSS-BASED ALGORITHMS

The original congestion control algorithms from [29] were loss-based algorithms with TCP Reno being the first widely deployed one. With the increase in network speeds, Reno's conservative approach of halving the congestion window became an issue. TCP connections were unable to fully utilize the available bandwidth, so that other loss-based algorithms were proposed, such as NewReno [30], Highspeed-TCP (HS-TCP [36]), Hamilton-TCP (H-TCP [37]), Scalable TCP (STCP [42]), Westwood (TCPW [49]), TCPW+ (TCP Westwood+ [50]), TCPW-A [51], and LogWestwood+ [44]. They all improved upon Reno by including additional mechanisms to probe for network resources more aggressively. However, they also react more conservatively to loss detection events, and dis-

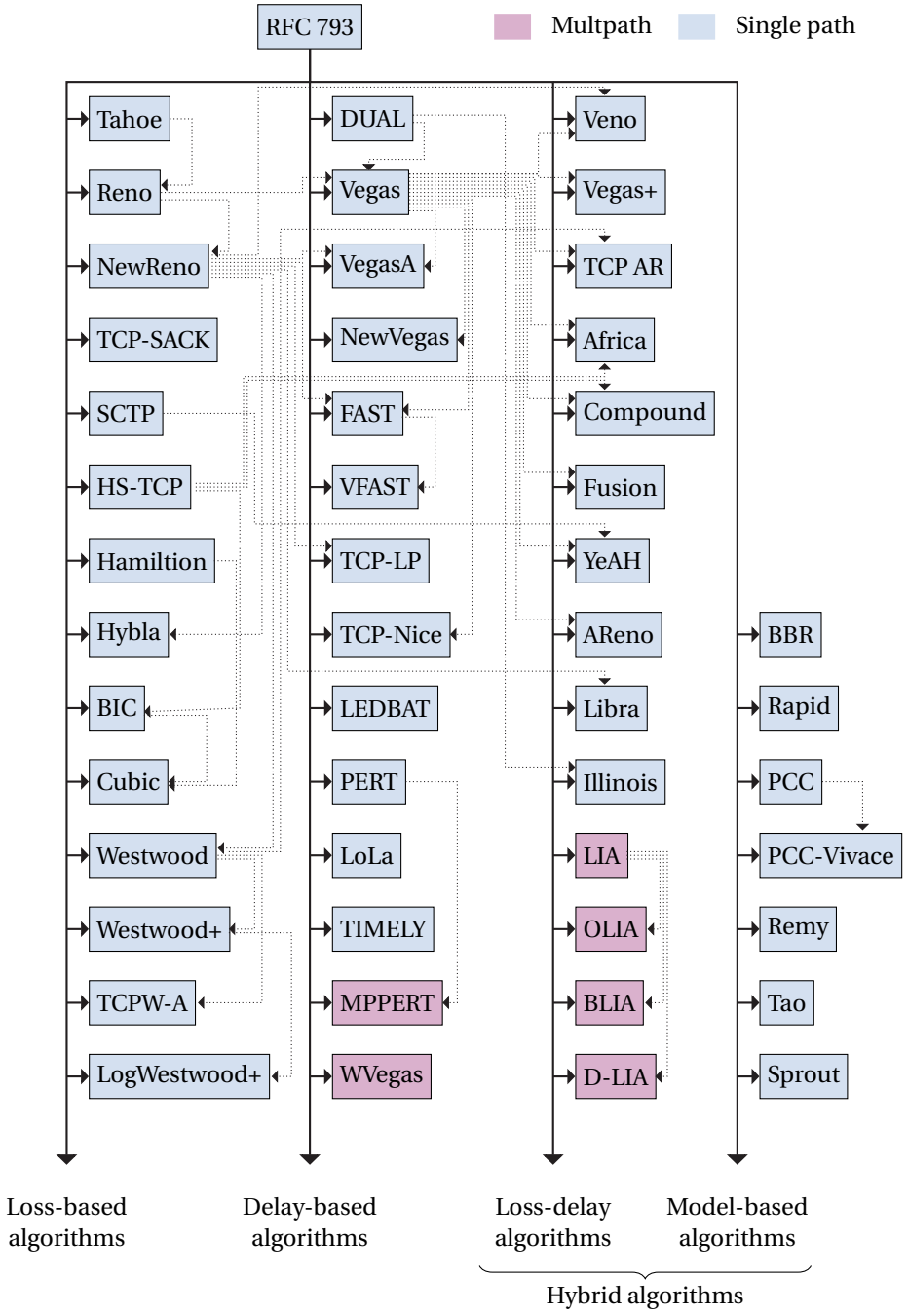


Figure 5.2: Classification of different congestion control algorithms. Dotted arrows indicate that one was based on the other.

criminate between different causes of packet loss.

However, these improvements did not address any of the existing RTT-fairness issues, but introduced new ones [56, 57]. Indeed, when two flows with different RTTs share the same bottleneck link, the flow with the lowest RTT is likely to obtain more resources than other flows. To resolve this issue, BIC [56] and Hybla [57] were proposed. Hybla modified NewReno's Slow Start and Congestion Avoidance phases and made them semi-independent of RTT. However, the achieved RTT-fairness meant that flows with higher RTTs behaved more aggressively. The main idea of BIC was to use a binary search algorithm to approach the optimal congestion window size. However, later evaluations showed that BIC can still have worse RTT-fairness than Reno [48]. In response, Cubic was proposed in [48]. Since **Cubic** is the current default algorithm in the Linux kernel, we will use it as a reference for loss-based algorithms throughout this thesis.

5.3.2. DELAY-BASED ALGORITHMS

In contrast to loss-based algorithms, delay-based algorithms are proactive. They try to find the point when the queues in the network start to fill, by monitoring the variations in RTT. An increase in RTT, or a packet drop, causes them to reduce their sending rate, while a steady RTT indicates a congestion-free state. Unfortunately, RTT estimates can be inaccurate due to delayed ACKs, cross traffic, routing dynamics, and queues in the network [33, 184].

The first algorithm that used queuing delay as a congestion indicator was TCP Dual. The first improvement to this algorithm was Vegas [185]. It focuses on estimating the number of packets in the queues and keeping it under a certain threshold. However, several issues were identified. First, when competing with existing loss-based algorithms, Vegas flows suffer from a huge decrease in performance [186, 187]. Second, it has a bias towards new flows and, finally, interprets rerouting as congestion [187]. To address these issues several modifications to Vegas were proposed, including VegasA [187], Vegas+ [186], FAST [188], VFAST [189], and NewVegas [190].

Recently, as low latency and service differentiation became important, several new algorithms have been proposed. TCP-LP [191, 192], TCP-Nice and LEDBAT [193] focused on the differentiation between high-priority (foreground) and low-priority (background) flow. Moreover, Hock et al. designed LoLa [32], focusing on low latency and convergence to a fair share between flows. To improve performance in datacenter networks, Google proposed TIMELY [33], which relies on very precise RTT measurements. Since **Vegas** is used as the base algorithm by many other delay-based and hybrid algorithms, we use it as a reference for delay-based algorithms.

5.3.3. HYBRID ALGORITHMS

Hybrid algorithms use both loss and delay as congestion indicators. The first hybrid algorithm was Veno [194]. It is a modification of the Reno congestion control that extends the additive increase and multiplicative decrease functions by also using queuing delay as the secondary metric. To efficiently utilize the available bandwidth in high-speed networks, many algorithms use similar modifications based on the Vegas or Dual network state estimations. Some of the most important ones are Africa [195], Compound [196], and YeAH [197]. Other algorithms modify the congestion window increase function to

follow a function of both the RTT and the bottleneck link capacity, such as Illinois [198], AR [199], Fusion [200], TCP-Adaptive Reno (AReno) [201], and TCP Libra [202].

However, in recent years, a set of new algorithms, not relying on the standard AIMD mechanism but building a network model using the previously mentioned metrics has emerged. In 2016, Google developed the bottleneck bandwidth and round-trip time (BBR) algorithm. However, several problems, mostly related to the Probe RTT phase, were discovered: (1) bandwidth can be shared unfairly depending on the timing of new flows and their RTT, and (2) unfairness towards other protocols, especially Cubic [182, 203, 204]. At the same time, a new approach to congestion control using online learning was proposed in PCC [45]. In this thesis, we will refer to these new hybrid algorithms as model-based algorithms. Similarly to all of the hybrid algorithms that still use the standard additive-increase/multiplicative-decrease (AIMD) algorithm, we will refer to this group as a loss-delay sub-group.

We use **BBR** as our representative for hybrid algorithms, since it is actually deployed (in Google's network) and implemented in the Linux kernel (since v4.9).

5.4. EVALUATION

Using the metrics described in Section 5.4.1 and via the set-up described in Section 5.4.2, in Sections 5.4.3 and 5.4.4 we evaluate the representatives of the three algorithm groups (Cubic, Vegas and BBR). Additional measurements and results of all other algorithms that have been implemented in the Linux kernel can be found in [183].

5.4.1. PERFORMANCE METRICS

To be exhaustive, we use the following metrics to compare different congestion control algorithms:

- **Sending rate.** Sending rate represents the bit-rate (incl. data-link layer overhead) of a flow generated by the source, per time unit.
- **Throughput.** Throughput measures the number of bits (incl. the data-link layer overhead) received at the receiver, per time unit.
- **RTT (round-trip time).** RTT represents the time between sending a packet and receiving an acknowledgement of that packet.
- **Goodput.** Goodput measures the amount of useful data (i.e., excl. overhead) delivered by the network between specific hosts, per time unit. This value is an indicator of the application-level QoS experienced by the end-users as their desire is to get the resources they requested in the shortest possible time. Goodput is defined as:

$$\text{Goodput} = \frac{(D_s - D_r - D_o)}{\Delta t} \tag{5.1}$$

where D_s is the number of useful bits transmitted, D_r the number of bits retransmitted and D_o the number of overhead bits in the time interval Δt. Additionally, we use the **goodput ratio**, i.e., the amount of useful data transmitted divided by

the total amount of data transmitted, computed as:

$$\text{Goodput\_ratio} = \frac{(D_s - D_r - D_o)}{D_s} \tag{5.2}$$

While goodput is comparable to throughput, it excludes packets that were retransmitted or dropped as well as protocol overheads.

- **Fairness.** Fairness describes how the available bandwidth is shared among multiple users. Ideally, each flow is allocated either the required amount of bandwidth, if available, or a fair portion if the total demand of all flows exceeds the available bandwidth. We consider three different types of fairness: (1) **intra-fairness** describes the resource distribution between flows running the same congestion control algorithm; (2) **inter-fairness** describes the resource distribution between flows running different congestion control algorithms, and (3) **RTT-fairness** describes the resource distribution between flows having different RTTs. Fairness is represented by Jain's index [205] and is computed as:

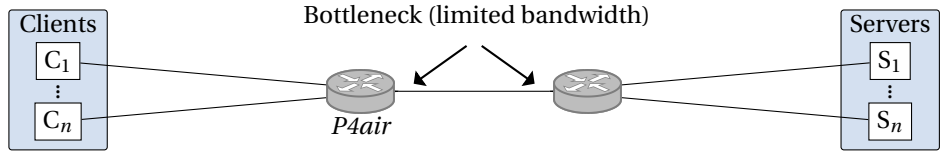
$$F = \frac{(\sum_{i=1}^{n} x_i)^2}{n \cdot \sum_{i=1}^{n} (x_i)^2} \tag{5.3}$$

where x_i is the throughput of the flow i. This index is based on the throughput and indicates how fair the available bandwidth at the bottleneck is shared between all flows present and ranges from $1/n$ (worst case) to 1 (best case), where n is the number of flows.

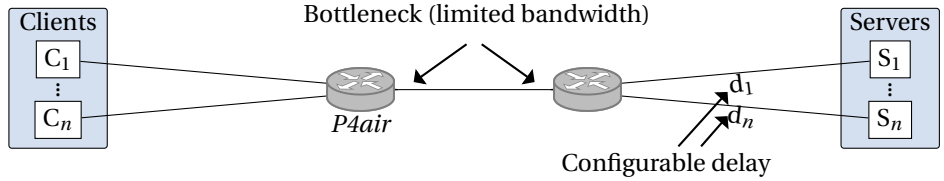
5.4.2. EXPERIMENT SETUP

Each server in our testbed has a 64-bit Quad-Core Intel Xeon CPU running at 3GHz with 4GB of main memory and has 6 independent 1 Gbps NICs. Each server can play the role of a 6-degree networking node. All nodes run Linux with kernel version 4.13 with the txqueuelen set to 1000, and were connected as shown in Figure 5.3 with degree $1 \leq n \leq 4$ (consequence of the limited number of NICs per server in the testbed). Given that the performance of congestion control algorithms is affected by the bottleneck link on the path, such a simple topology is sufficient for our purposes. The maximum bandwidth and the bottleneck (between s1 and s2) was limited to a pre-configured value ($100Mbps$ in the case of TCP and $10Mbps$ in the case of QUIC to make sure that the sending rate of the end-user applications is enough to saturate the bottleneck link) with the use of ethtool. To perform measurements, we rely on tshark, iperf, QUIC client and server (available in the Chromium project [206]) and socket statistics. From traffic traces (before and after the bottleneck), we calculate the metrics described in Section 5.4.1. All the values are averaged per flow, using a configurable time interval. We consider the following two scenarios:

- **BW scenario.** The purpose of this scenario was to test the interoperability of different congestion control algorithms. Each analyzed algorithm is compared to itself and all others. Host C_i generates TCP flows towards servers running at S_i using different congestion control algorithms (Figure 5.3a).



(a) BW Scenario.



(b) RTT scenario.

Figure 5.3: Experiment topology.

5

- **RTT scenario with flows having different RTTs.** The purpose of this scenario is to test the RTT-fairness of different congestion control algorithms. In addition to the setup of the previous scenario, the delay at links between S_i and node 2 is artificially increased using Linux TC (adding $0-400ms$, Figure 5.3b).

We ran these scenarios five times. For all of them, the results we observe lead to qualitatively similar interactions, as presented in Sections 5.4.3 and 5.4.4.

5.4.3. BW SCENARIO

Intra-Fairness. Delay-based and loss-based algorithms have the best intra-fairness properties, with an average fairness index within $0.94-0.95$ (Table 5.1). Figure 5.4 shows that Jain's index is always close to 1, indicating that all present flows receive an equal share of the resources. In addition, delay-based algorithms operate without filling the buffers, in contrast to the loss-based algorithms that periodically fill the buffers and drop packets (Figures 5.4 and 5.6). Further, the convergence time of loss-based algorithms is higher ($\approx 20\ s$, compared to $5s$ needed for 2 Vegas flows) and their throughput oscillates the most from all the evaluated approaches (Figure 5.4). When the number of Cubic flows increases to 4, bandwidth oscillations increase as well, and fairness decreases to 0.82 [183].

In contrast, hybrid-based algorithms (BBR) unexpectedly had the worst intra-fairness properties. Figure 5.4 shows that they rarely converge to the same bandwidth, but oscillate between 30 $Mbps$ and 70 $Mbps$ (every probeRTT phase), even in scenarios in which they claim a similar share of the available resources on average. The flow that measures a higher RTT adopts a more aggressive approach and claims more resources, even if the measured RTT difference is very small ($\leq 0.5ms$). Hence, they are not particularly stable. Unexpectedly, when the number of flows increases to 4, the fairness index improves, and although oscillations go down they are still present.

Inter-Fairness. As expected, flows that use delay-based algorithms experience a huge

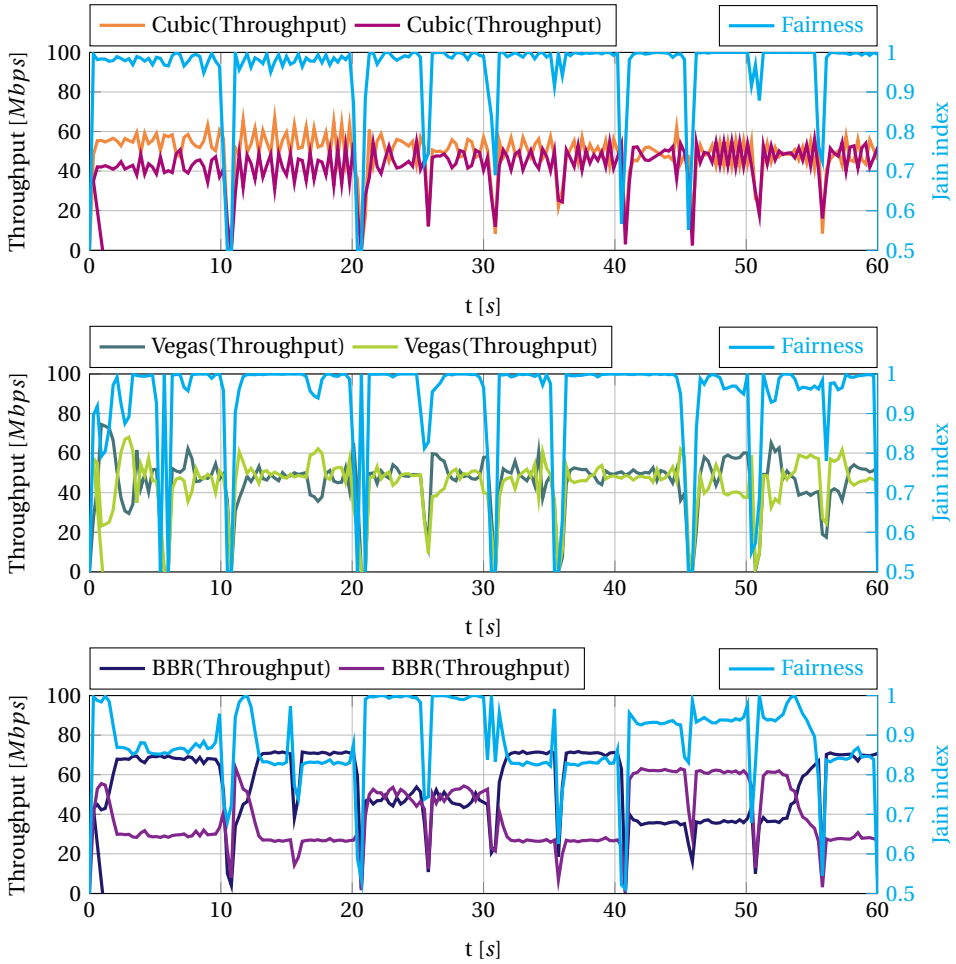


Figure 5.4: BW scenario: Comparison of average RTT, average throughput, and fairness index for representatives of the congestion control algorithm classes groups in case the link is shared by 2 flows using the same algorithm (time unit 300ms).

decrease in throughput if they share the bottleneck with loss-based flows (Figure 5.5). This is because they detect congestion earlier, at the point when the queues start to fill. Loss-based algorithms on the other hand continue to increase their sending rate as no loss is detected. This increases the observed RTT (Figure 5.6) of all flows, triggering the delay-based flow to back off [186, 187]. As a consequence, only a few hundred milliseconds after the start of the connections, delay-based algorithms reduced their sending rate to 1/10 – 1/15 of the sending rate of the loss-based algorithms. This process continued until almost no resources were available for the delay-based algorithm (Figure 5.5).

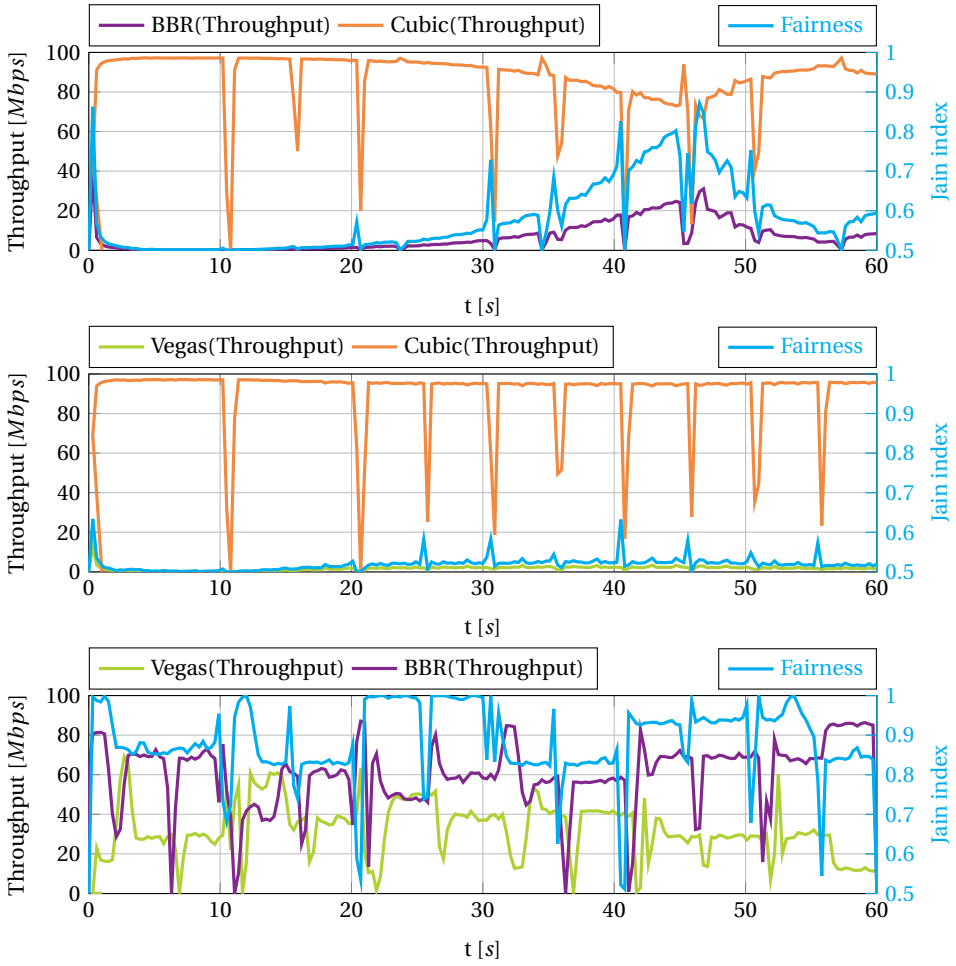


Figure 5.5: BW scenario: Comparison of average throughput and fairness index for representatives of the congestion control algorithm groups in case the link is shared by 2 flows using different algorithms (time unit 300ms).

A similar behaviour is observed when a bottleneck is shared between flows from a hybrid and a delay-based algorithm: BBR outperforms Vegas. However, the difference in the throughput is less significant than the one observed in the previous scenario, with the Vegas flow claiming almost $40Mbps$ on average (Table 5.1). When we increase the number of Vegas or BBR flows at the bottleneck to four, the new flows increase their bandwidth at the expense of the BBR flow, reducing its share from $50Mbps$ down to $20Mbps$, and increasing the fairness index to $0.9-0.94$ [183]. This is a consequence of the fact that BBR tries to operate without filling the queues, allowing the delay-based algorithm to grow and claim more bandwidth. Thus, we conclude that, in contrast to loss-based algorithms, delay-based algorithms can co-exist with hybrid-based ones.

Table 5.1: BW scenario with 2 flows: Different metrics for representatives of the three congestion control algorithm groups (calculated for 5 different runs).

Protocol	Group	Algorithm	Average goodput [$Mbps$]	Average goodput ratio [%]	Average RTT [$\#packets$]	Average sending rate [ms]	Average throughput [$Mbps$]	Average Jain index [$Mbps$]
TCP	Loss- vs. Loss-based	Cubic Cubic	44.98 43.15	93.57 93.78	76.65 78.32	48.77 50.98	46.59 46.59	0.95
	Delay- vs. Delay-based	Vegas Vegas	43.81 42.72	94.81 94.76	1.66 1.68	48.65 49.79	45.47 44.38	0.94
	Hybrid vs. Hybrid	BBR BBR	44.98 42.72	92.32 94.39	3.21 3.24	52.18 46.89	46.70 44.36	0.86
	Loss-based vs. Hybrid	Cubic BBR	82.29 7.56	94.27 88.86	70.37 174.38	90.91 8.87	85.05 7.89	0.59
	Loss- vs. Delay-based	Cubic Vegas	87.73 1.74	94.34 91.57	67.16 139.79	97.30 2.00	90.66 1.82	0.52
	Delay-based vs. Hybrid	Vegas BBR	38.37 48.56	94.34 94.68	4.55 4.25	37.31 61.65	39.83 50.37	0.84

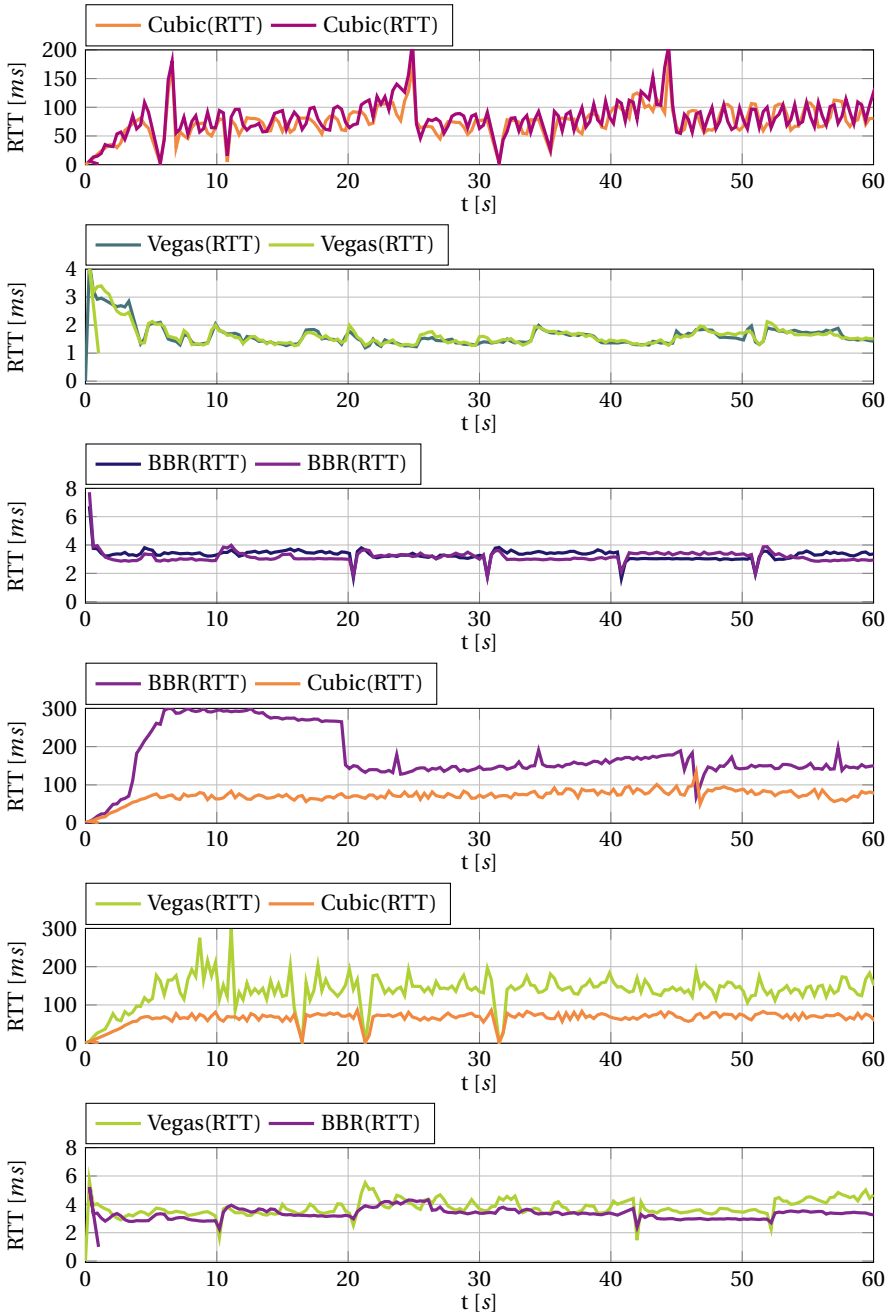


Figure 5.6: BW scenario: Comparison of average RTT for representatives of the congestion control algorithm classes groups (time unit 300ms).

When the bottleneck is shared between a hybrid and a loss-based algorithm, Cubic outperforms BBR, reducing its share of resources to as little as 8% on average (Table 5.1), confirming results from [207]. The fairness index at the start of the connection is very low as Cubic claims all the available bandwidth at the expense of the BBR flow. After the Cubic flow fills the buffers, BBR measures an increased RTT and adopts, as a consequence, a more aggressive approach (Figures 5.6 and 5.5). However, packet loss triggers Cubic's back-off mechanism, allowing BBR to measure a lower RTT estimate. Consequently, BBR reduces its rate, allowing the Cubic flow to claim more bandwidth again.

Moreover, when we increase the number of Cubic flows to three, the throughput of the BBR flow drops close to zero. Similarly, even three BBR flows are not able to compete with one Cubic flow, with each of them claiming approximately 5% of the total bandwidth on average [183].

Delay. Even if one loss-based algorithm is present at the bottleneck, the observed delay is determined by it, nullifying the advantages of delay-based and hybrid algorithms, namely the prevention of the queue buildup. BBR, as well as Vegas, which claim to be able to operate with a small RTT, suffer from a huge increase in average RTT (by more than 100 ms, Table 5.1) when competing with Cubic (compared to $1-5ms$ without Cubic). However, when a link is shared between a hybrid and a delay-based flow, both of them are able to maintain a low RTT. In such scenarios, hybrid algorithms, such as BBR, due to their more aggressive approach compared to delay-based algorithms, determine the RTT. Vegas flows, as a consequence, suffer from a small increase in RTT (from 1.68ms to 4.55ms, Table 5.1).

Summary. In terms of fairness, the only combination that works well together is delay and hybrid algorithms. In such a scenario, delay is low and the throughput fairly shared, the more flows the fairer the distribution of resources. Hybrid, as well as delay-based algorithms, suffer from a huge increase in the observed delay if even one loss-based algorithm is present at the bottleneck making them unusable in typical networks consisting of many different flows. We observe that the most popular TCP flavour, Cubic, is prone to oscillation and has a high convergence time ($\approx 20s$). Further, we observe that BBR is not stable, reacting to very small changes in the observed RTT, which was not previously reported in the literature.

5.4.4. RTT SCENARIO

We observe RTT-fairness issues for all three groups of algorithms. Even though loss-based algorithms such as Cubic claim good RTT-fairness properties, they favour the flow with a lower RTT [208]. This is most noticeable when analyzing two Cubic flows in Figure 5.7. Even when the number of flows increases to 4 (Figure 5.8), the flow with the lowest RTT immediately claims all the available resources, leaving less than half to the other flows in the first 30 s. Several improvements addressing this problem, such as TCP Libra [202] have been proposed. However, current kernel implementations do not capture these improvements.

The fairness index for delay-based algorithms slowly increases over time, but due to a very conservative congestion avoidance approach of Vegas, even after 60s, flows do not converge (Figure 5.7). When we increase the number of Vegas flows to four, the dynamics at the bottleneck become more complex with the newest flow (with the highest

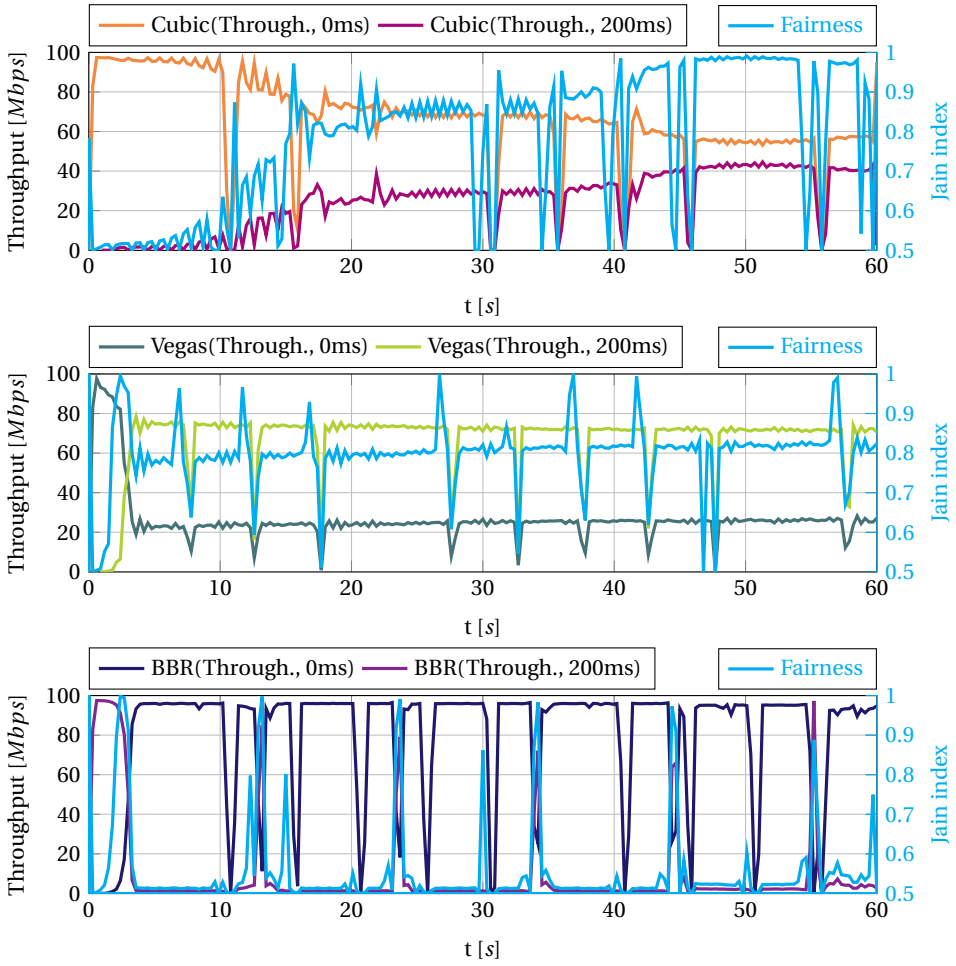


Figure 5.7: RTT scenario: Comparison of average throughput, and fairness index for representatives of the congestion control algorithm classes in case the link is shared by 2 flows using the same algorithm (time unit 300ms).

RTT) claiming the largest share of resources at the end (Figure 5.8). Moreover, contrary to the previous scenarios, in the slow start phase, Vegas flows fill the bottleneck queue and the observed queuing delay increases to $70ms$. However, after $30s$ the queues are drained, fairness improves, and the observed queuing delay is very low for all flows ($2-3ms$, Figure 5.9).

Hybrid-based algorithms, such as BBR, favour the flow with the higher RTT, confirming results from [182, 207]. The flow with a higher RTT overestimates the bottleneck link, claiming all the available resources and increasing the queuing delay (Figure 5.7) by a factor of more than 10 (from $\approx 4ms$ to $\approx 50ms$). Moreover, when we increase the

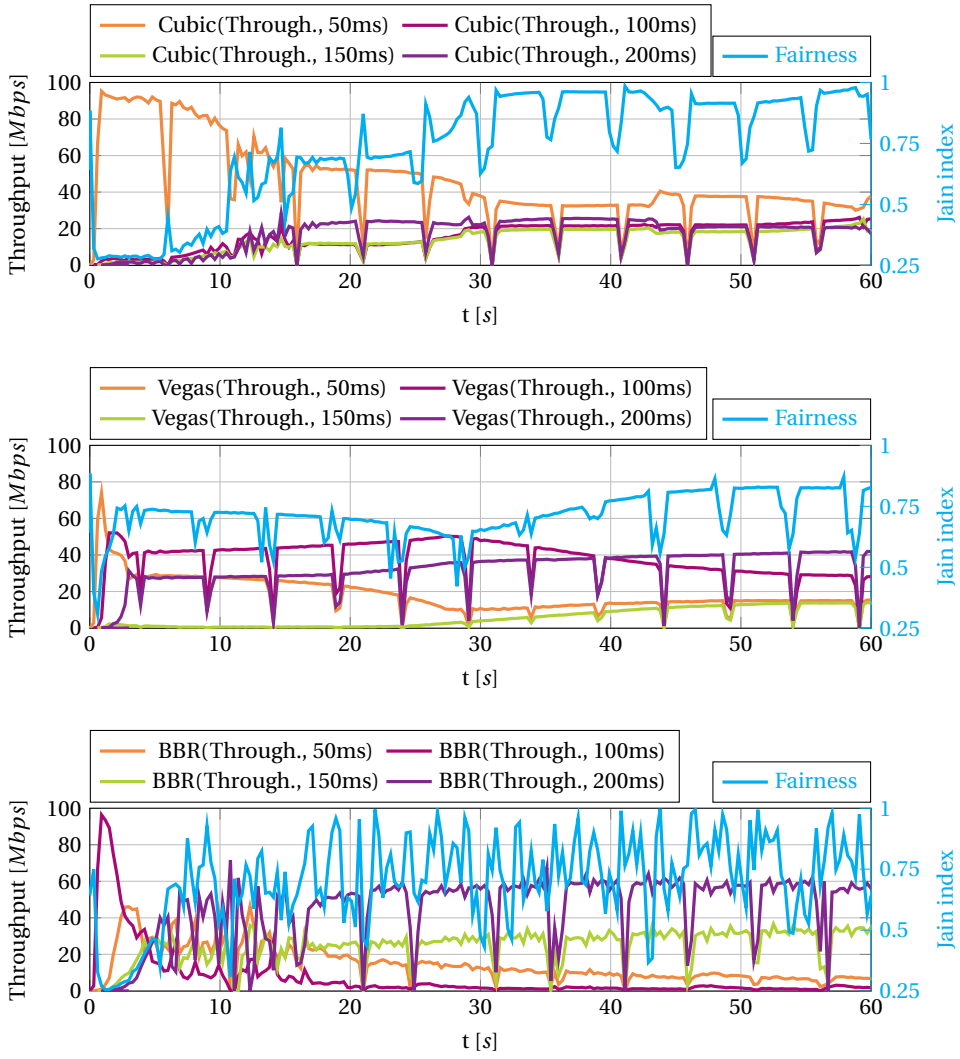


Figure 5.8: RTT scenario: Comparison of average throughput, and fairness index for representatives of the congestion control algorithm classes in case the link is shared by 4 flows using the same algorithm (time unit 300ms).

number of BBR flows to four, contrary to expectations, the average RTT increases significantly (by a factor of almost 30) reaching values comparable to the ones observed by the loss-based algorithms in the same scenario although only BBR flows were present at the bottleneck (Figure 5.8, Table 5.3).

Table 5.2: RTT scenario: Different metrics for representatives of the congestion control algorithm groups in case the link is shared by two flows using the same algorithm (calculated for 5 different runs).

Protocol	Group	Algorithm	Average goodput [$Mbps$]	Average goodput ratio [%]	Average RTT [$\#packets$]	Average sending rate [ms]	Average throughput [$Mbps$]	Average Jain index [$Mbps$]
TCP	Loss- vs. Loss-based	Cubic(0ms)	65.67	94.07	233.09	75.47	67.88	0.76
		Cubic(200ms)	21.88	93.80	435.53	25.36	22.92	
	Delay- vs. Delay-based	Vegas(0ms)	14.99	94.21	32.03	18.91	15.62	0.66
		Vegas(200ms)	72.60	94.31	228.96	81.48	75.08	
	Hybrid vs. Hybrid	BBR(0ms)	8.90	91.98	50.08	9.87	9.24	0.56
		BBR(200ms)	79.54	94.39	249.56	90.97	82.1	

Table 5.3: RTT scenario: Different metrics for representatives of the congestion control algorithm classes in case the link is shared by four flows using the same algorithm (calculated for 5 different runs).

Protocol	Group	Algorithm	Average goodput [$Mbps$]	Average goodput ratio [%]	Average RTT [$\#packets$]	Average sending rate [ms]	Average throughput [$Mbps$]	Average Jain index [$Mbps$]
TCP	Loss-based	Cubic(50ms)	47.48	93.86	216.60	53.66	49.59	0.69
		Cubic(100ms)	15.32	92.39	264.99	17.71	16.09	
		Cubic(150ms)	11.70	91.62	316.87	13.62	12.32	
		Cubic(200ms)	13.68	92.33	368.14	15.78	14.39	
	Delay-based	Vegas(50ms)	27.32	92.98	94.50	31.09	28.54	0.62
		Vegas(100ms)	41.85	93.88	144.11	47.13	43.63	
		Vegas(150ms)	7.50	90.80	196.87	8.62	7.90	
		Vegas(200ms)	11.57	91.47	245.18	13.22	12.16	
	Hybrid	BBR(50ms)	7.11	88.01	203.63	42.56	7.44	0.63
		BBR(100ms)	15.23	91.61	253.49	21.43	16.06	
		BBR(150ms)	22.20	93.59	302.70	18.81	23.45	
		BBR(200ms)	42.39	94.18	353.22	15.97	44.70	

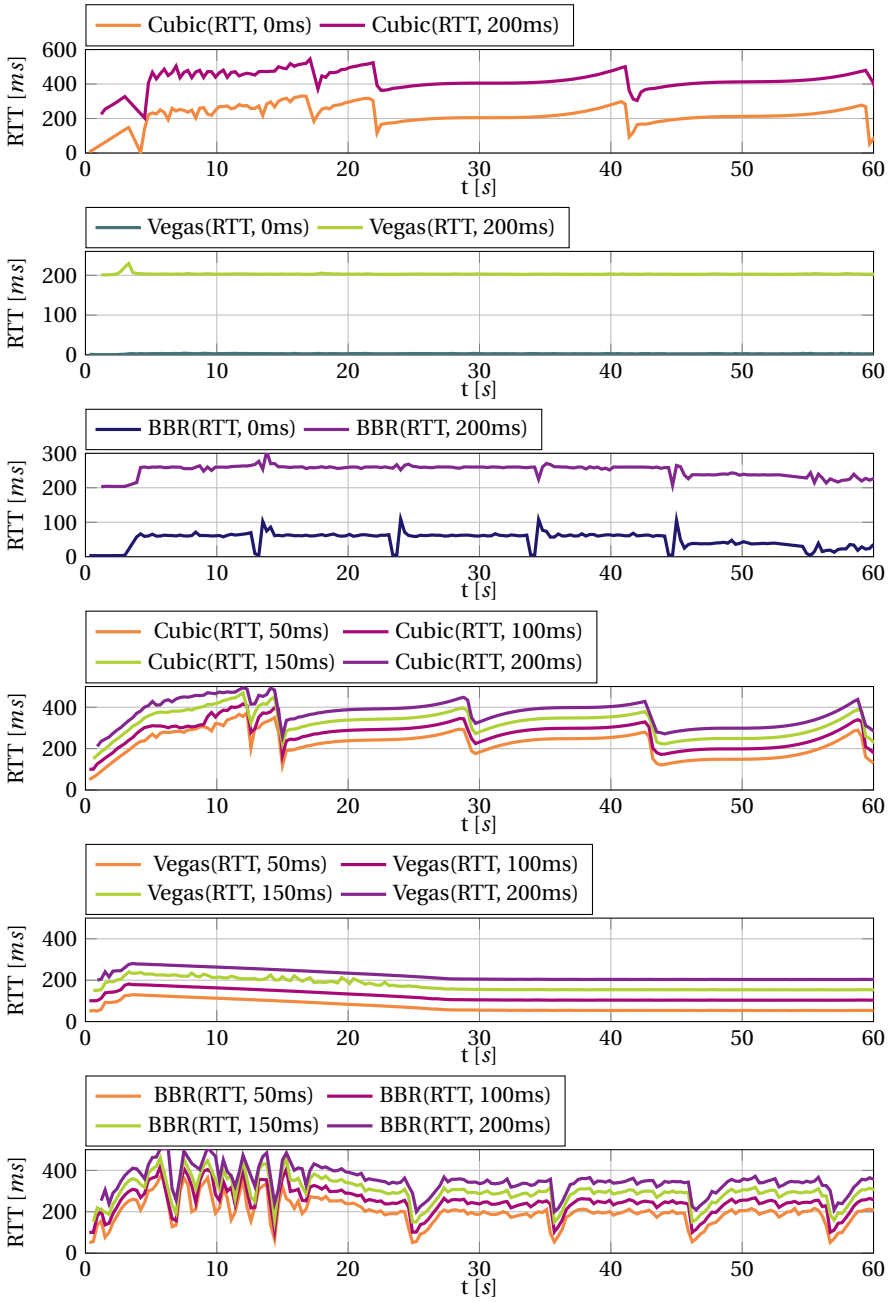


Figure 5.9: BW scenario: Comparison of average RTT for representatives of the congestion control algorithm classes groups (time unit 300ms).

Summary. We observe that RTT-fairness is poor for all groups of algorithms. Delay-based algorithms are the only ones that can maintain a low delay compared to the other two groups. However, they still do not converge towards their fair share. Loss-based algorithms such as Cubic perform poorly, contrary to expectations and their own claims, favouring flows with lower RTTs. When loss-based algorithms converge to a fair share, the convergence time is so slow that the average fairness index is still low (0.69 on average). Finally, hybrid algorithms such as BBR suffer from significant dynamics in the sharing among its own flows, favoring those with higher RTT and significantly increasing the queuing delay. Hence, we observe that even when only BBR flows are present on the bottleneck, the claim of being able to operate without filling the buffers is not true.

5.4.5. RESULTS: QUIC

When QUIC is used with different congestion control algorithms, we observe similar interactions as earlier. With BBR, we observe the same RTT-unfairness properties as with the TCP BBR, which always favours the flows with a higher RTT (with an average fairness index of 0.59). Similarly, QUIC with Cubic always favours the flow with a lower RTT. However, the difference between the throughput of the two QUIC Cubic flows is much smaller than the one observed for the TCP equivalent, with an average fairness index of 0.93. In all our QUIC scenarios where hybrid (BBR) and loss-based (Cubic) flows compete, Cubic outperforms BBR. Over time, as QUIC BBR flows detect a higher RTT and adopt a more aggressive approach, BBR grabs more bandwidth at the expense of the Cubic flows. However, this process is slow and the throughput of the BBR flow remains low. Detailed measurements of QUIC can be found in [183].

5.5. CONCLUSION

After dividing existing congestion control algorithms into three groups (loss-based algorithms, delay-based algorithms, and hybrid algorithms), we studied their interactions.

We observed multiple fairness issues, among flows of the same group, across different groups, as well as when flows having different RTTs were sharing a bottleneck link. We found that delay-based, as well as hybrid algorithms, suffer from a decrease in performance when competing with flows from the loss-based group, making them unusable in a typical network where the majority of flows will rely on a loss-based algorithm. Not only do they get an unfair share of the available bandwidth, but they also suffer from a huge increase in the observed delay when the loss-based algorithms fill the queues. The only combination that worked well together was delay and hybrid algorithms: the observed RTT was low and resources shared fairly (the more flows the fairer the distribution of resources). Finally, we found that hybrid algorithms, such as BBR, are very sensitive to changes in the RTT, even if that difference is very small ($\leq 0.5ms$). They not only favour the flow with a higher RTT at the expense of the other flows, but they also cannot maintain a low queuing delay as promised even if they are the only flows present in the network.

Therefore, our work shows that to support applications that require low latency, a good congestion control algorithm on its own won't be enough, especially since most networks typically process flows using different congestion control algorithms. Further,

as congestion control algorithms are not determined by the network but by the end-hosts, they can never be enforced by the network operator. However, having all the end-host agree on the same protocol is complex, and even then, does not necessarily guarantee fairness (see Section 5.4.4). Moreover, newer transport protocols such as QUIC, enable the end-users to design their own congestion protocol directly in user-space, thereby only increasing the already present diversity in the networks. Hence, guaranteeing that flows of a given group (in terms of the type of congestion control) will receive their expected share of resources requires that resource isolation be provided between the different groups, which is explored in more detail in the following chapter.

6

P4AIR: INCREASING FAIRNESS AMONG CONGESTION CONTROL ALGORITHMS

In the previous chapter, we showed that congestion control algorithms—because they are usually developed in isolation rather than with reference to interactions with other protocols and algorithms—tend to overpower each other. This results in unfair resource distribution, with a subset of the flows usually claiming most resources.

To solve this problem, we use programmable switches and the network programming language P4 to enforce fairness from within the network itself, instead of from the congestion control algorithms that run at the end-points. Our solution P4air*, continuously monitors the properties of all flows that pass through a switch and groups them based on the behavior of the congestion control algorithms used. Furthermore, it applies appropriate measures to suppress the aggressive flows and boost the smaller flows for each group. Using modern programmable hardware (Intel Tofino switch), our experiments demonstrate that in terms of fairness, P4air performs significantly better than current state-of-the-art solutions.*

This chapter is based on a published conference paper: B.Turkovic, F.A. Kuipers, *P4air: Increasing Fairness among Competing Congestion Control Algorithms*, 28th IEEE International Conference on Network Protocols (ICNP), (2020) [77]

6.1. INTRODUCTION

The field of congestion control – a key component of transport-layer protocols – continues to see innovation through many novel proposals, each claiming superiority for specific applications or scenarios [27, 31–35, 38–41, 43, 45–47, 52–55, 58, 209]. Furthermore, new transport protocols, such as QUIC and MPQUIC, facilitate the rapid development of new transport features directly in the user space, enabling even more customization and more diverse network protocols in the future [59–61].

However, due to this abundance of new protocols and algorithms, it has become almost impossible to take their interactions with other protocols and algorithms into account. Consequently, even for algorithms designed with good fairness properties in mind, multiple fairness issues exist, especially when bottleneck links are shared between flows using different congestion control algorithms or having different Round-Trip Times (RTTs) [57, 183, 187, 210–215]. For example, classic TCP flows (using loss-based algorithms) fill the bottleneck queues (resulting in high queuing delay) and only react to the resulting packet loss. These algorithms overpower newer congestion control algorithms that also take delay measurements into account, nullifying their inherent advantages (e.g., low queuing delay) and making them unusable in a typical network, where the majority of flows still rely on loss-based algorithms. Furthermore, even when only flows using newer algorithms are present at the bottleneck, fairness can still be low, especially among flows having different RTTs.

Active queue management (AQM) solutions [62, 66, 67, 71–73, 76] have been proposed to improve fairness by deploying different dropping policies at the bottleneck. They detect congestion in the queue buildup phase, improving the end-to-end latency, and forcing the most aggressive flows to back off. However, by doing so, they only target one of the many metrics congestion control algorithms use to detect congestion (i.e., packet loss). In other words, they treat all flows as loss-based and are oblivious to the specific congestion control algorithms used. This has multiple disadvantages:

1. Algorithms that do not use loss as a metric (e.g., BBRv1) are never targeted by AQMs, potentially allowing them to overpower traditional loss-based algorithms (that would back-off upon detecting loss).

2. For algorithms that use delay as their primary metric, instead of targeting the more appropriate metric (increase in RTT) and avoiding the unnecessary retransmissions, AQMs trigger a more severe back-off mechanism by targeting packet loss.

3. AQMs usually react too late, i.e., when the buffer is already partially full, and the back-off mechanism of the delay-based algorithm was already triggered (due to the increase in RTT).

Moreover, most AQM solutions target the network edge and are not well suited for the network core that simultaneously processes thousands of flows. For example, state-of-the-art algorithms have problems when operating with many flows or are too expensive to be implemented in devices, especially due to the high number of queues needed for ideal performance [216–220].

6.1.1. MAIN CONTRIBUTIONS

In this chapter, by taking advantage of the possibilities of switches with P4-programmable data planes, we develop *P4air*, a P4 application, run entirely in the data plane, that enforces fairness between all flows present on a switch. We show that *P4air*, in addition to improved fairness, can run on modern programmable hardware at line-rate (speeds reaching Tbps) without any loss of accuracy or performance.

First, we extend the analysis of the inter-, intra-, and RTT-fairness properties of congestion control algorithms from the previous chapter using all the algorithms present in the Linux kernel in Section 6.2. We use this as a base to determine the metrics *P4air* will use to classify the congestion control algorithms into the previously defined four groups with high inter-fairness properties and similar behavior. Second, in Section 6.3.1, we develop a "fingerprinting" solution that can classify, directly in the data plane, the algorithms into the previously defined groups. Furthermore, for each group, we allocate several queues. To adapt to the current network state, we develop, in Section 6.3.2, a queue reallocation algorithm (in the data plane) that favors groups with most flows by assigning more queues to them. Next, in Section 6.3.3, we complement our fingerprinting solution by developing an AQM-like solution, leveraging different metrics per group to detect congestion by applying custom actions to flows. In Section 6.4, we evaluate our solution by comparing it to different queue management techniques available on programmable hardware and implementable in P4 and show that our solution can increase fairness while maintaining high utilization. Section 6.5 highlights several deployment considerations. We present related work in Section 6.6 and conclude in Section 6.7.

6.2. CLASSIFICATION PATTERNS

6.2.1. GROUPS OF CONGESTION CONTROL ALGORITHMS

To determine the patterns *P4air* will track, we start by extending our evaluation from Chapter 5 to all available algorithms in the Linux kernel (Figure 6.1). Using the same setup as in Chapter 5, and the same fairness index (i.e., Jain's index [205]), we observe similar results, i.e., the fairness index is (1) high if flows use algorithms that rely on the same metric, or (2) low if flows use algorithms that rely on different metrics (Figure 6.1a).

First, we use these observations to further split the hybrid group of algorithms in two distinct groups with good fairness properties: called the model and the loss-delay algorithms. Next, we use these observations to identify the metrics and patterns *P4air* will track for each identified group of congestion control algorithms:

- **Purely loss-based algorithms.** Algorithms from this group, such as Reno [221] and Cubic [222] (default algorithm in the Linux kernel), only rely on packet loss to detect congestion and are the most aggressive among all the analysed groups. Queues are filled periodically and the sending rate is reduced only after detecting loss. Consequently, we choose a consistent queue build-up as the main identifying pattern of this group.

- **Delay-based algorithms.** Algorithms from this group are proactive and among the least aggressive of the analysed algorithms. They try to detect the point at which the queues start to fill and reduce their sending rate after detecting an increase

	Purely loss-based (Metric: loss)							Loss-delay (Metric: loss, delay)				Model-based	Delay-based (Metric: delay)	
	HS-TCP	STCP	HTCP	BIC	Cubic	New Reno	Hybla	YeAH	Illinois	Veno	Westwood+	BBR	Vegas	LoLa
HS-TCP	0.98	0.75	0.92	0.95	0.88	0.94	0.72	0.73	0.76	0.66	0.70	0.60	0.53	0.58
STCP	0.75	0.99	0.80	0.83	0.83	0.81	0.77	0.78	0.83	0.71	0.70	0.58	0.53	0.57
HTCP	0.92	0.80	0.99	0.84	0.96	0.99	0.81	0.88	0.88	0.78	0.86	0.57	0.52	0.56
BIC	0.95	0.83	0.84	0.98	0.80	0.85	0.66	0.68	0.66	0.61	0.67	0.59	0.53	0.67
Cubic	0.88	0.83	0.96	0.80	0.99	0.97	0.87	0.89	0.88	0.82	0.88	0.58	0.53	0.56
New Reno	0.94	0.81	0.99	0.85	0.97	0.99	0.83	0.88	0.89	0.78	0.87	0.57	0.53	0.55
Hybla	0.72	0.77	0.81	0.66	0.87	0.83	0.99	0.96	0.98	0.92	0.97	0.58	0.52	0.56
YeAH	0.73	0.78	0.88	0.68	0.89	0.88	0.96	0.99	0.98	0.92	0.97	0.62	0.52	0.56
Illinois	0.76	0.83	0.88	0.66	0.88	0.89	0.98	0.98	0.99	0.92	0.95	0.58	0.52	0.54
Veno	0.66	0.71	0.78	0.61	0.82	0.78	0.92	0.92	0.92	0.98	0.93	0.60	0.52	0.54
Westwood+	0.70	0.70	0.86	0.67	0.88	0.87	0.97	0.97	0.95	0.93	1.00	0.58	0.52	0.54
BBR	0.60	0.58	0.57	0.59	0.58	0.57	0.58	0.62	0.58	0.60	0.58	0.94	0.65	0.79
Vegas	0.53	0.53	0.52	0.53	0.53	0.53	0.52	0.52	0.52	0.52	0.52	0.65	1.00	0.67
LoLa	0.58	0.57	0.56	0.67	0.56	0.55	0.56	0.56	0.54	0.54	0.54	0.79	0.67	0.80

(a) Inter- and Intra-fairness (100 $Mbps$, 100 ms).

	Purely loss-based (Metric: loss)							Loss-delay (Metric: loss, delay)				Model-based	Delay-based (Metric: delay)	
	HS-TCP	STCP	HTCP	BIC	Cubic	New Reno	Hybla	YeAH	Illinois	Veno	Westwood+	BBR	Vegas	LoLa
0 ms	0.98	0.99	0.99	0.98	0.99	0.99	0.99	0.99	0.99	0.98	1.00	0.94	1.00	0.80
20 ms	0.79	0.92	0.94	0.74	0.84	0.85	0.89	0.86	0.92	0.91	0.86	0.56	0.83	0.73
40 ms	0.70	0.83	0.89	0.68	0.80	0.74	0.89	0.78	0.82	0.85	0.77	0.54	0.82	0.59
60 ms	0.67	0.79	0.88	0.66	0.72	0.69	0.94	0.74	0.77	0.83	0.71	0.55	0.78	0.59
80 ms	0.62	0.73	0.87	0.63	0.75	0.67	0.95	0.74	0.74	0.80	0.69	0.56	0.82	0.59
100 ms	0.59	0.74	0.84	0.63	0.73	0.66	0.95	0.73	0.80	0.79	0.65	0.56	0.80	0.62
120 ms	0.59	0.68	0.82	0.60	0.82	0.60	0.96	0.74	0.82	0.78	0.63	0.58	0.82	0.56
140 ms	0.57	0.65	0.80	0.59	0.78	0.59	0.95	0.71	0.83	0.76	0.61	0.57	0.85	0.57
160 ms	0.56	0.64	0.79	0.58	0.76	0.60	0.95	0.69	0.83	0.75	0.59	0.58	0.72	0.55
180 ms	0.56	0.63	0.74	0.56	0.78	0.59	0.95	0.82	0.79	0.72	0.59	0.58	0.81	0.55
200 ms	0.54	0.61	0.70	0.55	0.73	0.58	0.95	0.90	0.78	0.74	0.57	0.59	0.77	0.54
220 ms	0.54	0.61	0.69	0.55	0.76	0.59	0.95	0.79	0.64	0.65	0.56	0.58	0.79	0.59
240 ms	0.56	0.55	0.70	0.56	0.71	0.58	0.94	0.82	0.68	0.65	0.55	0.58	0.8	0.59
260 ms	0.55	0.54	0.65	0.56	0.69	0.56	0.94	0.80	0.63	0.56	0.55	0.58	0.73	0.56

(b) RTT-fairness (100 $Mbps$).

Figure 6.1: Fairness between two flows sharing a bottleneck that (a) use different algorithms, but have the same RTT, and (b) use the same algorithm, but have different RTTs (the first column indicates the difference in RTTs). Red squares represent the intra- and RTT-fairness properties of the four groups. The fairness index ranges from 0.5 (worst) to 1 (best). The results were obtained using the Mininet emulation environment with a bandwidth limit of 100 $Mbps$.

in RTT (or eventually packet loss). Consequently, they prevent queue build-up, and, if no flows using algorithms belonging to other congestion control groups are present, the queue should remain nearly constant, which we use as the identifying pattern.

- **Loss-delay algorithms.** Some of the best-known algorithms from this group are TCP Compound (default algorithm for Windows Server until 2019 [223]) and TCP Illinois [198]. Since they incorporate delay measurements in the congestion window calculation, they are less aggressive than the purely loss-based group. However, they still mostly use loss as their primary metric and only reduce the sending rate upon detecting loss. Thus, queues are still filled (albeit at a slower pace). Consequently, we use the same pattern as for purely loss-based algorithms, *i.e.,* constant queue build-up, for this group of algorithms as well. To differentiate between these two groups, we, in addition, track how fast the queue is being filled.

- **Model-based hybrid algorithms.** Algorithms from this group try to build a model of the network, instead of using the standard AIMD (additive increase/multiplicative decrease) algorithm. The bottleneck bandwidth and round-trip time (BBR) algorithm [34], with its periodic pattern of increasing/decreasing the sending rate, is the best-known example of this group. However, unlike other protocols, it relies neither on packet loss or increase in RTT to detect congestion. Thus, to detect this algorithm, we instruct the switches to track the pattern of increasing/decreasing the sending rate.

Since model-based algorithms can employ very different methods to estimate the available resources in the network, further sub-groups with their distinct patterns could be formed. Moreover, since the aggressiveness of the loss-delay and purely-loss algorithms can vary, they also can be further subdivided.

6.2.2. RTT FAIRNESS

Grouping flows based solely on the metric used to detect congestion does not guarantee good fairness (Figure 6.1b). On the one hand, algorithms using the AIMD algorithm usually favor the flow with a lower RTT. This flow has a faster update loop and can, therefore, adjust its sending rate more often, claiming more resources. On the other hand, model-based algorithms, such as BBR, favor flows with higher RTTs, by allowing them more time to probe for resources and to dominate the queues in the process [210, 224, 225]. Consequently, when designing *P4air* we take these differences into account by choosing custom actions targeting each group's specifics.

6.3. P4AIR

To detect the patterns and features of different algorithm groups, we decided to make use of switches with programmable data planes. On the one hand, they offer the possibility to gather and export important packet meta-data (e.g., timestamps from different stages of processing, queue depth, etc.) directly from the data plane [99]. On the other hand, they support stateful processing, which enables the switches to track the way flows

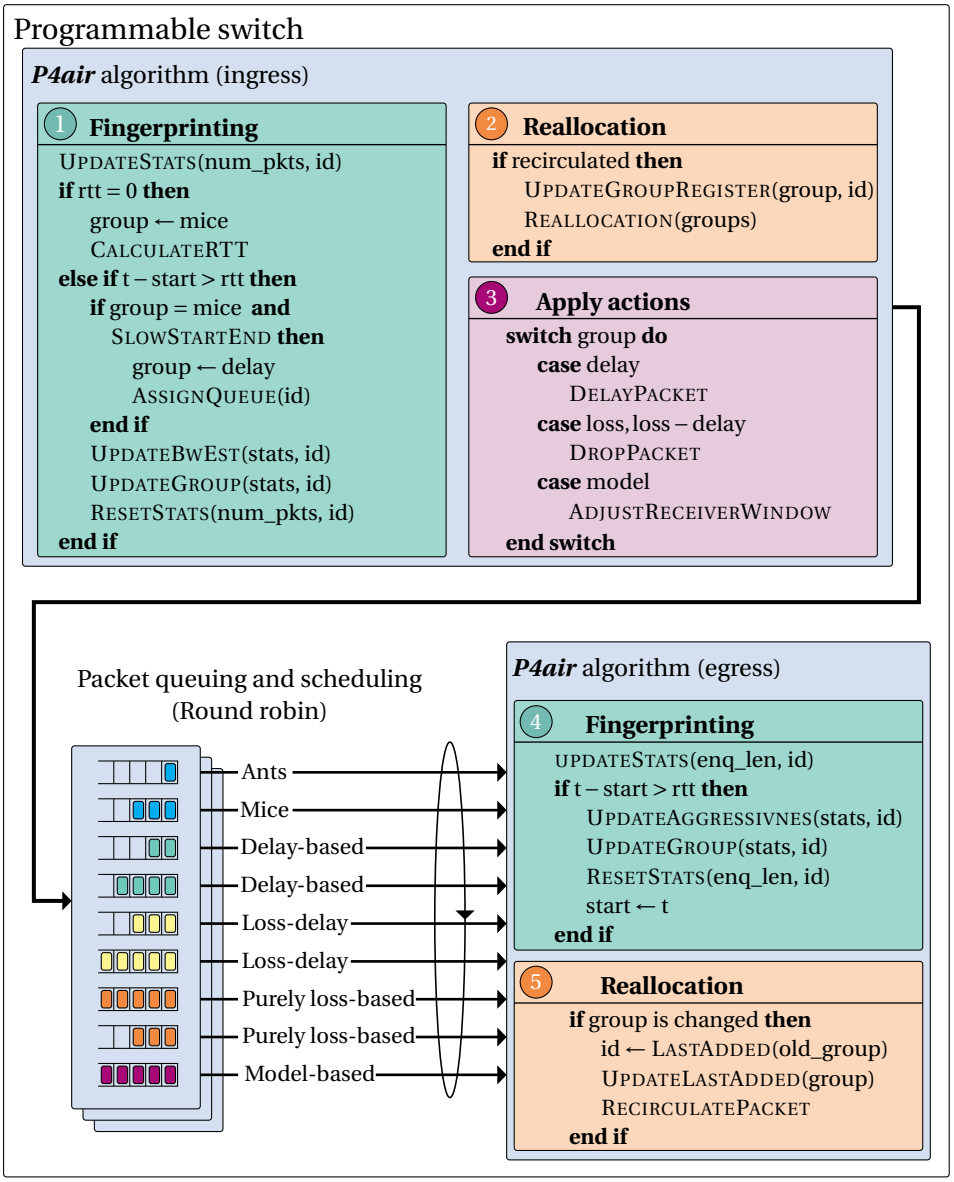


Figure 6.2: *P4air* algorithm. Every incoming packet is processed through three modules: (1) **Fingerprinting module** that determines the group (or sub-group) of the flow the packet belongs to (Section 6.3.1); (2) **Reallocation module** that processes the groups' updates, as well as the allocation of queues between groups (Section 6.3.2); and (3) **Apply actions module** that executes custom actions on packets to enforce fairness between flows being processed in the same queue (Section 6.3.3).

react to certain events, such as loss or an increase in queue size. By leveraging these two features, we have designed an algorithm, called *P4air*, that enforces fairness among

competing congestion control algorithms.

P4air consists of three modules split between the ingress and egress blocks (Figure 6.2): (1) **Fingerprinting module** that groups flows based on their congestion control algorithm (Section 6.3.1); (2) **Reallocation module** that, when necessary, redistributes the queues between groups (Section 6.3.2); and (3) **Apply actions module** that enforces fairness among flows processed in the same queue by applying custom actions (Section 6.3.3).

6.3.1. FINGERPRINTING MODULE

The fingerprinting algorithm tracks the way the flows react to certain events, distinguishing between short-lived and long-lived flows, as well as different groups of congestion control algorithms (as discussed in Section 6.2) used by long-lived flows.

Short-lived flows. Whenever *P4air* receives a packet belonging to a new flow, it classifies it as a short-lived flow (Figure 6.2). It distinguishes between two groups of short-lived flows: (1) ant flows, i.e., short, sparse flows that only transport a few, but usually critical, packets (e.g., ARP, DNS, DHCP) and (2) mice flows, i.e., TCP/QUIC flows still in the slow-start phase.

End of the slow-start phase. Mice flows typically transport only a small amount of data and, consequently, do not send enough to congest the switch on their own. However, if a long-lived flow would reside in this queue, it could significantly degrade their performance, causing unnecessary delays or even dropped packets. Thus, *P4air* implements a mechanism to detect the end of the slow-start phase, thereby distinguishing between long- and short-lived flows.

To do so, *P4air* first uses the timestamps of packets involved in the 3-way handshake, to estimate the flow's RTT. To be precise, it subtracts the timestamp of the first SYN from the first packet sent after this SYN. Next, it estimates the bandwidth-delay product (BDP) as the product between the flow's "fair" share – namely the output rate divided by the number of long-lived connections – and the estimated RTT of the flow. This value describes the number of packets (bytes) that should be sent per RTT for the TCP connection to fully utilize its share without filling the queues. Finally, *P4air* will reclassify a flow into one of the long-lived groups upon detecting either of the two following patterns: (1) a decrease in the number of processed packets (bytes) per RTT or (2) the number of processed packets (bytes) reaching the BDP within an RTT interval. Additionally, upon detecting the second pattern, *P4air* proactively drops a packet, forcing the flow to enter the congestion-avoidance phase. This way, the very aggressive slow-start phase is reduced to only the time needed to reach the bandwidth share the flow should ideally claim, avoiding the queue buildup for the mice queue.

Long-lived flows. Upon reclassifying a flow as a long-lived flow, *P4air* continuously executes two actions: (1) tracking of flow statistics, used to determine the group of the congestion control algorithm; and (2) recalculating the group, upon detecting specific patterns.

Tracking of flow statistics. For each processed packet *P4air* updates the following two statistics: (1) the number of processed packets (or bytes), and (2) the depth of the queue at the moment before the packet is placed in (enqueue queue length). In addition, as

most congestion control algorithms change their behavior each RTT to react to events (or lack of them), we decided to aggregate these statistics per RTT. Furthermore, due to constraints of P4 programs (imposed to make sure that switches will run at line-rate), in particular, the lack of division and floating-point operations needed to track average values, we decided to store their maximum values.

Moreover, based on them, *P4air* tracks two additional metrics, called aggressiveness and BwEst counter. Aggressiveness tracks how fast a queue is being filled, differentiating between delay-, loss-delay, and purely loss-based algorithms. Every time the maximum enqueue length increases by 1%, aggressiveness is increased by 1. Otherwise, the aggressiveness is reset to 0. The BwEst counter tracks the number of patterns of increasing sending rate (by a factor of at least 1.125), which is typically used while probing for more bandwidth.

As illustrated in Figure 6.2, the fingerprinting module is split between the ingress and egress blocks. To account for all packets belonging to a flow, including the ones dropped due to congestion, we decided to track the number of packets, as well as BwEst (calculated using (BwEst), in the ingress block. Similarly, since queuing statistics are not available in the ingress block (as the packet was not yet processed in the queue), the enqueue length, as well as aggressiveness, are tracked in the egress block.

Recalculating the group. We decided to, initially, classify all long-lived flows into the most conservative group (delay-based). Only upon detecting a more aggressive behavior, we reclassify them into more aggressive groups: at first loss-delay and, finally, the purely loss-based group. This way, for delay-based flows, a queue build-up is avoided, preventing them from sharing the queue with more aggressive flows (and triggering their back-off mechanism in the process). To do so, at the end of each RTT interval, *P4air* uses the flow's statistics (as well as the statistics from the previous RTT interval) to tracks the following patterns:

- A continuous increase in the maximum enqueue depth for at least m_{LD} RTT intervals, without any reduction in the sending rate, causes the newest flow assigned to the delay-based group to be reclassified as loss-delay.

- A continuous increase in the maximum depth of the queue for m_{PL} ($m_{PL} > m_{LD}$) subsequent RTT intervals, causes the newest flow assigned to the loss-delay group to be reclassified as purely loss-based. This way, algorithms that use delay as their secondary metric (loss-delay) are differentiated from purely loss-based algorithms.

- A periodic pattern of increasing/decreasing the sending rate (tracked using the BwEst metric and parameter m_M), causes the flow to be classified as model-based (BBR), exploiting the fact that in each probe (drain) bandwidth phase, a BBR flow deliberately increases (decreases) the sending rate by 1.25 (0.75) times the measured bandwidth-delay product.

Parameters m_{LD}, m_{PL}, and m_M, are configurable and define the sensitivity of the Fingerprinting module. By lowering these values, we decrease the time needed to detect each group. However, the probability of misclassification might increase, especially for

the more conservative groups. By increasing these values, more aggressive classes (e.g., the loss-based group) might never be detected, which reduces the accuracy.

Figure 6.3 illustrates the fingerprinting process for the representatives of the four groups: Cubic for purely loss-based, Illinois for loss-delay, Vegas for delay-based, and BBR for model-based algorithms. First, *P4air* classifies all four flows as mice flows. After they reach their BDP, *P4air* drops a packet, forcing all four flows to enter the congestion-avoidance phase, and classifies them into the delay-based group. However, Cubic, Illinois, and BBR start filling the queues, without backing off and are reclassified into the loss-delay group. Next, due to Cubic's very aggressive approach, the queues and the aggressiveness continue to increase, causing *P4air* to classify it into its correct group: purely loss-based. Similarly, after *P4air* detects the periodic increase in BBR's sending rate, it reclassifies it into the model-based group. In this scenario, this occurs after this pattern is recognized twice.

Location vs. accuracy. As Figure 6.3 illustrates, to correctly detect the more aggressive flows, *P4air* needs to be deployed on the bottleneck switch, i.e. a switch at which the queues are formed. Otherwise, due to no increase in the queuing delay, the Fingerprinting module would classify these algorithms as delay-based algorithms. However, when loss and loss-delay algorithms are not filling the queues, they would also not interfere with the present delay-based algorithms. In other words, if there would not be a bottleneck, there would also not be a problem that *P4air* needs to solve. In contrast, algorithms like BBR (that do not rely on the queuing metrics) can always be detected due to their periodic pattern.

6.3.2. REALLOCATION MODULE

Every time the Fingerprinting module reclassifies a flow, the **Reallocation module** executes two actions: (1) it stores and updates the flow's group, and (2) it runs the queue reallocation algorithm, making sure that long-lived flows are distributed evenly across all available queues.

Updating and storing of the group. For most flows, group recalculation can only be done in the egress block, i.e., after the egress statistics (e.g., aggressiveness, enqueue depth) are known. However, to ensure that the packet is queued correctly, the flow's group needs to be known in the ingress block. Consequently, the register (stateful memory array) storing the estimated group must be allocated in the ingress block and is, as such, not accessible from the egress block. To solve the abovementioned problem, *P4air* recirculates all packets that trigger a reclassification (Figure 6.2).

Queue reallocation algorithm. To leverage standard scheduling mechanisms, such as Round Robin (RR) or FQ (Fair Queuing), we designed a queue reallocation algorithm. Moreover, to be able to deploy this algorithm on the hardware switches running at line-rate, we avoided operations that would introduce significant computational overhead (e.g., loops and floating-point operations). Our algorithm makes sure that groups that have more flows are also assigned more queues, thereby distributing all the flows evenly across all the available queues. First, to ensure that packets belonging to the same flow are processed in the same queue, *P4air* uses an additional register array to store the queue information. Furthermore, for every recirculated packet, *P4air* updates this register to make sure that flows belonging to the same group are processed together, by

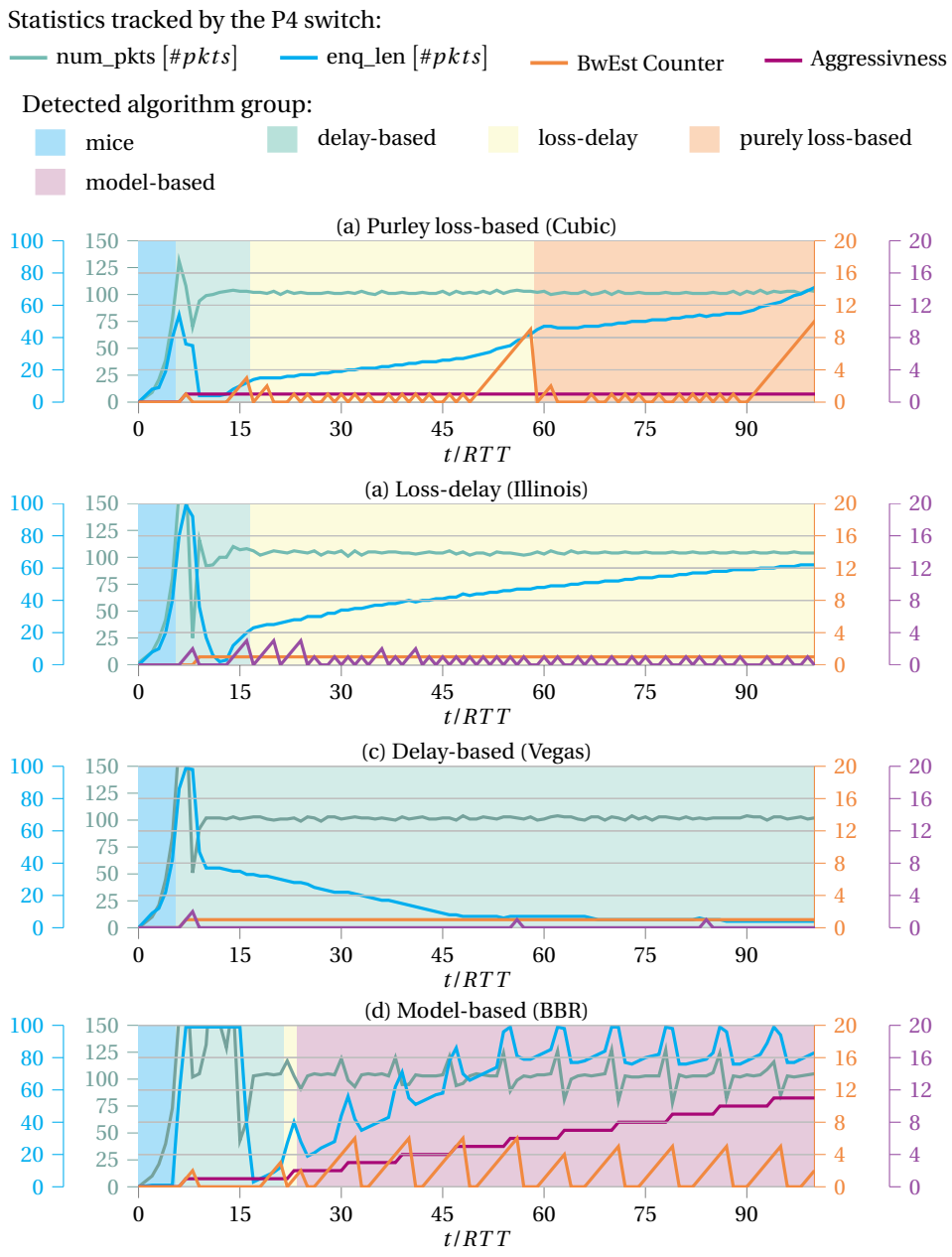


Figure 6.3: Fingerprinting module. Illustration of the *P4air* data plane fingerprinting algorithm for the representatives of the four groups of congestion control algorithms. The lines represent flow statistics and different background colors represent the outcome of the fingerprinting algorithm as measured by the switch. The following configuration was used: the maximum queue size was 100 packets, the output rate 1000 pps, RTT 100 ms, $m_{LD} = 4$, $m_{PL} = 10$, $m_M = 2$, $n_{flows} = 1$.

assigning a new queue using a sequential index (one per-group). As the next step, it updates the boundaries of each group (parameters l_i) according to Figure 6.4.

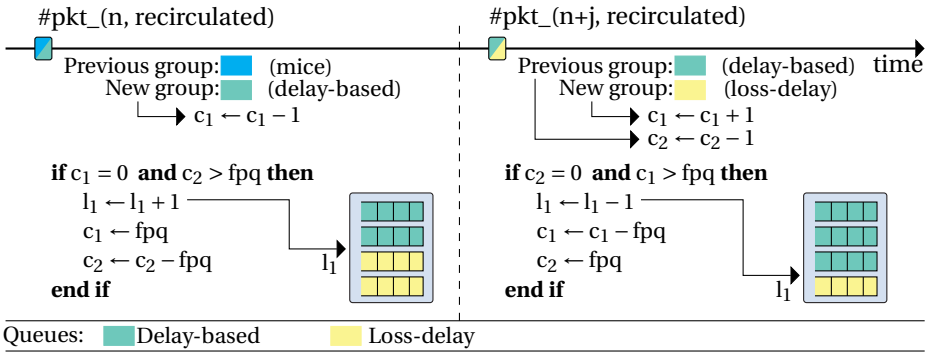


Figure 6.4: Simplified representation of the reallocation algorithm's mechanism for recalculating l_i (boundaries between groups) for two groups. Every time a packet is recirculated, a set of counters c_i, which track the number of flows that need to be added to the group i for l_1 to change, is either incremented (counter belonging to the previous group) or decremented (counter belonging to the new group). If any of the counters reaches zero, while the other one is higher than fpq (flows per queue), l_1 is recalculated. Every time fpq is increased, values c_i are increased by the number of queues assigned per class (2 in this example).

For every other packet (not recirculated), *P4air* checks if the stored queue is outside of the corresponding l_i values (that might have been updated) and, if so, assigns a new queue using the sequential index (Figure 6.5). This way, whenever reallocation occurs, *P4air* reassigns all flows processed in the queue that was assigned to a different group uniformly among the other remaining queues belonging to the group. Similarly, it assigns the first $fpq-1$ flows (and the flow that triggered the l_i update) belonging to the group that gained a queue to the new queue.

6.3.3. APPLY ACTIONS MODULE

Traditional AQMs can only probabilistically drop packets or use ECN marking to trigger the senders to back off. However, as mentioned earlier, for some congestion control algorithms, less severe actions, such as delaying a packet, might be more appropriate. Furthermore, newer congestion control algorithms might not react to these indicators and require new actions to be designed. Hence, to target each group's specifics, *P4air* uses the custom actions listed below.

Dropping a packet. This action targets the algorithms that use loss as the primary metric (purely loss-based and loss-delay algorithms), similar to standard AQMs.

Delaying a packet. This action targets delay-based algorithms. By delaying (instead of dropping) a packet, they will back off, reducing the need for retransmissions and improving the average end-to-end delay in the process. Just delaying a packet is not possible in P4, thus, we implemented this action by recirculating a packet back to the ingress.

Changing the receiver window. A flow's sending rate is determined by the minimum of the receiver and congestion windows. Thus, by reducing the receiver window, the sender is forced to back off. However, to do so, the window in the ACK packets, sent from

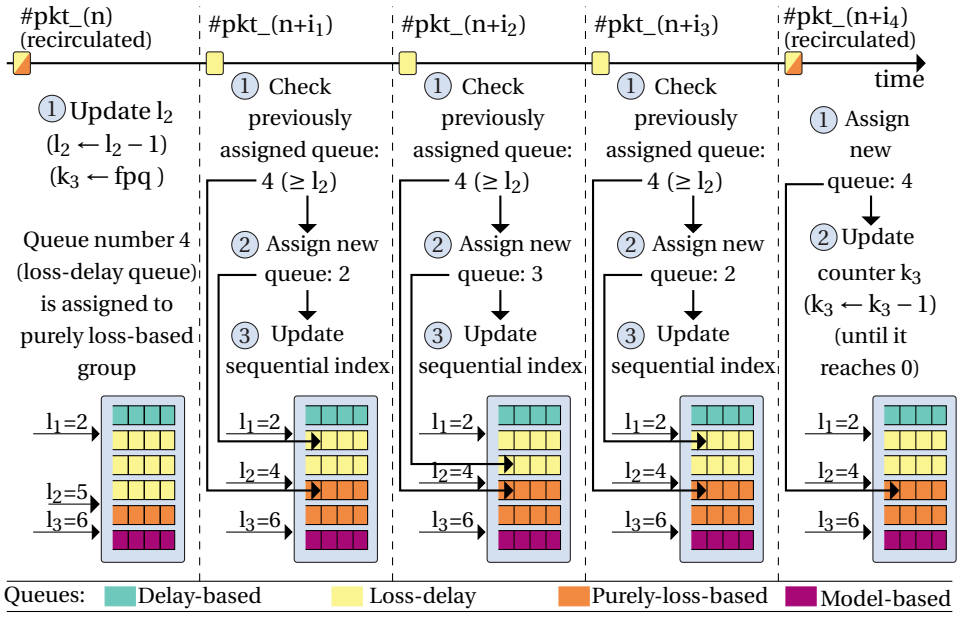


Figure 6.5: Reassigning flows. Packet n, a recirculated loss-delay packet that is reclassified to the purely-loss group, updates l_2 (the boundary between loss-delay and loss algorithm). After loss-delay packets ($n+i_1$, $n+i_2$, $n+i_3$, shown in yellow) belonging to flows that were previously processed in queue 4 are received, their queue is reassigned using a sequential index. Similarly, new purely-loss based flows (packet $n+i_4$) are assigned to queue 4.

receiver to sender, needs to be modified. For this action to work, packet transfers in both directions have to cross the same bottleneck.

Sensitivity. The sensitivity of the Apply actions module, i.e., how often *P4air* applies actions to the flows, determines the link utilization, as well as the distribution of resources among flows. If the sensitivity is set too high, back-off mechanisms are triggered before the flows even started congesting the network, leading to low utilization. In contrast, if configured too low, they are triggered too late (or at all), allowing aggressive flows to claim more resources and leading to unfairness. Consequently, to keep the utilization high, while still targeting aggressive flows, we decided to execute this module only when the flow the packet belongs to is sending above its BDP.

6.3.4. OVERHEAD & LIMITATIONS.

Memory overhead. Contrary to standard AQM solutions, such as Codel, the memory overhead of *P4air* scales linearly on both the number of flows it wishes to track (127b per flow), as well as the number of output ports ($(5+n_{queues})\log_2(n_{flows})+3\log_2(n_{queues})+8\log_2(n_{flows}/n_{queues})$ bits per output port). However, as Table 6.1 illustrates, memory is mostly consumed by the Fingerprinting module to track the current RTT interval and current flow statistics (parameter α). Since flows from the loss- and model-based groups

Table 6.1: Memory consumption on a 24-port switch. β is the memory used to track the current group and queue and α the memory used to track the RTT and flow statistics.

n_{flows}	Total [kB]	Fingerprinting α [%]	Fingerprinting β [%]	Reallocation [%]
2^{10}	17.53	86.89	5.84	7.27
2^{11}	33.92	89.81	6.04	4.16
2^{12}	66.57	91.53	6.15	2.32
2^{13}	131.73	92.51	6.22	1.28

should not be reclassified (except if m_M and m_{PL} are misconfigured, see Figure 6.7b), memory consumption can be significantly reduced by only keeping track of their group and queue (parameter β) and not their RTT and flow statistics (parameter α).

Recirculations. Recirculated packets compete for resources with incoming packets, causing potential drops in throughput. However, while updating the group, their amount per flow is at most 4 (maximum number of re-classifications per-flow). Furthermore, due to the lack of other ways to delay a packet, recirculations are used in the Apply actions module to target the delay-based algorithms. However, due to the very conservative nature of these algorithms, we did not experience any noticeable negative effect on the switch's performance.

Collisions. As one of the main building blocks of *P4air*, we use hash tables, since they are supported on all programmable hardware. To generate an index to access them, *P4air* calculates a hash based on the flow identifier (5-tuple consisting of source and destination IP, layer 4 protocol, source and destination ports). However, when the number of concurrent unique flows increases, so does the probability of hash collisions. When two flows collide, *P4air* will see them as one, potentially misclassifying them and reducing fairness.

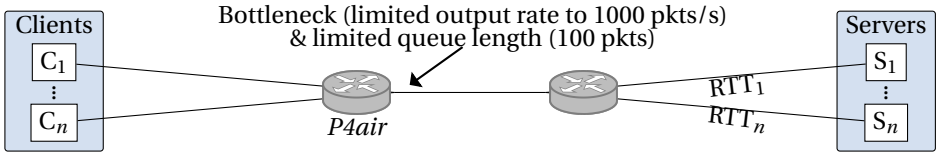
Packet reordering. During reallocation, flows might be processed by two queues at the same time, potentially leading to packet reordering. However, we did not experience any related noticeable issues in our experiments.

BDP calculation. Due to the lack of support for floating-point operations on hardware switches, the sensitivity of the Apply actions module must be approximated using the estimated RTT, i.e. $RTT_{Est} >> s$, where s is calculated to be close to the BDP of the flow, e.g. $\lceil log2(num\_flows) + log2(packet\_length \cdot Throughput) \rceil$. Consequently, to keep the utilization high, we decided to slightly postpone the actions, allowing flows to partially fill the queues.

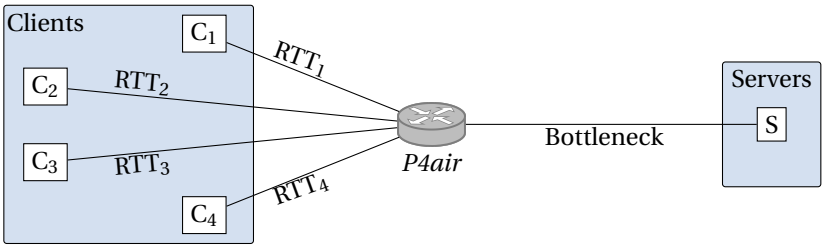
6.4. EVALUATION

6.4.1. EXPERIMENT SETUP

Performance metrics. To evaluate *P4air*, we used the following metrics: (1) detection delay as the number of RTT intervals needed to recognize the correct congestion control group; (2) detection accuracy as the percentage of correctly classified flows; (3) utilization as the percentage of the total available bandwidth used by all the connections, (4)



(a) Simulation topology (using the bmv2 switch and Mininet).



(b) Experiment topology (using the Tofino switch [98]). All links are 10 *Gbps*.

Figure 6.6: Topologies.

RTT increase (due to queuing), (5) the fraction of throughput each flow received, and (6) fairness index.

Comparison baselines. We compared our solution against (1) a simple switch without an algorithm to improve fairness (No AQM) and (2) an algorithm that provides flow separation based on the hash of the 5-tuple (as commonly used in vendor implementations [226]) by enqueuing packets into different queues (Different Queues). Furthermore, we tested two versions of *P4air*: (1) Idle *P4air* (Fingerprinting + Reallocation) and (2) *P4air* (Fingerprinting + Reallocation + Apply actions).

Topology. We have used the topologies shown in Figure 6.6. Given that the performance of congestion control algorithms is affected by the bottleneck link on the path, such simple topologies suffice for our purposes. We performed the experiments using (1) the Mininet emulation environment with the P4 software switch (bmv2[141], Figure 6.6a) and (2) a testbed with a Intel Tofino switch [98] (Figure 6.6b). To perform measurements, we relied on tcpdump, iperf3, socket statistics, and P4 statistics exported directly from the switch.

6.4.2. TUNING OF THE FINGERPRINTING ALGORITHM

P4air has multiple tunable parameters (m_{LD}, m_{PL}, and m_M) that provide a trade-off between detection accuracy and detection delay. Figures 6.7a and 6.8c show the impact of changing the m_{LD}, m_{LD}, and m_{LD} on the detection of each group.

Choosing $m_{LD} = 4$. As all flows are, by default, assigned into the delay-based group, the choice of m_{LD} should ensure that only delay-based flows remain, while all the others are reclassified into the loss-delay group. By analyzing the detection time and accuracy for different values of m_{LD} (Figure 6.7a - 6.8a), we find that for $m_{LD} = 4$, the probability of false positives for the delay-based algorithms is low enough.

Choosing $m_{PL} = 12$. By increasing m_{PL}, the probability of misclassifying a loss-delay algorithm decreases, even for the aggressive algorithms from this group (e.g., Illinois, Figure 6.7b). However, in addition to the increase in detection delay (Figure 6.8b), the accuracy of the fingerprinting module for the least aggressive purely loss-based algorithms, such as New Reno, decreases as well (Figure 6.7b). Thus, we choose $m_{PL} = 12$, as the best compromise between accuracy and detection delay.

Choosing $m_M = 4$. Finally, we evaluated the influence of m_M on the detection accuracy of the model-based algorithms. All algorithms probe for bandwidth in their slow-start phase and, depending on their classification into the delay-based group (when the tracking for aggressiveness and BWEst starts), they can cross the threshold m_M. However, only BBR does so periodically, every 10 seconds, and will always be correctly identified if m_M is set high enough.

6.4.3. P4AIR PERFORMANCE

Resource utilization. Our Tofino implementation, when tracking a maximum of 2^{16} flows, used less than 14% of the available header and metadata memory, less than 18% of the total register memory and less than 5% of hash generators available on the switch (shared between forwarding and *P4air*).

The RTT-estimation algorithm. First, we evaluated the accuracy of the RTT estimation by varying the link delay and external traffic. In all the scenarios without external traffic, the difference between the configured and estimated RTT was less than $0.52ms$ ($\leq$ 1.5% of RTT_{conf}, Figure 6.9a). As this value was nearly constant in our experiments, we conclude that this overhead is the processing delay on both the servers and the switch. Furthermore, as Figure 6.9b illustrates, by processing the new flows in a separate queue (as in *P4air*), the effect of long-lived connections on the RTT accuracy is not significant.

Sensitivity. If the sensitivity of the Apply actions module (s) is set too high, all flows are punished too aggressively and the overall utilization drops significantly, reaching as little as 50%. In contrast, if s is set too low, aggressive flows are never punished and the fairness will remain low (Figure 6.9c).

Different actions. The action to change the receiver window offered the biggest performance boost (Figuress 6.10b, Figure 6.12b).

6.4.4. INTER- AND INTRA-FAIRNESS: P4AIR VS. EXISTING SOLUTIONS

Effect of fingerprinting. Distributing flows to queues based on their congestion control group significantly improves fairness (Figure 6.10a), especially when the number of flows increases and their interactions become more detrimental. As Figure 6.11b illustrates, *P4air* (and Idle *P4air*) leverage the good intra-fairness properties of most algorithms and flows, consequently, rarely overpower each other, i.e., their throughput is clustered around the ideal throughput. In contrast, by queuing flows without taking into account their group (Different Queues), two distinct clusters are present: (1) overpowered flows at 6 – 7% of the ideal throughput and (2) aggressive flows at multiples of the ideal throughput.

Effect of the Apply actions module. Fingerprinting flows improves fairness, but does not prevent queue buildup. However, the Apply actions module targets the aggressive flows,

m_{LD}	HS-TCP	STCP	HTCP	BIC	Cubic	New Reno	Hybla	YeAH	Illinois	Veno	Westwood+	BBR	Vegas	LoLa
1	100	100	100	100	100	100	100	100	100	100	100	100	100	100
2	100	100	100	100	100	100	100	100	100	100	100	100	100	35
3	100	99	100	100	100	100	99	99	100	100	100	97	8	1
4	100	100	100	100	100	100	99	99	100	100	100	95	0	0
5	100	100	100	100	100	100	100	100	100	100	100	98	0	0
6	100	100	100	100	100	100	100	97	100	70	100	99	0	0

(a) Percentage of flows classified as belonging to the loss-delay group depending on m_{LD}.

m_{PL}	HS-TCP	STCP	HTCP	BIC	Cubic	New Reno	Hybla	YeAH	Illinois	Veno	Westwood+	BBR	Vegas	LoLa
6	100	100	100	100	100	100	100	93	96	33	100	100	0	0
8	100	100	100	100	100	90	100	55	60	0	89	33	0	0
10	100	100	100	100	100	87	89	36	38	0	34	0	0	0
12	99	100	100	80	100	30	100	0	0	0	0	0	0	0
14	100	100	100	37	100	0	100	0	0	0	0	0	0	0
16	100	100	100	0	100	0	100	0	0	0	0	0	0	0
18	100	92	100	0	100	0	100	0	0	0	0	0	0	0

(b) Percentage of flows classified as belonging to the purely loss-based group depending on m_{PL}.

m_M	HS-TCP	STCP	HTCP	BIC	Cubic	New Reno	Hybla	YeAH	Illinois	Veno	Westwood+	BBR	Vegas	LoLa
1	19	47	39	77	17	14	62	49	22	15	14	100	18	19
2	21	24	8	9	2	0	37	21	2	0	0	100	0	0
3	0	11	1	0	0	0	9	19	1	0	0	100	0	0
4	0	0	0	0	0	0	0	0	0	0	0	100	0	0
5	0	0	0	0	0	0	0	0	0	0	0	100	0	0
6	0	0	0	0	0	0	0	0	0	0	0	100	0	0

(c) Percentage of flows classified as belonging to the model-based group depending on m_M.

Figure 6.7: Tuning of the fingerprinting module (bmv2 switch). The algorithms that should be classified as such are shown in blue. Misclassified algorithms are shown in red. Algorithms that use different metrics (model-based group) are shown in yellow. Each scenario is run 10 times for 10 different RTTs ranging from 50ms to 150ms with a step of 10ms (100 in total).

forcing them to back off and, consequently, lowers the increase in RTT due to queuing (Figures 6.11a). Furthermore, when the number of flows per queue increases, the Apply actions module makes sure that they remain fair to each other (Figure 6.10b and 6.10a).

Idle *P4air* vs. *P4air*. When a small number of flows compete per queue ($n_{flows} < 128$), the Fingerprinting and the Reallocation modules are enough to achieve good fairness properties (Figure 6.10a). However, to reduce the queuing delay and to target a higher number of flows, the Apply actions module is needed.

Utilization. As Figure 6.11a illustrates, both versions of *P4air* were able to maintain a

m_{LD}	HS-TCP	STCP	HTCP	BIC	Cubic	New Reno	Hybla	YeAH	Illinois	Veno	Westwood+	BBR	Vegas	LoLa
1	5.8	5.8	5.8	6.0	5.8	6.0	4.7	5.8	5.8	5.8	5.9	5.8	5.7	9.9
2	7.2	7.2	7.2	7.2	19.0	7.2	9.0	7.2	7.2	7.3	7.2	7.3	7.2	62.1
3	14.2	20.5	16.9	16.7	16.6	18.4	15.8	36.1	182.7	128.1	121.8	19.7	7.8	21.0
4	15.5	23.3	19.2	19.5	17.3	38.7	17.8	37.7	192.5	135.3	123.9	22.3	-	-
5	18.5	26.8	21.6	23.1	19.6	63.8	18.9	40.6	189.7	133.2	126.0	24.2	-	-
6	20.4	28.9	23.4	44.5	19.6	96.3	19.4	61.5	199.0	134.7	133.8	25.9	-	-

(a) Average RTT interval in which the flow was classified as belonging to the loss-delay group depending on m_{LD}.

m_{PL}	HS-TCP	STCP	HTCP	BIC	Cubic	New Reno	Hybla	YeAH	Illinois	Veno	Westwood+	BBR	Vegas	LoLa
6	29.7	42.4	27.8	26.8	51.5	159.4	21.2	197.4	324.4	265.1	136.9	32.0	-	-
8	35.4	46.2	30.3	69.1	51.9	219.8	24.4	224.2	386.7	-	264.3	245.8	-	-
10	41.1	56.4	33.3	168.8	55.5	261.5	56.1	351.0	377.4	-	300.4	-	-	-
12	60.3	75.4	52.0	257.9	60.5	296.7	25.3	-	-	-	-	-	-	-
14	163.1	105.3	59.5	329.3	63.5	0	41.6	-	-	-	-	-	-	-
16	163.9	149.7	66.4	0	64.9	0	49.7	-	-	-	-	-	-	-
18	250.9	189.7	122.2	0	71.9	0	49.9	-	-	-	-	-	-	-

(b) Average RTT interval in which the flow was classified as belonging to the purely loss-based group depending on m_{PL}.

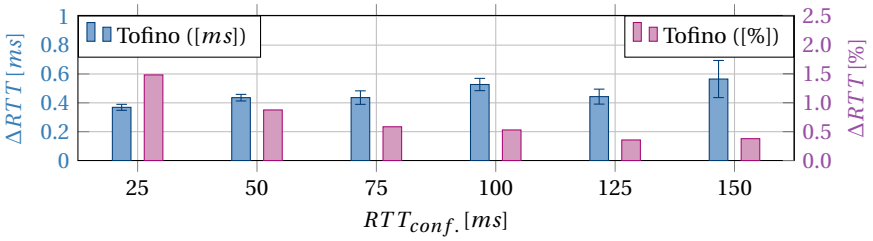
m_M	HS-TCP	STCP	HTCP	BIC	Cubic	New Reno	Hybla	YeAH	Illinois	Veno	Westwood+	BBR	Vegas	LoLa
1	22.3	11.4	25.2	9.5	19.9	10.0	11.9	11.2	74.3	10.4	10.2	18.1	10.2	6.7
2	51.3	15.4	31.8	106.8	29.0	-	30.4	13.2	244.5	-	-	24.4	-	-
3	-	17.0	-	28.0	-	-	17.4	13.1	556.0	-	-	40.1	-	-
4	-	-	-	-	-	-	-	-	-	-	-	47.7	-	-
5	-	-	-	-	-	-	-	-	-	-	-	58.2	-	-
6	-	-	-	-	-	-	-	-	-	-	-	71.8	-	-

(c) Average RTT interval in which the flow was classified as belonging to the model-based group depending on m_M.

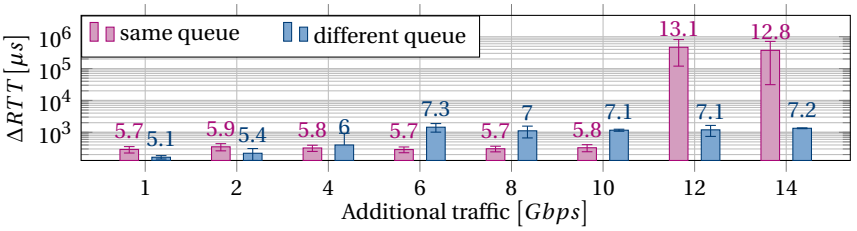
Figure 6.8: Classification speed of the fingerprinting module (bmv2 switch). Each scenario is run 10 times for 10 different RTTs ranging from 50ms to 150ms with a step of 10ms (100 in total).

high link utilization (≥ 91%), similarly to the results achieved by the comparison baselines (Different Queues and NoAQM).

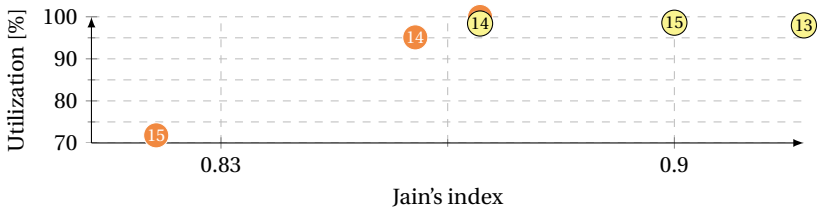
Inter-Fairness. Both *P4air* versions were able to significantly outperform the comparison baselines and realize a fair distribution of resources (Figure 6.10a). Therefore, our results show that by using more information about the flow, such as the group of the congestion control algorithm, the switch can target the flow's specifics, thereby enabling a fairer distribution of resources.



(a) Average error in the RTT estimation for different RTTs. Each scenario is run 10 times (Confidence intervals 90%).



(b) Average error in the RTT estimation for different levels of congestion. Each scenario is run 10 times (Confidence intervals 90%).



(c) Sensitivity. Cubic is shown in red, Illinois in orange, and the value *s* inside the circles. Each scenario is run 4 times.

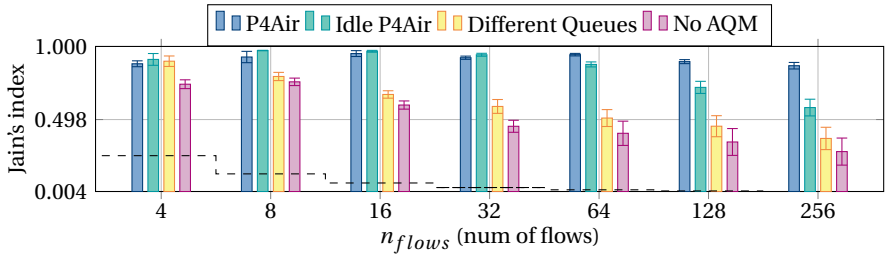
Figure 6.9: Evaluation of the *P4air* algorithm on a Intel Tofino switch.

6.4.5. RTT FAIRNESS: P4AIR VS. EXISTING SOLUTIONS

Effect of fingerprinting. While distributing flows to different queues, Idle *P4air* does not take into account the flows' RTT (but only the group), causing the flows with different RTTs to compete inside the same queue. However, in comparison to Different Queues, the Reallocation module makes sure that flows are more uniformly distributed between the queues. Consequently, the fairness index is higher (Figures 6.12a and 6.12d).

Effect of the Apply actions module. To increase the fairness, the Apply actions module is needed, especially when the RTT differences between the flows increase. As Figure 6.12d illustrates, *P4air* merges the two groups, the very aggressive flows at the right side and the overpowered flows at the left side, into one.

Utilization. As a consequence of the Apply actions module, i.e. the actions applied to the most aggressive flows, *P4air* had the lowest utilization compared to all the other so-



(a) Average Inter- and intra- fairness. The share of flows per group was varied between 0% and 100% with a step of 25%. All flows had the same RTT. For each n_{flows}, each combination (35 different) is run 4 times (140 in total). Theoretical minimum is shown as a dashed black line.

Cubic [%]	100	75	75	75	50	50	50	50	50	50	25	25	25	25	25	25	25	25
Illinois [%]	0	25	0	0	50	0	0	25	25	0	0	0	75	50	0	25	50	25
BBR [%]	0	0	25	0	0	50	0	25	0	25	0	75	0	25	50	0	0	50
Vegas [%]	0	0	0	25	0	0	50	0	25	25	75	0	0	0	25	50	25	0
P4Air	0.92	0.89	0.93	0.92	0.89	0.94	0.92	0.90	0.89	0.91	0.89	0.96	0.88	0.86	0.92	0.89	0.87	0.90
Idle P4Air	0.91	0.75	0.82	0.93	0.61	0.92	0.76	0.78	0.75	0.69	0.55	0.68	0.95	0.63	0.58	0.76	0.64	0.78
Different Queues	0.85	0.42	0.23	0.89	0.40	0.83	0.38	0.22	0.43	0.28	0.43	0.55	0.87	0.40	0.26	0.38	0.43	0.22
No AQM	0.94	0.29	0.19	0.94	0.32	0.95	0.20	0.18	0.43	0.18	0.34	0.22	0.95	0.29	0.17	0.17	0.19	0.15

Cubic [%]	25	25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Illinois [%]	0	25	100	0	0	75	75	0	25	25	0	50	50	0	50	25	25
BBR [%]	25	25	0	100	0	0	25	75	75	0	25	50	0	50	25	50	25
Vegas [%]	50	25	0	0	100	25	0	25	0	75	75	0	50	50	25	25	50
P4Air	0.89	0.88	0.87	1.00	0.94	0.84	0.87	0.96	0.93	0.86	0.88	0.86	0.88	0.77	0.88	0.89	0.87
Idle P4Air	0.75	0.69	0.48	0.69	0.94	0.55	0.53	0.71	0.59	0.76	0.81	0.76	0.57	0.64	0.60	0.70	0.62
Different Queues	0.49	0.24	0.49	0.74	0.91	0.46	0.26	0.59	0.58	0.38	0.25	0.36	0.41	0.39	0.24	0.21	0.36
No AQM	0.60	0.18	0.39	0.16	0.96	0.32	0.17	0.18	0.22	0.39	0.18	0.19	0.17	0.31	0.19	0.15	0.18

(b) Average inter- and intra- fairness for 128 flows (zoomed-in version of one of the scenarios shown in Figure 6.10a). The share of flows per group is varied between 0% and 100% with a step of 25% (shown at the top). Each ratio is run 4 times (Confidence intervals 90%).

Figure 6.10: Inter- and intra-Fairness. Comparison of the *P4air* algorithm to the comparison baselines on a Intel Tofino switch.

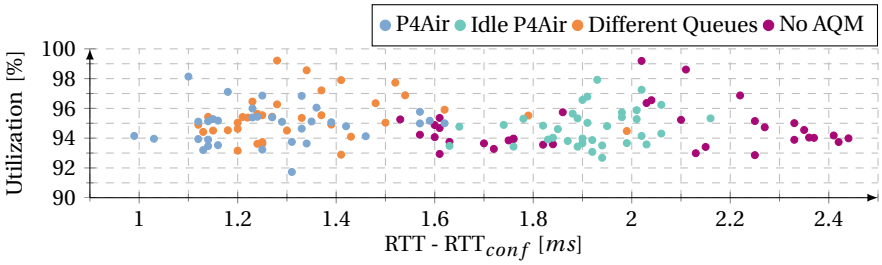
lutions (although still ≥ 90%).

Large ΔRTT. *P4air* was able to maintain a high fairness index, especially for lower ΔRTT values. However, as we increased ΔRTT, the fairness index reduced, although it was still higher than the comparison baselines.

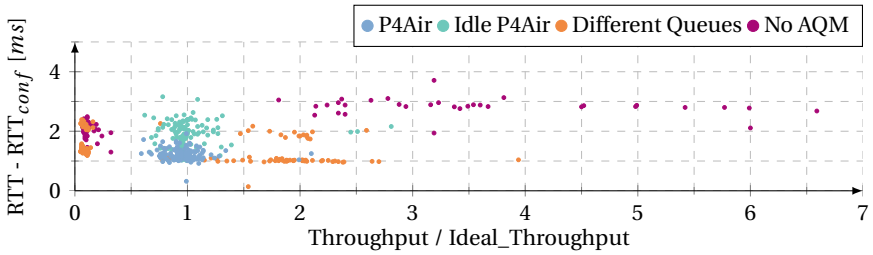
6.5. DEPLOYMENT CONSIDERATIONS

While our evaluation demonstrates performance gains, especially in terms of fairness, a more extensive evaluation in more complex scenarios (involving multiple switches) is recommended. Furthermore, several limitations of the current implementation, listed below, should be considered.

Weighted queuing algorithms. If an algorithm that supports dynamic weights per queue is supported by the switch, flows belonging to the same group can be assigned into one queue with a weight set to the number of flows present, ensuring that all groups get their fair share of resources, simplifying *P4air* by making the queue reallocation algorithm



(a) Average delay vs. average utilization for scenarios shown in Figure 6.10b. Ideal operating point (0, 100%).



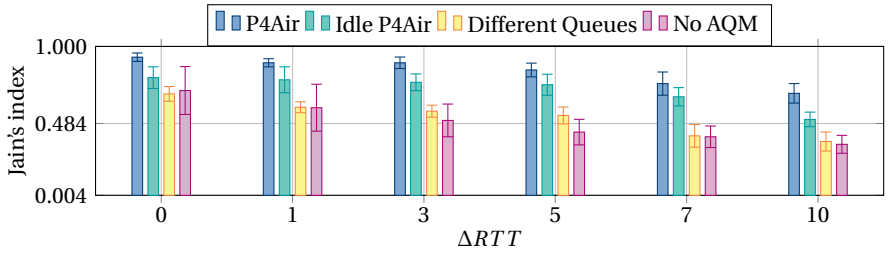
(b) Average RTT vs. throughput per flow for a scenario with 128 flows, with each group having 25% of the flows (zoomed-in version of one of the scenarios shown in Figure 6.10b). Ideal operating point is (1, 0).

Figure 6.11: Inter- and intra-Fairness. Comparison of the *P4air* algorithm to the comparison baselines on a Intel Tofino switch.

obsolete.

RTT-estimation algorithm. In our current implementation, to ensure an accurate RTT estimate, packets involved in the 3-way handshake should not be delayed at any other switch in the network, nor should the connection's RTT change. An inaccurate RTT estimate has the biggest effect on the sensitivity of the Apply actions module, as actions on flows with an overestimated RTT are applied later, allowing them to claim more bandwidth. Consequently, *P4air*'s fairness properties might decrease, but should remain as high as those of Idle *P4air*. There are three possible solutions to this problem: (1) the RTT-estimation algorithm can be extended to make use of the other switches (or at least all the bottlenecks) in the path to periodically summarize the total queuing delay (similar to [99]); (2) the RTT algorithm can be replaced with the one presented in [227]; (3) the Apply actions module could be modified to avoid the metrics that depend on the RTT estimate (e.g., queuing delay instead of BDP).

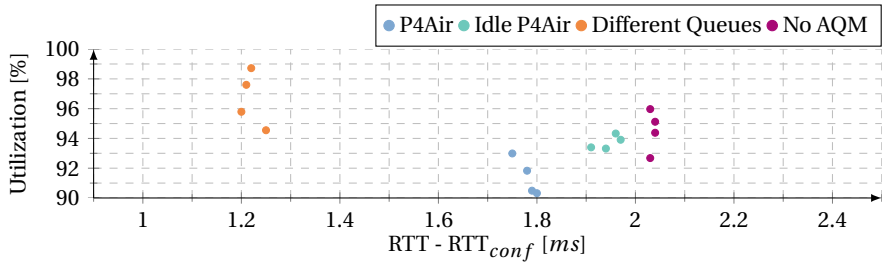
Queue-assignment imbalance. In the current implementation, we assume that flows complete uniformly across all queues. Otherwise, an imbalance in the queue assignment might occur, leading to more flows competing inside the same queue and receiving a lower share of the resources. To remove this assumption, *P4air* can be modified to keep track of the number of flows processed by each queue, as well as the identifiers of the queues (per group) having $\leq fpq$ flows. This way, by sacrificing more memory ($n_{queues} log_2(n_{flows}) + 4 log_2(n_{queues})$), we can make sure that all queues process a similar amount of flows by enqueuing new flows into the saved queue (with $\leq fpq$ flows).



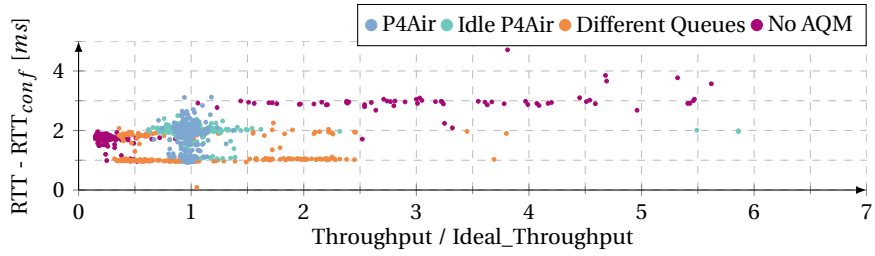
(a) Average RTT Fairness for 256 different flows. Each link delay was configured to a multiple of ΔRTT. All flows used the same congestion control algorithm (one of the four groups). For each ΔRTT, each group (4 different) is run 4 times (16 in total).

	$\Delta RTT = 1ms$				$\Delta RTT = 10ms$			
	Cubic	Illinois	BBR	Vegas	Cubic	Illinois	BBR	Vegas
P4air	0.92	0.81	0.90	0.91	0.73	0.55	0.83	0.55
Idle P4Air	0.93	0.51	0.69	0.91	0.66	0.49	0.38	0.48
Different Queues	0.68	0.50	0.58	0.60	0.39	0.31	0.19	0.52
No AQM	0.88	0.35	0.24	0.88	0.34	0.32	0.18	0.49

(b) RTT Fairnes for 256 different flows running the same congestion control algorithm for two different values of ΔRTT (two zoomed-in scenarios shown in Figure 6.12a). Each scenario is run 4 times (Confidence intervals 90%). Theoretical minimum is $1/256 \approx 0.004$.



(c) Average delay vs. average utilization for scenarios shown in Figure 6.12b. Ideal operating point is (0, 100%).



(d) Average RTT vs. throughput per flow for a scenario with 256 BBR flows with $\Delta RTT = 1ms$ (zoomed-in version of one of the scenarios shown in Figure 6.12b). Ideal operating point is (1, 0).

Figure 6.12: RTT-Fairness. Comparison of the *P4air* algorithm to the comparison baselines on a Intel Tofino switch.

Note that such an imbalance is also possible with all AQMs, which usually assign flows to queues based on the hash of a flow identifier.

Traffic shaping mechanisms & Multiple bottlenecks. Traffic shaping mechanisms (e.g., other AQMs, *P4air*) deployed at other switches on the path and/or the presence of multiple bottlenecks might impact the patterns tracked by *P4air* and lead to misclassification. As (some) flows are shaped already and thus perform fair, this may not be an issue (or they could form a separate "marked" group). Nonetheless, a more extensive evaluation of the impact of different shaping mechanisms and multiple bottlenecks on the fingerprinting accuracy is needed.

***P4air* placement in complex topologies.** With only a few strategically placed *P4air* switches, overall network behavior might benefit greatly. Developing a placement algorithm to determine the locations and amount of *P4air* switches in complex topologies has been beyond the scope of this work, but is important to consider when implementing P4air.

6.6. RELATED WORK

Many AQM algorithms (RED [62], ARED [63], SRED [64], FRED [65], REM [66], CHOKe [67], BLUE [73], AVQ [68], AN-AQM [69], DC-AQM [70], CoDel [71], PIE [72], SFB [73], SBQ) [74], SFQRED [75], FQ_CODEL [76]) were designed to detect and overcome static queues, reducing the queuing delay in the process. Classic AQMs, like RED, probabilistically drop packets based on the average number of packets inside a queue. However, studies have shown that their optimal configurations vary depending on parameters, such as capacity and number of flows, which causes network instabilities and traffic disruptions [65, 228–230].

Newer AQMs, like CoDel (Controlled Delay) and PIE (Proportional Integral controller Enhanced), were designed to overcome these issues and are, consequently, easier to manage and configure [71]. Moreover, when combined with scheduling algorithms providing isolation, such as FQ, they ensure high fairness at a wide range of bandwidths and flows. However, as a reaction to queue build-up, they can only drop a packet. Hence, they (1) cannot target newer congestion control algorithms that do not use loss as a metric, (2) lead to (potentially unnecessary) retransmissions, (3) usually require many queues, which are scare resources in hardware switches, and (4) do not take advantage of the inherently good intra-fairness properties that most congestion control algorithms have.

In contrast, solutions such as Virtualized Congestion Control create a translation layer in a hypervisor, enabling an easy upgrade of legacy algorithms and offering a data-center operator the ability to implement a single fair algorithm [231]. However, multiple issues might occur: (1) tenants might expect more isolation, (2) it is unusable in cases when flows are originating from multiple data-centers (using different "fair" algorithms), and (3) the solution can only implement certain TCP flavors by violating the TCP end-to-end semantics (i.e., acknowledging the packets not yet received).

6.7. CONCLUSION

In this chapter, we first developed a fingerprinting algorithm that harnesses the power of programmable data planes to detect the congestion control algorithms used by flows.

By instructing the bottleneck switch to track very simple metrics, such as queuing delay and sending rate, our algorithm is able to track the way the flows react to specific events (e.g., queue buildup), allowing it to classify the flows into one of the delay-, loss-, and hybrid (and its sub-groups delay-loss, and model-based) groups. Second, we used this knowledge, and the fact that most algorithms have very good intra-fairness properties, to enqueue flows using similar algorithms into the same queue. Third, for each of these groups, we developed custom actions, able to target their specifics, which we used to punish the most aggressive flows. P4air incorporates the aforementioned three elements, thereby enabling a switch to target each flow specifically and ensure a fair distribution of resources among all flows in the process.

7

IN-NETWORK FAST CONGESTION DETECTION AND AVOIDANCE

In the previous chapter, we presented a solution called P4air *that can ensure fairer resource distribution and lower queuing delay when deployed on a switch. However, P4air treats all flows in the same way and, to be effective, relies on the end-hosts, implementing the congestion control algorithms, to react to the applied actions and back off upon detecting congestion. However, some low-latency applications, such as remote surgery, might not rely on protocols with such a feedback loop. Additionally, they require extremely low end-to-end latency and almost zero packet loss and, consequently, cannot be targeted by P4air.*

To provide high QoS for these flows and to target protocols without a feedback loop (such as UDP), we use programmable data planes to gather and react to important packet metadata, such as queue load, while the switch processes the data packets. To do so, we develop a method for congestion avoidance in which switches (1) track processing and queuing delays of latency-critical flows, and (2) react immediately in the data plane to congestion by rerouting the affected flows. Through a proof-of-concept implementation in software and on real hardware, we demonstrate that our data plane approach reduces jitter and average and maximum delay compared to non-programmable approaches.

This chapter is based on a published workshop paper: B. Turkovic, F.A. Kuipers, N. van Adrichem, K. Langendoen, *Fast network congestion detection and avoidance using P4,* Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies, 45-51 (2018) [106]

7.1. INTRODUCTION

For long, available network capacity has been the most important Quality-of-Service (QoS) parameter to optimize for. Recently, with the emergence of novel application domains such as the Tactile Internet – where the objective is to transport a sense of touch over the Internet – and supporting communications technologies such as 5G, low latency has also become a crucial QoS parameter. Tactile Internet applications need very low latency ($\approx 1ms$), low jitter, high bandwidth (in the order of Gbps) and high reliability [131, 232].

Tactile Internet traffic could be very bursty, depending on the required modalities (audio, video, and/or haptic) and the compression levels that can be achieved at a given time. While haptic prediction algorithms might relax the latency requirement, consistent feedback and a maximum delay bound are necessary for a haptic system to be stable. Consequently, to minimize the end-to-end latency, packets of Tactile Internet flows should not be delayed on any node on the path nor be dropped by the network. This requires network nodes to be able to quickly detect and react to any changes in the network state, such as buffers filling up.

As explained in Chapter 1, a packet typically encounters four types of network delay, namely transmission, propagation, processing, and queuing delays. Out of them, the propagation and transmission delays only depend on parameters such as the physical distance, propagation speed, packet size and/or the data rate of the link, and as such are constant for a packet of some constant size on a given path (Chapter 1). However, the processing and queuing delays depend on the amount of traffic and how it is handled in the network. As such, they may vary significantly and controlling and reducing them is of importance and therefore the main topic of this chapter.

7.1.1. PROBLEM DEFINITION

One of the most important factors that contributes to queuing delay is congestion, which occurs when a network node is trying to process more data than the link can periodically handle.

Congestion control mechanisms of traditional transport protocols such as TCP detect congestion at the sender node and modify the sending rate accordingly. In the case of tactile traffic, such an approach is not feasible as we are not allowed to buffer or increase/decrease the rate at the tactile source. Furthermore, many congestion control algorithms only kick in after congestion has occurred and need at least one round-trip time (RTT) to react to the perceived congestion. Software-defined networking (SDN [78, 79]), as a new paradigm in networking, offers an alternative. Because every node in the network is controlled from software-based controllers, these controllers have a centralized view of the network and are able to react and adapt to changing network conditions faster. A common method to provide QoS in SDN is to implement virtual slicing of the available bandwidth on all the nodes on the path, reserving parts of it for different services or to use priority queuing. But, as the required bandwidth can be in the order of a few Gbps [131], reserving the maximum required bandwidth for every flow is not scalable. Priority queuing, while initially minimizing queuing delay for the higher prioritized flows, can lead to starvation of flows and does not prevent congestion. In fact, high-priority flows will starve when congestion forces low-priority flows

to occupy all available queue space. Alternatively, IEEE 802.1TSN works on standardizing specialized schedulers for Time-Sensitive Networks (TSN) such as time-aware traffic shapers [233], though those solutions require a closed-circuit network to operate. There are many frameworks that use some form of QoS routing to find the path that satisfies different QoS requirements. However, SDN frameworks from this group depend on some form of monitoring ([87, 88]). Incorrectly set monitoring intervals have direct influence on the usefulness of the gathered data as well as the number of probe packets sent. Additionally, after congestion is detected, a certain time is needed for the controller to recompute the path and reconfigure table entries before switching the flow to a better path. To avoid the aforementioned artifacts, the main problem to be solved is *How to enable congestion control and avoidance in the forwarding nodes?*, instead of at the source or via a controller.

7.1.2. MAIN CONTRIBUTIONS

In Section 7.2, we propose a hierarchal control model for latency-critical flows. Our solution contains a small program running directly on the switches and has real-time access to latency monitoring data to quickly reroute traffic when degradation is detected.

In Section 7.3, we evaluate our solution both by emulation through software switches as well as with physical P4 hardware. We compare our solution against a congestion-agnostic approach as well as to congestion-avoidance approaches that make use of probing.

7.2. CONGESTION DETECTION AND AVOIDANCE IN THE DATA PLANE

If end-to-end delay, as well as jitter, needs to be kept under a certain threshold, the main challenge is to detect and react to any increase in delay when data is being processed at the switches and not (only) at the source. If every switch minimizes the total delay per node (for certain traffic flows) to a configurable value, maximum end-to-end delay becomes predictable.

Recently, in the wake of SDN, programmable switches have appeared along with domain-specific programming languages such as P4 to program them [140]. P4 offers the possibility to gather and export important packet meta-data (timestamps from different stages of processing, queue depths, etc.) directly from the data plane while the data-packets are being processed. To leverage this unique possibility of collecting packet meta-data, we propose a hierarchal architecture, as shown in Figure 7.1, to detect and avoid congestion:

A local control module at the P4 switches, as elaborated on in Section 7.2.1, monitors the state of all the low-latency flows, while a central controller configures the latency thresholds and other parameters.

7.2.1. LOCAL CONTROL

A local Congestion Detection and Avoidance module, see Figure 7.2, is developed to monitor the processing and queuing delays.

If the module determines that one of these delay components is increasing for a

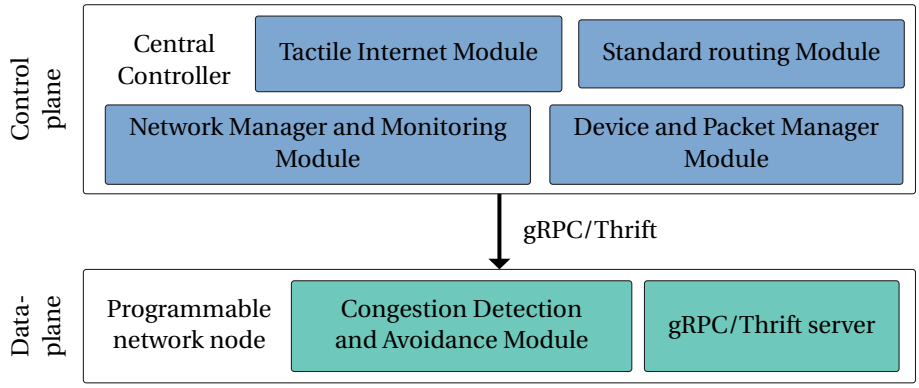


Figure 7.1: Hierarchical design of the control plane.

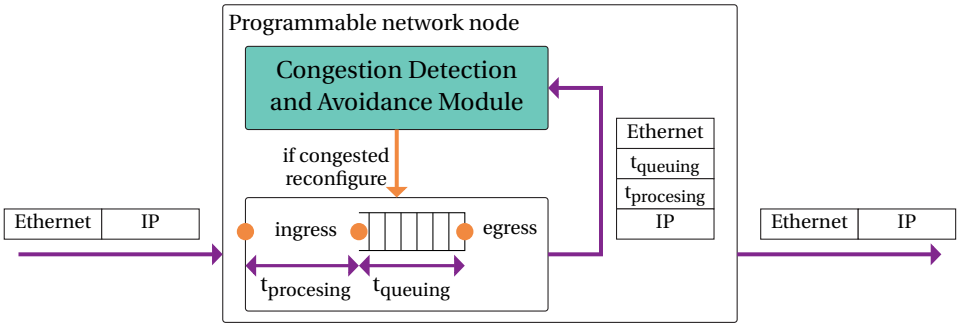


Figure 7.2: Detection of congestion in the data plane. Every switch has a small congestion avoidance module, gathering statistics (processing and queuing delay).

latency-critical flow, and congestion is likely to occur, it preemptively switches the traffic to a better backup path if it exists or signals to the previous node in the path that it is congested and that it should not forward any more packets belonging to that flow.

According to the P4_14 [234] and P4_16 [235] language specifications, table entries at a switch cannot be modified without the intervention of the control plane (controller). Thus, in order to achieve rerouting in the data plane we are left with two choices: (1) add both entries (backup and primary) to a table and decide which rule to apply based on some meta-data stored in the registers, or (2) send packets or packet digest notifications to a local listener that tracks the flows and acts as a small local control plane.

If we use meta-data and registers, we need to apply more tables, increasing processing delay per packet, and store more table entries in the switches than necessary. Additionally, because meta-data and register values would affect table lookups, there will be no gain in doing table lookup caching, which can have a significant influence on performance.

The second option uses packet copies or packet digests, which are two of the mechanisms by which the data plane can send notifications to the control plane. One of the

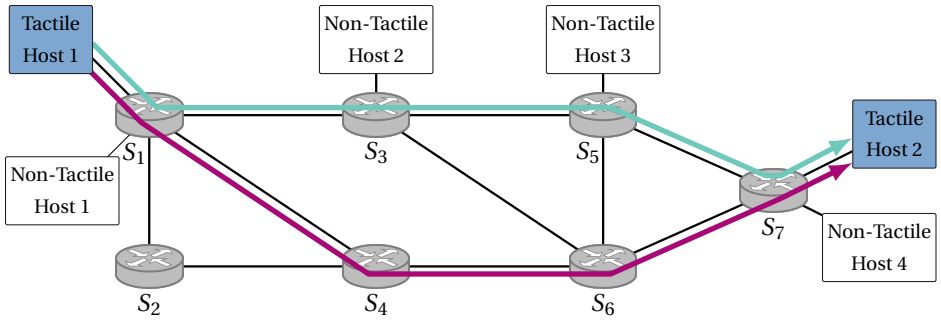


Figure 7.3: The green line is primary path and the purple line is the path that traffic will take in case any of the routers in the path detects congestion. To be effective, both paths have the same weight and are equally good.

advantages of a digest packet is that it is typically much smaller than the original packet. Also, different digest messages can be used for different types of traffic, making this approach easily extensible without any significant changes in the data/control plane. We have implemented a digest listener that is running on the same machine as the switch itself and makes local routing decisions based on the digest data.

This module, based on the measured values of the queuing and processing delays, reroutes the flow if it detects an increase in delay for the subsequent m packets. While digest notifications are very small, the rate at which they are generated can be very high if we want to obtain delay information about every packet on the path. To avoid overloading the local control application, we decided to shift the detection of the congestion to a P4 program and use digests to generate congestion notification to the local module only when the delay increases above a certain threshold.

The number of packets m, as well as the threshold values for queuing t_q and processing t_p delays are configurable and depend on the type of hardware used, as well as the sensitivity needed. If the thresholds are too small, the local controller might reroute traffic unnecessarily, potentially increasing the jitter as well as creating additional load on the central controller that needs to recalculate a new backup route, delete rules from the old primary path and install new backup rules. Otherwise, if too big, the controller might react too late, detecting congestion too late and thus providing little increase in performance when compared to legacy solutions.

7.2.2. REROUTING EXAMPLE

In order to have adequate backup paths available, for every new tactile flow two paths that satisfy the latency requirement of that flow are calculated, as shown in Figure 7.3. The green path (s1-s3-s5-s7) is used as the primary path. The purple path (s1-s4-s6-s7) is used as backup. Multiple primary and/or backup paths could also have been used, but we have opted for single paths since it requires less processing (in terms of packet re-ordering) at the end-node.

In case an increase in queuing delay is detected on switch s1, the local control program will change the output port for this flow to s4 (as shown in Figure 7.4). Switch s1 is

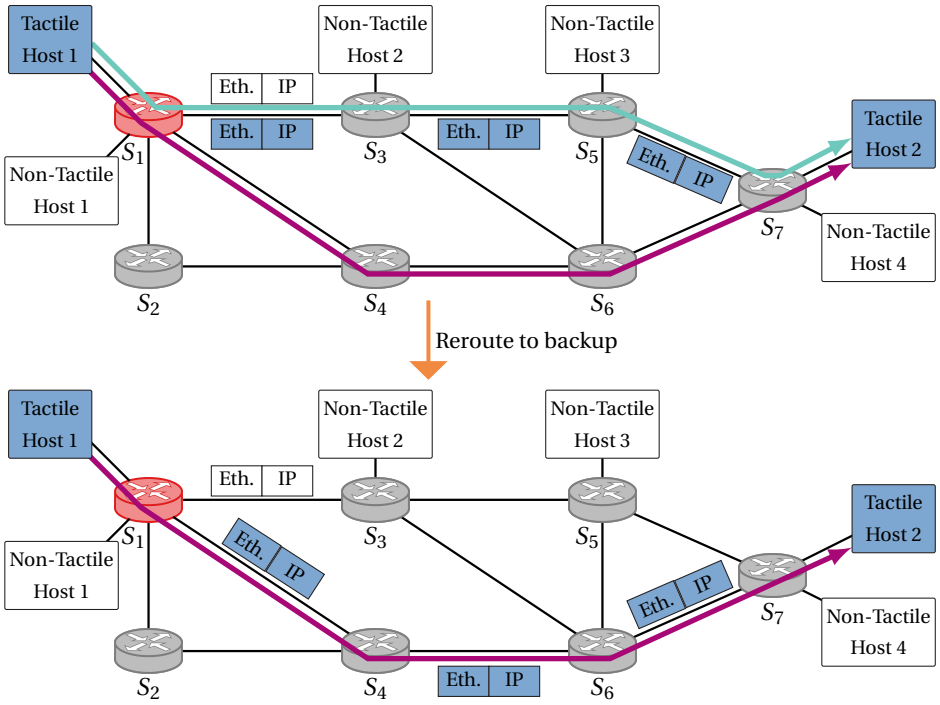


Figure 7.4: If a new flow is initialized between non-tactile hosts 1 and 2, the total traffic processed on the output link on switch S_1 (to switch S_3, that is part of the primary path for tactile traffic as well), might exceed the available bandwidth. Consequently, switch S_1 will detect this, and since it already knows a better backup route it will consequently reroute the traffic to S_4. Finally, the backup route (purple) becomes the new primary route, and the controller calculates a new backup route and removes the previous primary route (green).

7

still used, as output ports and consequently output queues are different and not affected by the detected queue build-up. The tactile route is already configured on switches s4 and s6 and thus rerouting is achieved instantly. The local control programs at the switches determine the output port based on the input they receive from a central controller that tracks the link utilizations and delays on all the nodes in the network by sampling the network constantly.

In case switch s5 detects congestion (increased queuing delay on the tactile queue to s7 or increased processing delay) it has no better route configured and can thus only signal to its predecessor that the output link is congested. It thus sends a control message (congestion notification) to s3, who forwards it to s1. When s1 receives this message it will switch the affected tactile flow to s4 (as shown in Figure 7.5).

It is important to notice that links of the backup path are different than the links of the primary path after s1 (the switch that can actually perform the fail-over). The paths are calculated this way in order to prevent the backup path from forwarding the traffic to the same congested link as the primary path. The central controller, which has knowledge about the whole network, computes these paths periodically, based on the

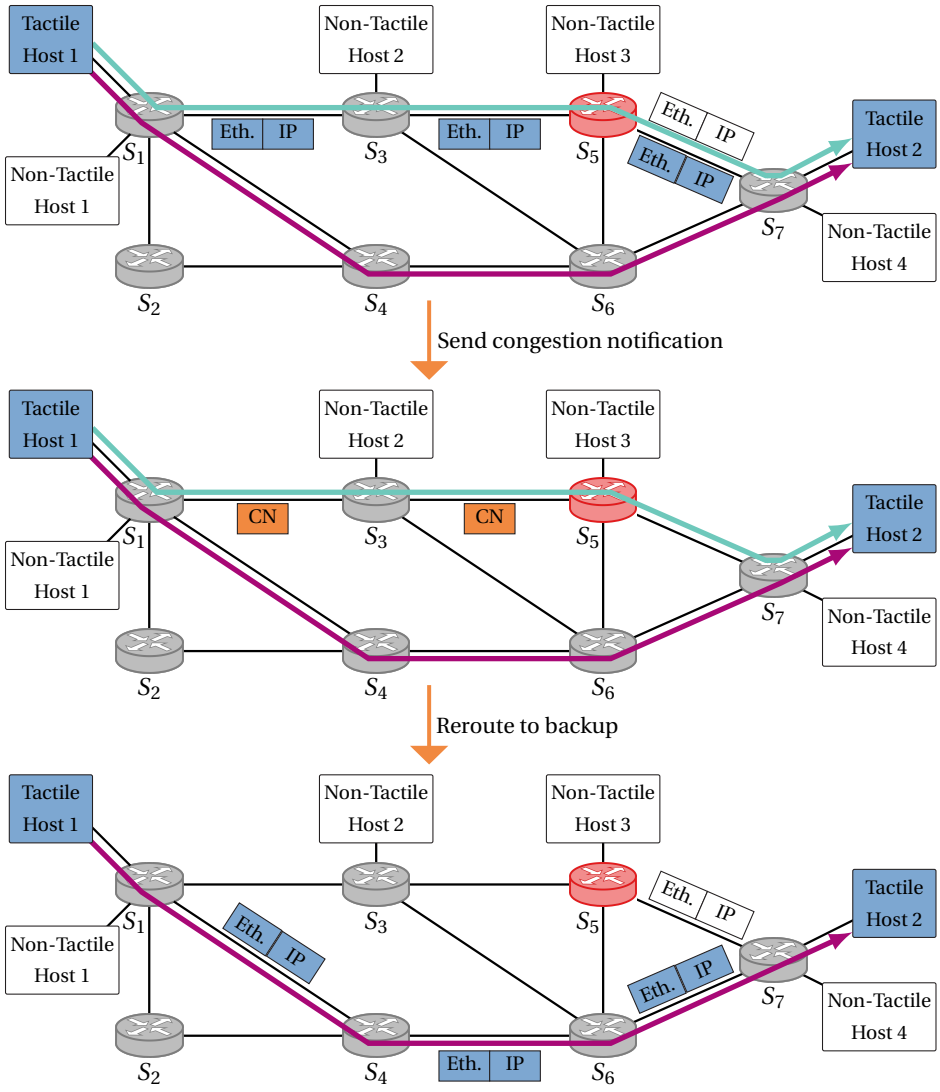


Figure 7.5: If a new flow is initialized between non-tactile hosts 3 and 4, the total traffic processed on the the output link on switch S_5 (to switch S_7, that is part of the primary path for tactile traffic as well), might exceed the available bandwidth. Consequently, switch S_5 will detect this, and since it does not know a better route to tactile host 2 (backup route) it will send a Congestion Notification (CN) packet to switch S_3 informing S_3 not to send packets through him. When S_1 (that knows the backup route) receives this control packet it will consequently reroute the traffic to S_4. Finally, the backup route (purple) becomes the new primary route, and the controller calculates a new backup route and removes the previous primary route (green).

current network state. When a flow is switched to a backup route, that route becomes the new primary path and a new backup must be computed and installed.

7.3. EVALUATION USING EMULATION

We have evaluated our solution, via the set-up described in Section 7.3.1, using the Mininet emulation environment with the P4 software switch (behavioral model, nicknamed *bmv2* [141]).

7.3.1. EXPERIMENT SETUP

Multiple flows were generated and RTT, maximum RTT, packet loss, as well as ingress processing, queuing, jitter, detection and reaction delays were measured. Per tactile flow, one primary and one backup route were configured. Additional traffic was generated to create congestion on different intermediate nodes on the primary route. Each tactile traffic trace was 15 seconds long, and these scenarios were repeated 40 times.

We varied the detection threshold for processing and queuing delays, as well as the number of consecutive packets m that need to have an increased delay in order to detect congestion. Scenario DataplaneX_m represents a scenario where the thresholds for processing and queuing delays were X times the processing and queuing delays on the switch if no additional load was generated and m is as defined before.

Our approach was compared to (1) an approach that uses no congestion detection and never recomputes paths (scenario No CC), which is mimicking traditional routing protocols such as OSPF and (2) an SDN-like approach that uses a centralized controller and periodically sends probe packets (scenarios Probing$X$sec), to determine the current network state and detect congestion. We used different monitoring intervals, namely 1, 2, and 5 sec.

7.3.2. MININET RESULTS

We have used the network topology displayed in Figure 7.3. The rate of all the bmv2 output queues was limited to 170000 pkts/s ($\approx 200Mbps$) in order to make sure that there would be a queue build-up. With this configuration, as the packet arrival rate is smaller than what the bmv2 ingress pipeline can process ($\approx 1Gbps$ on a server in our testbed), the bandwidth, and not the processing is the bottleneck. If the rate of the output queues is not limited, when the maximum throughput is reached, packets are dropped before the ingress pipeline, and there is no queue build-up, since the egress pipeline is usually faster than that of the ingress in bmv2.

In our scenario, 8000 packets per second ($\approx 4Mbps$ with a packet size of 64B) were injected by the tactile flow that we were interested in. If the amount of additional traffic was below the configured bottleneck bandwidth of $\approx 200Mbps$, the switches could process the low-latency data at line rate (Figure 7.6). When the volume of additional traffic approached 200 Mbps, the delay on node s3 increased, as the total amount of traffic exceeded the configured rate of the output queue. This was also the point were all the evaluated approaches correctly detected congestion and reconfigured the path for the tactile data.

Detection time. In the probing scenarios, as the controller uses increased delay of probe packets as an indication of congestion, the smaller the probing interval, the faster the controller was able to detect congestion, as shown in Figure 7.6c. As the volume of additional traffic increased, the number of dropped probe packets, as well as the maximum

delay, increased as well. In these scenarios, when no probe was returned within the probing interval, the controller assumed that the packet was lost and the link congested. This is why the detection delay in Figure 7.6c is higher than expected (half the monitoring interval). By comparing the values for the maximum detection delay, we observed that in the worst case it is approximately two times the value of the probing interval, which corresponds to one probe packet being sent immediately after congestion (in the queue build-up phase) and the subsequent packet being lost. Thus, the controller needed to wait for the timeout value (one monitoring interval) to expire.

In case the detection was done using the measurements in the data plane, the controller was always able to detect the changes very fast, by observing the data itself independently of the probe packets that were sent. The advantage of this approach is especially noticeable when detection time is compared to other approaches, as shown in Figure 7.6c. An increased number of subsequent packets m (Dataplane12_20) increases detection delay. However, this increase is very small when compared to scenarios Probing$X$sec.

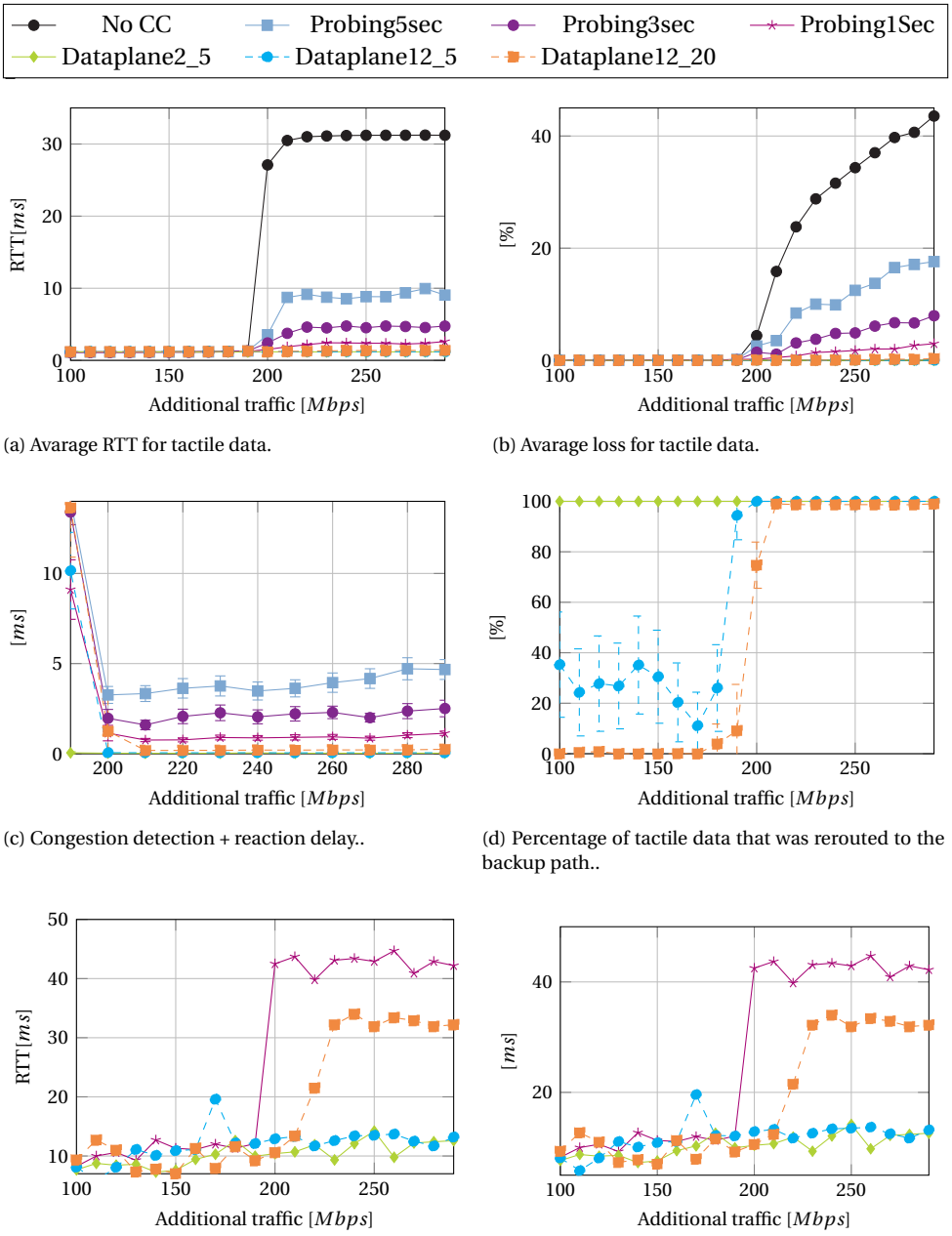
Reaction time. After detecting congestion, in case of the probing scenarios, the controller needed to find a new route and install new table entries starting from the end of the path in order to minimize the number of dropped packets. After traffic was switched, some packets were still present in the queues of the congested node. Consequently, packets arrived in the wrong order at the endpoints.

All data plane schemes only needed to update one table entry. The switches could immediately forward traffic on the new path and thus the total time needed to switch the traffic was minimized.

Delay and jitter measurements. The data plane schemes, as a consequence of fast detection, had the lowest average and maximum delay, as can be seen in Figure 7.6e. Increasing the number of subsequent packets m has a negative influence on the maximum delay, as well as maximum jitter, especially in case of very high additional traffic.

Average loss. In the case of no congestion control (scenario No CC), packets were queued until the buffer limit on s3 was reached, causing an increased number of dropped packets as can be seen in Figure 7.6b. All probing scenarios (Probing5sec, Probing3sec and Probing 1Sec) were able to detect and reduce the number of dropped packets. By comparing the loss values, we can see that data plane approach was the only one that could keep the loss value at 0%. For the probing solutions the loss increased with the amount of additional traffic, due to faster overruns of buffers.

Artifacts caused by the environment: One of the identified problems was that, depending on the configured threshold for the detection in the data plane, the probability of false negatives was significant (scenario Dataplane2_5). In these scenarios, although the threshold was set to twice the value of the queuing delay when no additional traffic was generated, switches detected congestion every time. By increasing the value of the threshold, or the number of subsequent packets needed, the value of false positives can be reduced, as shown in Figure 7.6d, while maintaining the QoS parameters at almost the same level.



(a) Avarage RTT for tactile data.

(b) Avarage loss for tactile data.

(c) Congestion detection + reaction delay..

(d) Percentage of tactile data that was rerouted to the backup path..

(e) Maximum RTT for tactile data.

(f) Comparison of maximum variation in RTT..

Figure 7.6: Mininet scenario (Confidence intervals 90%). Comparison of different QoS parameters for different schemes when congestion is present.

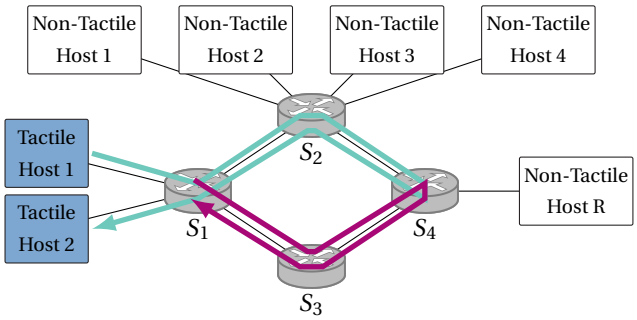


Figure 7.7: Hardware topology

7.4. PROOF OF CONCEPT USING P4 HARDWARE

7.4.1. EXPERIMENT SETUP

We have built a proof of concept using our P4 hardware testbed that consists of physical general-purpose servers enhanced with smart network interface cards (Netronome Agilio CX 2x10GbE), which were connected as shown in Figure 7.7. All the servers used Thrift RPC as the control interface and ran Ubuntu with kernel version 4.10.

Two different data plane approaches were evaluated. The first one (DP_direct) did all the processing in the data plane, while the second one (DP_listener) implemented the detection of delay increase in the data plane and did all the subsequent processing of the notifications in the local digest listener module. Our approach was compared to an approach that does no congestion control (scenario No CC, as in Section 7.3) as well as to an approach that uses periodic sampling of the current state of the network stored in switch registers (Probing1sec-5sec). All scenarios were repeated 50 times.

A tactile flow was generated between switches s1 and s4, while additional traffic was passing between hosts h1-h4 and hr, generating congestion on the output port of switch s2. The tactile flow had a throughput of 20 kpps ($\approx 240 Mbps$ with packet size of 1500 B), while the additional traffic had a throughput of 1.5 Mpps (where the packet size varied between 64 B and 1100 B), creating load in the range of ≈ 750 Mbps to ≈ 13 Gbps. The first and last second of the trace were not taken into account for latency and jitter measurements and additional traffic started 2s after the tactile traffic in order to observe queue build-up. To achieve high accuracy (nanosecond range), as well as to limit the influence of external factors (e.g., processing in the driver, kernel, etc.), latency was measured in the data plane at switch s1. Every tactile packet that was processed was equipped with an additional header field storing a 64-bit ingress time-stamp (when the packet was received from t1) or an egress time-stamp (when the packet was forwarded to t2). Since there is no external syncing between the switches, tactile traffic was routed back from s4 to switch s1, which inserted both timestamps, as shown in Figure 7.7.

7.4.2. HARDWARE LIMITATIONS

We encountered several limitations when we evaluated our scheme using the above-mentioned testbed. In an initial experiment, while measuring the ingress and egress processing delays, the delay between these two stages (which should represent queuing delay) was constant, even when the switch was congested and the total end-to-end delay increased. Because we were unable to obtain queuing delay information directly from the P4 program, we measured the total delay on the switch (from the ingress MAC component to the egress MAC component). An ingress time-stamp (the time in nanoseconds when the ingress MAC component receives the packet) was attached to the packet data structure while the packet was being processed at the card and could be inserted by the P4 program itself. In order to get the egress time-stamp, we added a special 32-bit header to the start of the packet. When the egress MAC component of the SmartNIC receives this special header it attaches the egress counter-value "time-stamp" and forwards the packet to the next switch. Since no external syncing is implemented, counter values can only be used for latency measurements inside one card. A subsequent switch in the path keeps track of the difference between these values in a register and based on that value decides whether the previous switch is congested or not. If it determines that the previous switch is congested it will send a congestion notification back, that, when received by s1, will trigger the rerouting to the backup path (s1-s3-s4). Thus, the detection of the congestion was shifted by one node, increasing the reaction time when compared to the emulated environment.

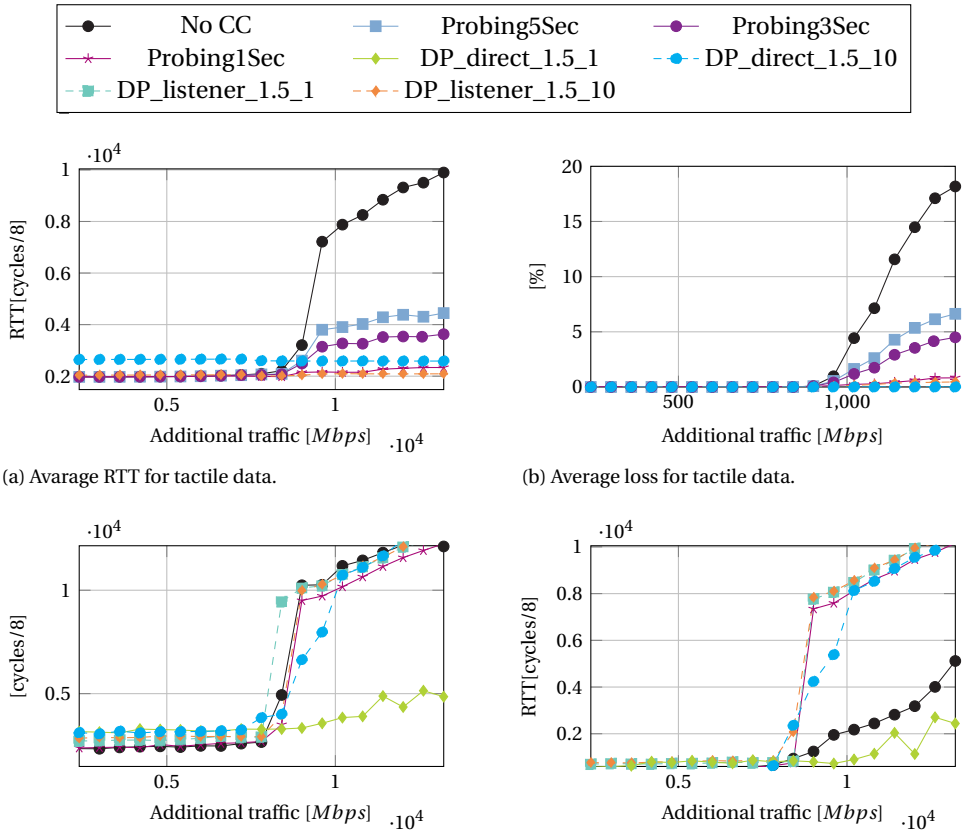
7.4.3. NETRONOME AGILIO CX SMARTNICS RESULTS

Measurements shown in Figure 7.9 demonstrate a functional proof of concept. All the evaluated data plane approaches, DP_listener and DP_direct, outperformed the other analyzed approaches by keeping all the analyzed QoS parameters on par with scenarios where no congestion was present. While we have plotted only DP_direct_1.5_10 and DP_listener_1.5_10 in Figures 7.8a and 7.8b, all other analyzed data plane scenarios had similar performance.

Average and maximum delay When the switches were not congested, the data plane approaches, as a consequence of additional processing, had higher average and maximum delay than the other evaluated approaches (from $\approx 1,900$ in scenario No CC to $\approx 2,000$ for Dataplane_listener and $\approx 2,500[cycles/8]$ in case of Dataplane_direct). The significant increase in average as well as maximum delay for the Dataplane_direct scenario is a consequence of a more complex data plane implementation, since multiple tables and registers are needed to keep the per-flow state.

For higher volumes of additional traffic, only direct data plane approaches were able to keep the maximum delay at the same level as before, as shown in Figure 7.8a. Switch s1 was the one that rerouted the traffic, causing delay between detection and reaction. Even the data plane approaches were unable to keep the maximum latency value below a certain threshold, especially for higher m.

Maximum jitter was lowest for No CC approach and DP_direct (Figure 7.8d). The relatively high jitter for the other approaches is a consequence of switching paths. The first packet that is processed on the backup path has a very low RTT compared to the ones that are still processed by the congested nodes.



(a) Avarage RTT for tactile data.

(b) Average loss for tactile data.

(c) Maximum RTT for tactile data.

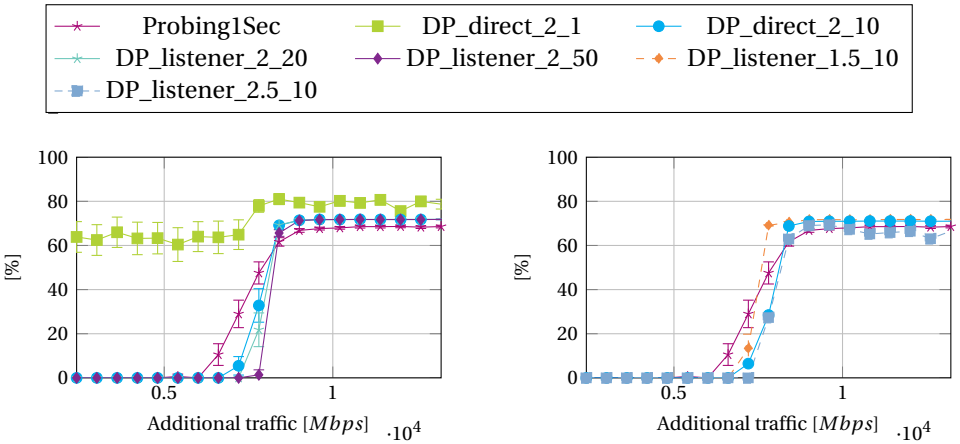
(d) Maximum jitter for tactile data.

Figure 7.8: Netronome SmartNIC scenario (Confidence intervals 90%). Comparison of different QoS parameters for different schemes when additional traffic is generated to create congestion at node s2.

Congestion detection and sensitivity. Increasing the detection threshold, shown in Figure 7.9b, has a negative influence as we miss the start queue buildup phase and, consequently, more packets are affected by congestion. By decreasing the threshold, even when the value of additional traffic was not high enough to cause congestion, all analyzed approaches (including the Probing scenarios) detected it. In cases when both primary and backup paths have high link utilizations, this behavior may lead to too many recalculations and path switching, which would degrade the overall performance. This can be resolved by increasing the number of packets used to detect congestion, as shown in Figure 7.9a.

7.5. CONCLUSION

To quickly detect and avoid congestion within a network, we have proposed a P4-based technique that enables measuring delays and rerouting in the data plane. Our approach



(a) The influence of the number of packets m used to detect congestion on the average percentage of packets that were processed on the backup path.

(b) The influence of detection threshold $(t_p + t_q)$ on the average percentage of packets that were processed on the backup path.

Figure 7.9: Netronome SmartNIC scenario (Confidence intervals 90%). Comparison of different QoS parameters for different schemes when additional traffic is generated to create congestion at node s2.

offers two main advantages. First, the detection time is reduced and congestion is detected per flow. Thus, only the affected flows are rerouted and QoS degradation of other flows is avoided. Second, after detection, the reaction time is minimized as a local controller, based on input from a central controller, intervenes by configuring a better route. Therefore, no new flow rules need to be installed and the load on the central controller is reduced.

We encountered some limitations during evaluation of our solution using Netronome P4 SmartNICs, such as a limit on range matching, a performance penalty due to disabled caching, as well as lack of information about the queuing delay of the current switch. Nonetheless, we were able to show the feasibility of our solution in both emulated and physical networks.

While the presented approach requires specialized hardware (P4-capable switches), in a hybrid network where only some nodes can be programmed, detection time as well as reaction time to congestion might still be reduced using this scheme, when the nodes are placed at crucial points in the network such as the network edge. Additionally, our solution can easily be extended to a solution that uses bandwidth reservation and/or priority queuing.

8

CONCLUSION

In the thesis, we systematically proposed, implemented, and evaluated solutions to enable low-latency services as well as the coexistence of these services with the traditional bandwidth-oriented services. Our solutions, built upon the premise that future networks will have both programmable control, and data planes, demonstrate how offloading the sensitive tasks (e.g., load detection, congestion control group detection, congestion detection) to the data plane significantly improves the reactiveness and flexibility of a network, a crucial requirement for emerging services.

This chapter first addresses the sub-questions posed in the introduction and, subsequently, summarizes them to answer the main research question. Finally, it discusses possible future research directions that could further increase the flexibility of networks and, consequently, enable the transparent coexistence of multiple services with different network requirements.

8.1. RESEARCH QUESTIONS & CONTRIBUTIONS

In addition to the main research question, this thesis identified three sub-challenges. Bellow, key contributions to each of them are listed.

What techniques can be used to overcome the challenges associated with programmable hardware when designing and deploying network algorithms?

Chapter 2 investigated the limits imposed by programmable switches through developing an application to detect heavy hitters. We demonstrated that, although programming languages such as P4 decouple the forwarding program from the hardware, the choice of the hardware platform still significantly impacts the design of the program and its feasibility. Accordingly, the programmer must stay aware of the hardware and tailor its program around its limitations (e.g., limited memory, a limited number of memory accesses).

For example, by introducing the counter reuse through modulo sketching, we demonstrated a way to reduce the heavy hitters program's memory footprint while maintaining high detection accuracy. Consequently, the switch could house more network func-

tions and/or more complex network functions (e.g., sequential window with more sub-intervals). Similarly, by introducing the Zeroing window, we demonstrated a way to overcome the stringent memory access limitations through ping-ponging the memories and "slowly" (i.e., one counter each *N/width* packets) resetting the counting sketches. Therefore, by altering our program to fit the way the switch processes packets, we could develop programs that would be infeasible at first glance.

Further, we noticed difficulties in syncing between the control and data plane in our hardware experiments, primarily when the size of the interval (or sub-interval in case of the Sequential Window) was reduced. The main reason for this was the speed difference between the control and data plane. Currently available programmable switches already have data planes that can process packets at speeds reaching *Tbps*. As a consequence, the control plane program that resets all analyzed interval-based sketches (and the sub-sketches in the Sequential Window) can quickly become the bottleneck. Ultimately, we avoided the control plane's intervention by combining all the introduced techniques and introduced the *Sequential Zeroing Window* sketch, a heavy-hitter detection solution with high accuracy that provides per-packet granularity at line-rate performance.

How can we allocate network resources to different applications in real-time by taking into account their individual requirements?

Chapter 3 and Chapter 4 focused on the ways resources can be assigned to applications/services. Chapter 3 illustrated that applications, such as remote teleoperation, have varying dynamics and that, consequently, their network requirements often stay far below their highest value. Therefore, a resource reservation solution that supports elasticity, i.e., the ability to automatically adjust the applications' (services') network resources based on their current needs, could achieve a high network utilization while ensuring a high QoS for all the applications' (services') flows.

However, as these chapters demonstrate, a *purely* control plane solution (with a "dumb" data plane) cannot satisfy the requirements of low latency applications due to: (1) the speed difference between the control and the data plane, and (2) the delays between the controller and the individual switches. Moreover, the observed degradation of the QoS was the highest in situations when the applications required very low latency and very high bandwidth (e.g., switching from a very static to a highly dynamic environment), which could be potentially critical for applications such as remote surgery.

To address the identified issues, in Chapter 3 we designed a resource scaling solution that split the tasks between the data plane and the control plane. With its global overview of the network, the control plane guides the switches by calculating and pre-configuring (in advance) the rules corresponding to different application dynamics. The calculated rules consider both the requested bandwidth and latency and ensured that the traffic corresponding to more dynamic environments (requiring lower latency and higher bandwidth) is routed over lower-latency paths and assigned more bandwidth. Based on these rules, the data plane executes bandwidth scaling and rerouting tasks, ensuring almost instantaneous elasticity and ensuring the users' experience is not degraded.

Chapter 4 built on top of the solution introduced in Chapter 3 and introduced a more comprehensive scaling solution that, in addition to being able to scale a single network path (e.g., by assigning more resources to it), also supports (1) stateful network func-

tions, (2) horizontal scaling (e.g., splitting the traffic flows from one service over multiple paths) and (3) does not depend on the end-applications to determine the current service resource requirements. To achieve that, two additional tasks were offloaded to the data plane: (1) load monitoring to detect the optimal scaling moments in the data plane; and (2) state management to maintain a consistent network state for stateful network functions.

To conclude, Chapters 3 and 4 demonstrated that offloading time-sensitive tasks to the data plane can enable almost instantaneous scaling of applications' network resources based on current traffic volumes and/or flow requirements while maintaining state consistency. Moreover, they showed that, due to the faster reaction speed of the data plane, such solutions could satisfy the QoS needed by the low latency applications.

How can we provide a fair resource distribution between different flows belonging to the same service, thus guaranteeing the same performance for all service users?

Chapter 5 investigated the interactions between different flows and their coexistence, specifically flows that use transport protocols such as TCP and QUIC. Congestion control algorithms used by these protocols use various metrics (e.g., lost packets, increase in the measured RTT) to determine if the network is congested and/or more bandwidth is available. Hence, in Chapter 5 congestion control algorithms were first divided depending on these metrics into loss-based, delay-based, and hybrid congestion control algorithms.

Next, through head-to-head comparisons of the representatives of these algorithms, Chapter 5 showed that resources are rarely distributed fairly between flows. Moreover, it showed that the fairness index was lowest when flows sharing a link had different round-trip times or used congestion control algorithms that belonged to different groups. In contrast, it demonstrated that flows using similar metrics had mostly high fairness.

Based on these observations, we used programmable switches in Chapter 6 to enforce fairness from within the network itself. Our solution *P4air* continuously monitored the properties of all flows passing through a switch and grouped them based on the behavior of the congestion control algorithms used. Hence, it relied on the good intra-fairness properties most congestion control algorithms were developed with and tried to avoid the competition for resources between flows belonging to different groups. Therefore, it showed that the resource distribution can be improved if more properties of the flows are taken into account and if flows present on a switch (or in a slice) are not treated equally.

Finally, in Chapter 7, we investigated how programmable switches can be used to provide a high QoS when the network (or slice) is shared between multiple types of flows: (1) very low latency and (2) best effort. To do so, we developed a data-plane program that can detect congestion and, instantaneously, reroute congested low-latency flows to alternative routes. In contrast to the solution presented in Chapter 6, it did not rely on the end-hosts' congestion detection algorithms (a process that can introduce a delay of multiple RTTs), nor did it expect the end-hosts to reduce their rate upon detecting congestion (a feature not supported by all protocols). In fact, this rerouting was done at the expense of other best-effort flows. Therefore, such a solution can be ideal for an application that should not reduce/adapt their rate due to their criticality (e.g., remote surgery), but share network resources with other less critical applications.

8.1.1. MAIN RESEARCH QUESTION

The following research question was proposed in the introduction of this thesis:

What mechanisms need to be developed and deployed in a programmable network to support low-latency applications?

To support different application domains simultaneously, a network slicing mechanism allocating to each application domain a part of the network tailored to its requirements will isolate the various domains and, consequently, prevent the competition for the underlying network resources from their respective users. Moreover, an additional in-slice mechanism preventing flows of the same slice from overpowering each other will ensure that different application/service users would achieve the same QoS. In addition to this, low-latency applications have very stringent latency requirements that must be taken into account when designing these mechanisms. Hence, as demonstrated in this thesis, these mechanisms need to be able to react on shorter time scales to prevent QoS degradation of the low-latency flows.

Accordingly, this thesis explored ways to combine programmable switches with the concept of software-defined networking (SDN) to implement the above-mentioned mechanisms. By combining these two, the central SDN controller can still use its global overview of the network and all the current flows and their requirements to assign the resources to different applications and configure the switches. However, at the same time, the network switches can react to short-term changes in the networking traffic by acting independently (or with limited input from the controller) as required for low-latency services.

Therefore, as part of the first sub-question, we explored the challenges of deploying algorithms to the programmable switches and demonstrated ways to overcome these challenges. Next, as part of the second sub-question, we explored ways to implement the first identified (network slicing) by identifying the latency-sensitive tasks (e.g., monitoring, rerouting), and offloading them to the data plane. Moreover, through a thorough evaluation, we demonstrated the benefits of doing so for the low-latency applications. Finally, to ensure fairness between multiple flows of the same service (third sub-question), we investigated the fairness properties of TCP/QUIC flows. After demonstrating that flows generally overpower each other when they employ different congestion control algorithms and/or have different RTTs, we proposed a data plane solution that ensures fairness.

8.2. FUTURE WORK

In this section, we offer some ideas for future work for the topics covered in this thesis. While we tackled some of key challenges for the deployment of low-latency services, there still exist many open challenges. We list some of them bellow.

Interactions between other transport and/or application protocols. Chapters 5 and 6 examined the interaction of common TCP/QUIC congestion control algorithms. While these protocols constitute most of the traffic today, they might not be suitable for many real-time applications. Hence, many research papers introduced new transport and application-specific protocols. For example, for Tactile Internet applications, many other

protocols, such as the Interactive Real-Time Protocol (IRTP), Supermedia TRansport for teleoperations over Overlay Networks (STRON), Haptics over Internet Protocol (HoIP)), Interoperable Telesurgical Protocol (ITP), can be used. Consequently, more research into the interactions and coexistence of these protocols is needed.

Support for slices with different requirements. Chapters 3 and 4 introduced solutions that assign network resources to flows (or group of flows) whenever necessary (e.g., when a flow requests it, a slice is fully utilized). While this improves the QoS of critical flows and ensures that critical procedures such as remote surgery can be conducted, it is unnecessary for many other applications with less stringent requirements (e.g., video streaming). However, similar approaches as proposed in this thesis can be used for these services to ensure high slice utilization and avoid overprovisioning. For example, by adjusting the load detection module and the scaling decision process in the central controller and choosing different "inter-slice" mechanisms, slices with different requirements can be supported. More research is needed to determine the impact of the arrangements of each of the presented modules on different application domains. Moreover, such analysis could result in a generalized plug-and-play solution that would have a different "template" for each application domain.

On the fly reconfigurable programmable switches. To reconfigure and/or reshape a network slice on the fly, programs running on programmable switches must be replaceable without any downtime and/or interruption to the flows processed by the switch. This way, the network operator can add/remove a network function (NF) to/from the switch, adding/removing custom functionality to/from it. Moreover, while doing that, the switch would still process the other (potentially stateful) network functions (potentially belonging to other slices and processing traffic belonging to other tenants).

However, currently available programmable hardware does not support this functionality. Hence, the only way to reconfigure a switch is to recompile the program and deploy the new firmware to the switch. However, this not only introduces downtime (sometimes in the order of minutes), but also erases all the internal states needed for the correct functioning of many network functions. One way to avoid this problem would be to have all the network functions already implemented in the switch with a register array to indicate which ones are used at the moment (an approach used in Chapter 4). While such a solution has zero downtime, it can only support a limited number of network functions, and it does not allow for the addition of novel network functions on the fly. Several approaches were proposed to address this problem, usually by creating an additional virtualization layer (e.g., a P4 program that can emulate all other P4 programs by changing runtime table rules). However, while such approaches have zero downtime, they also have a considerable overhead, even for elementary P4 programs, and usually do not support stateful network functions. Consequently, additional research is needed to enable this crucial functionality.

REFERENCES

[1] Sandvine., *The mobile internet phenomena report,* Tech report (2019).

[2] Sandvine., *The global internet phenomena report,* Tech report (2019).

[3] I. Grigorik, *High Performance Browser Networking: What every web developer should know about networking and web performance* (" O'Reilly Media, Inc.", 2013).

[4] D. Van Den Berg, R. Glans, D. De Koning, F. A. Kuipers, J. Lugtenburg, K. Polachan, P. T. Venkata, C. Singh, B. Turkovic, and B. Van Wijk, *Challenges in haptic communications over the tactile internet,* IEEE Access **5**, 23502 (2017).

[5] G. P. Fettweis, *The tactile internet: Applications and challenges,* IEEE Vehicular Technology Magazine **9**, 64 (2014).

[6] T. Driscoll, S. Farhoud, S. Nowling, *et al.*, *Enabling mobile augmented and virtual reality with 5G networks*, Tech. Rep. (Tech. rep. Tech. Rep, 2017).

[7] G. Intelligence, *Understanding 5g: Perspectives on future technological advancements in mobile,* White paper , 1 (2014).

[8] 5G Applications Market Potential & Readiness Matrix., *5G Applications Market Potential & Readiness Matrix.* https://www-file.huawei.com/-/media/corporate/pdf/x-lab/5g-applications-market-potential_readiness-matrix.pdf?la=nl-nl, [Online; accessed 21-October-2020].

[9] Data center interconnection fabric for edge cloud., *Data center interconnection fabric for edge cloud.* https://pf.content.nokia.com/t004h1-webscale-data-center-networking/dci-fabric-white-paper?lx=5Gr64m, [Online; accessed 21-October-2020].

[10] M. Simsek, A. Aijaz, M. Dohler, J. Sachs, and G. Fettweis, *5g-enabled tactile internet,* IEEE Journal on Selected Areas in Communications **34**, 460 (2016).

[11] M. Maier, M. Chowdhury, B. P. Rimal, and D. P. Van, *The tactile internet: vision, recent progress, and open challenges,* IEEE Communications Magazine **54**, 138 (2016).

[12] A. Aijaz, M. Dohler, A. H. Aghvami, V. Friderikos, and M. Frodigh, *Realizing the tactile internet: Haptic communications over next generation 5g cellular networks,* IEEE Wireless Communications **24**, 82 (2016).

[13] K. Antonakoglou, X. Xu, E. Steinbach, T. Mahmoodi, and M. Dohler, *Toward haptic communications over the 5g tactile internet,* IEEE Communications Surveys & Tutorials **20**, 3034 (2018).

[14] G. Fettweis, H. Boche, T. Wiegand, E. Zielinski, H. Schotten, P. Merz, S. Hirche, A. Festag, W. Häffner, M. Meyer, *et al.*, *The tactile internet-itu-t technology watch report,* Int. Telecom. Union (ITU), Geneva (2014).

[15] J. Leigh, T. A. DeFanti, A. Johnson, M. Brown, and D. Sandin, *Global tele-immersion: Better than being there,* in *Proceedings of ICAT*, Vol. 97 (1997) pp. 3–5.

[16] J. J. LaViola Jr, *A discussion of cybersickness in virtual environments,* ACM Sigchi Bulletin **32**, 47 (2000).

[17] H. Farhangi, *The path of the smart grid,* IEEE Power and Energy Magazine **8**, 18 (2010).

[18] J. Deshpande, A. Locke, and M. Madden, *Smart choices for the smart grid: Using wireless broadband for power grid network transformation,* Alcatel-Lucent, technology white paper **19**, 20 (2010).

[19] M. Elgenedy, A. Massoud, and S. Ahmed, *Smart grid self-healing: Functions, applications, and developments,* in *2015 First Workshop on Smart Grid and Renewable Energy (SGRE)* (IEEE, 2015) pp. 1–6.

[20] 5G Network Slicing Enabling the Smart Grid., *5G Network Slicing Enabling the Smart Grid.* http://www-file.huawei.com/-/media/CORPORATE/PDF/News/5g-network-slicing-enabling-the-smart-grid.pdf, [Online; accessed 22-October-2020].

[21] USE CASES FOR THE ADOPTION OF 5G TELECOMMUNICATIONS WITHIN THE OPERATIONS OF ELECTRIC UTILITIES., *USE CASES FOR THE ADOPTION OF 5G TELECOMMUNICATIONS WITHIN THE OPERATIONS OF ELECTRIC UTILITIES.* https://eutc.org/wp-content/uploads/2019/04/EUTC-5G-USE-CASES-31102017.pdf, [Online; accessed 22-October-2020].

[22] Game on! How broadband providers can monetize ultra-low latency services for gamers., *Game on! How broadband providers can monetize ultra-low latency services for gamers.* https://www.nokia.com/blog/game-on-how-broadband-providers-can-monetize-ultra-low-latency-services-f [Online; accessed 22-October-2020].

[23] Why gaming is a promising 5G market., *Why gaming is a promising 5G market.* https://www.lightreading.com/5g/why-gaming-is-a-promising-5g-market/a/d-id/756465/, [Online; accessed 22-October-2020].

[24] Google Stadia - Welkom bij Stadia - Klik en speel meteen., *Google Stadia - Welkom bij Stadia - Klik en speel meteen.* [Online; accessed 24-October-2020].

[25] Find out the 5 things you need to know about 5G if you're a gamer., *Find out the 5 things you need to know about 5G if you're a gamer.* https://www.ericsson.com/en/5g/what-is-5g/5-things-to-know-about-5g-if-you-are-a-gamer, [Online; accessed 22-October-2020].

[26] R.-S. Schmoll, S. Pandi, P. J. Braun, and F. H. Fitzek, *Demonstration of vr/ar offloading to mobile edge cloud for low latency 5g gaming application,* in *2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC)* (IEEE, 2018) pp. 1–3.

[27] B. Turkovic, F. Kuipers, N. van Adrichem, and K. Langendoen, *Fast network congestion detection and avoidance using p4,* in *Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies* (2018) pp. 45–51.

[28] V. Jacobson, *Congestion avoidance and control,* ACM SIGCOMM computer communication review **18**, 314 (1988).

[29] J. Postel, *Transmission control protocol specification,* RFC 793 (1981).

[30] M. Allman, V. Paxson, and E. Blanton, *TCP Congestion Control,* RFC 5681 (Draft Standard) (2009).

[31] A. Narayan, F. Cangialosi, P. Goyal, S. Narayana, M. Alizadeh, and H. Balakrishnan, *The case for moving congestion control out of the datapath,* in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (ACM, 2017) pp. 101–107.

[32] M. Hock, F. Neumeister, M. Zitterbart, and R. Bless, *TCP LoLa: Congestion Control for Low Latencies and High Throughput,* in *2017 IEEE 42nd Conference on Local Computer Networks (LCN)* (2017) pp. 215–218.

[33] R. Mittal, V. T. Lam, N. Dukkipati, E. Blem, H. Wassel, M. Ghobadi, A. Vahdat, Y. Wang, D. Wetherall, and D. Zats, *TIMELY: RTT-based Congestion Control for the Datacenter,* (2015).

[34] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, *Bbr: Congestion-based congestion control,* Queue **14**, 20 (2016).

[35] T. Flach, N. Dukkipati, A. Terzis, B. Raghavan, N. Cardwell, Y. Cheng, A. Jain, S. Hao, E. Katz-Bassett, and R. Govindan, *Reducing web latency: the virtue of gentle aggression,* in *ACM SIGCOMM Computer Communication Review*, Vol. 43 (ACM, 2013) pp. 159–170.

[36] S. Floyd, *HighSpeed TCP for large congestion windows*, Tech. Rep. (2003).

[37] D. Leith and R. Shorten, *H-TCP: TCP for high-speed and long-distance networks,* in *Proceedings of PFLDnet*, Vol. 2004 (2004).

[38] R. Mittal, J. Sherry, S. Ratnasamy, and S. Shenker, *Recursively cautious congestion control,* in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)* (2014) pp. 373–385.

[39] C. Raiciu, M. Handley, and D. Wischik, *Coupled congestion control for multipath transport protocols*, Tech. Rep. (IETF RFC 6356, Oct, 2011).

[40] R. Khalili, N. Gast, M. Popovic, and J. yves Le Boudec, *Opportunistic linked-increases congestion control algorithm for mptcp,* (2013).

[41] A. Walid, Q. Peng, J. Hwang, and S. Low, *Balanced linked adaptation congestion control algorithm for mptcp,* Working Draft, IETF Secretariat, Internet-Draft draft-walid-mptcp-congestion-control-04 (2016).

[42] T. Kelly, *Scalable TCP: Improving performance in highspeed wide area networks,* ACM SIGCOMM computer communication Review **33**, 83 (2003).

[43] M. Xu, Y. Cao, and E. Dong, *Delay-based congestion control for mptcp,* IETF, work in progress, Internet-draft draft-xu-mptcpcongestion-control-01 (2015).

[44] D. Kliazovich, F. Granelli, and D. Miorandi, *Logarithmic window increase for TCP Westwood+ for improvement in high speed, long distance networks,* Computer Networks **52**, 2395 (2008).

[45] M. Dong, Q. Li, D. Zarchy, P. B. Godfrey, and M. Schapira, {*PCC*}*: Re-architecting congestion control for consistent high performance,* in *12th* {*USENIX*} *Symposium on Networked Systems Design and Implementation ({NSDI} 15)* (2015) pp. 395–408.

[46] M. Dong, T. Meng, D. Zarchy, E. Arslan, Y. Gilad, B. Godfrey, and M. Schapira, {*PCC*} *vivace: Online-learning congestion control,* in *15th* {*USENIX*} *Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (2018) pp. 343–356.

[47] K. Winstein and H. Balakrishnan, *Tcp ex machina: Computer-generated congestion control,* ACM SIGCOMM Computer Communication Review **43**, 123 (2013).

[48] S. Ha, I. Rhee, and L. Xu, *CUBIC: a new TCP-friendly high-speed TCP variant,* SIGOPS Oper. Syst. Rev. **42**, 64 (2008).

[49] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang, *TCP westwood: Bandwidth estimation for enhanced transport over wireless links,* in *Proceedings of the 7th annual international conference on Mobile computing and networking* (ACM, 2001) pp. 287–297.

[50] L. A. Grieco and S. Mascolo, *Performance evaluation and comparison of Westwood+, New Reno, and Vegas TCP congestion control,* ACM SIGCOMM Computer Communication Review **34**, 25 (2004).

[51] K. Yamada, R. Wang, M. Y. Sanadidi, and M. Gerla, *TCP westwood with agile probing: dealing with dynamic, large, leaky pipes,* in *2004 IEEE International Conference on Communications (IEEE Cat. No.04CH37577)*, Vol. 2 (2004) pp. 1070–1074 Vol.2.

[52] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan, *An experimental study of the learnability of congestion control,* ACM SIGCOMM Computer Communication Review **44**, 479 (2014).

[53] K. Winstein, A. Sivaraman, and H. Balakrishnan, *Stochastic forecasts achieve high throughput and low delay over cellular networks,* in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)* (2013) pp. 459–471.

[54] M. T. Arashloo, A. Lavrov, M. Ghobadi, J. Rexford, D. Walker, and D. Wentzlaff, *Enabling programmable transport protocols in high-speed nics,* in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)* (2020) pp. 93–109.

[55] M. T. Arashloo, M. Ghobadi, J. Rexford, and D. Walker, *Hotcocoa: Hardware congestion control abstractions,* in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks* (2017) pp. 108–114.

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

[57] C. Caini and R. Firrincieli, *TCP Hybla: a TCP enhancement for heterogeneous networks,* International journal of satellite communications and networking **22**, 547 (2004).

[58] J. Jiang and Y. Zhang, *An accurate congestion control mechanism in programmable network,* in *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)* (IEEE, 2019) pp. 0673–0677.

[59] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, *et al.*, *The quic transport protocol: Design and internet-scale deployment,* in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (2017) pp. 183–196.

[60] Q. De Coninck, F. Michel, M. Piraux, F. Rochet, T. Given-Wilson, A. Legay, O. Pereira, and O. Bonaventure, *Pluginizing quic,* in *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19 (Association for Computing Machinery, New York, NY, USA, 2019) p. 59–74.

[61] T. Viernickel, A. Froemmgen, A. Rizk, B. Koldehofe, and R. Steinmetz, *Multipath quic: A deployable multipath transport protocol,* in *2018 IEEE International Conference on Communications (ICC)* (2018) pp. 1–7.

[62] S. Floyd and V. Jacobson, *Random early detection gateways for congestion avoidance,* IEEE/ACM Transactions on networking , 397 (1993).

[63] S. Floyd, R. Gummadi, S. Shenker, *et al.*, *Adaptive red: An algorithm for increasing the robustness of red's active queue management,* (2001).

[64] T. J. Ott, T. Lakshman, and L. H. Wong, *Sred: stabilized red,* in *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint*

Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320), Vol. 3 (IEEE, 1999) pp. 1346–1355.

[65] D. Lin and R. Morris, *Dynamics of random early detection,* SIGCOMM Comput. Commun. Rev. **27**, 127–137 (1997).

[66] S. Athuraliya, S. H. Low, V. H. Li, and Q. Yin, *Rem: Active queue management,* IEEE network **15**, 48 (2001).

[67] R. Pan, B. Prabhakar, and K. Psounis, *Choke-a stateless active queue management scheme for approximating fair bandwidth allocation,* in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, Vol. 2 (IEEE, 2000) pp. 942–951.

[68] S. S. Kunniyur and R. Srikant, *An adaptive virtual queue (avq) algorithm for active queue management,* IEEE/ACM Transactions on Networking (ToN) **12**, 286 (2004).

[69] J. Sun and M. Zukerman, *An adaptive neuron aqm for a stable internet,* in *International conference on research in networking* (Springer, 2007) pp. 844–854.

[70] F. Ren, C. Lin, and B. Wei, *A robust active queue management algorithm in large delay networks,* Computer communications **28**, 485 (2005).

[71] K. Nichols and V. Jacobson, *Controlling queue delay,* Queue **10**, 20 (2012).

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

[73] W.-c. Feng, D. Kandlur, D. Saha, and K. Shin, *BLUE: A new class of active queue management algorithms*, Tech. Rep. (Technical Report CSE-TR-387-99, University of Michigan, 1999).

[74] M. Li and H. Wang, *Study of active queue management algorithms—-towards stabilize and high link utilization,* .

[75] E. Dumazet, *net_sched: sfq: Optional RED on top of SFQ,* https://www.spinics.net/lists/netdev/msg185147.html (2012), [Online; accessed 15-August-2019].

[76] T. Hoeiland-Joergensen, P. McKenney, D. Taht, J. Gettys, and E. Dumazet, *The flow queue codel packet scheduler and active queue management algorithm,* RFC8290 [Online]. Available: https://tools. ietf. org/html/rfc8290 (2018).

[77] B. Turkovic and F. Kuipers, *P4air: Increasing fairness among competing congestion control algorithms,* in *2020 IEEE 28th International Conference on Network Protocols (ICNP)* (IEEE, 2020) pp. 1–12.

[78] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, *Openflow: Enabling innovation in campus networks,* SIGCOMM Comput. Commun. Rev. **38**, 69 (2008).

[79] S. Sezer, S. Scott-Hayward, P. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, *Are we ready for sdn? implementation challenges for software-defined networks,* IEEE Communications Magazine **51**, 36 (2013).

[80] H. E. Egilmez, S. T. Dane, K. T. Bagci, and A. M. Tekalp, *Openqos: An openflow controller design for multimedia delivery with end-to-end quality of service over software-defined networks,* in *Proceedings of The 2012 Asia Pacific Signal and Information Processing Association Annual Summit and Conference* (2012) pp. 1–8.

[81] J. W. Guck, A. V. Bemten, M. Reisslein, and W. Kellerer, *Unicast qos routing algorithms for sdn: A comprehensive survey and performance evaluation,* IEEE Communications Surveys Tutorials **PP**, 1 (2017).

[82] S. Civanlar, M. Parlakisik, A. M. Tekalp, B. Gorkemli, B. Kaytaz, and E. Onem, *A qos-enabled openflow environment for scalable video streaming,* in *2010 IEEE Globecom Workshops* (2010) pp. 351–356.

[83] D. Adami, L. Donatini, S. Giordano, and M. Pagano, *A network control application enabling software-defined quality of service,* in *2015 IEEE International Conference on Communications (ICC)* (2015) pp. 6074–6079.

[84] D. Adami, G. Antichi, R. G. Garroppo, S. Giordano, and A. W. Moore, *Towards an sdn network control application for differentiated traffic routing,* in *2015 IEEE International Conference on Communications (ICC)* (2015) pp. 5827–5832.

[85] A. Ishimori, F. Farias, E. Cerqueira, and A. Abelém, *Control of multiple packet schedulers for improving qos on openflow/sdn networking,* in *2013 Second European Workshop on Software Defined Networks* (2013) pp. 81–86.

[86] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba, *Policycop: An autonomic qos policy enforcement framework for software defined networks,* in *2013 IEEE SDN for Future Networks and Services (SDN4FNS)* (2013) pp. 1–7.

[87] N. L. M. van Adrichem, C. Doerr, and F. A. Kuipers, *Opennetmon: Network monitoring in openflow software-defined networks,* in *2014 IEEE Network Operations and Management Symposium (NOMS)* (2014) pp. 1–8.

[88] Z. Shu, J. Wan, J. Lin, S. Wang, D. Li, S. Rho, and C. Yang, *Traffic engineering in software-defined networking: Measurement and management,* IEEE Access **4**, 3246 (2016).

[89] *Streaming telemetry,* `http://www.openconfig.net/projects/telemetry/` (2016), accessed: 20-02-2022.

[90] S. Sharma, D. Staessens, D. Colle, D. Palma, J. Gonçalves, R. Figueiredo, D. Morris, M. Pickavet, and P. Demeester, *Implementing quality of service for the software defined networking enabled future internet,* in *2014 Third European Workshop on Software Defined Networks* (2014) pp. 49–54.

[91] H. Krishna, N. L. M. van Adrichem, and F. A. Kuipers, *Providing bandwidth guarantees with openflow,* in *2016 Symposium on Communications and Vehicular Technologies (SCVT)* (2016) pp. 1–6.

[92] F. Li, J. Cao, X. Wang, Y. Sun, and Y. Sahni, *Enabling software defined networking with qos guarantee for cloud applications,* in *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)* (2017) pp. 130–137.

[93] S. Tomovic, N. Prasad, and I. Radusinovic, *Sdn control framework for qos provisioning,* in *2014 22nd Telecommunications Forum Telfor (TELFOR)* (2014) pp. 111–114.

[94] R. C. R. Wallner, *An sdn approach: Quality of service using big switchs floodlight open-source controller, Asia-Pacific Advanced Network (APAN)*, Proceedings of the Asia-Pacific Advanced Network (2013).

[95] S. Dwarakanathan, L. Bass, and L. Zhu, *Cloud application ha using sdn to ensure qos,* in *2015 IEEE 8th International Conference on Cloud Computing* (2015) pp. 1003–1007.

[96] J. Guck, M. Reisslein, and W. Kellerer, *Function split between delay-constrained routing and resource allocation for centrally managed qos in industrial networks, IEEE Transactions on Industrial Informatics*, **12**, 1 (2016).

[97] The Technology Book: The Technology trends KPN has on its radar., *The Technology Book: The Technology trends KPN has on its radar.* https://www.kivi.nl/uploads/media/5a9539b06a1dc/1474897928_KPN_Technology_Book_2016.pdf, [Online; accessed 21-October-2020].

[98] Intel® Tofino™: P4-programmable Ethernet switch ASIC that delivers better performance at lower power, *Intel® Tofino™,* https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html, [Online; accessed 03-November-2020].

[99] C. Kim, P. Bhide, E. Doe, H. Holbrook, A. Ghanwani, D. Daly, and B. Hira, Mukesh amd Davie, *In-band network telemetry (int),* (2016), https://p4.org/assets/INT-current-spec.pdf, Last accessed on 10-06-2020.

[100] B. Turkovic, J. Oostenbrink, F. Kuipers, I. Keslassy, and A. Orda, *Sequential zeroing: Online heavy-hitter detection on programmable hardware,* in *2020 IFIP Networking Conference (Networking)* (IEEE, 2020) pp. 422–430.

[101] R. B. Basat, G. Einziger, I. Keslassy, A. Orda, S. Vargaftik, and E. Waisbard, *Memento: Making sliding windows efficient for heavy hitters,* arXiv preprint arXiv:1810.02899 (2018).

[102] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, *Heavy-hitter detection entirely in the data plane,* ACM SOSR , 164 (2017).

[103] R. Ben-Basat, G. Einziger, R. Friedman, and Y. Kassner, *Heavy hitters in streams and sliding windows.* in *INFOCOM* (2016) pp. 1–9.

[104] R. Ben-Basat, X. Chen, G. Einziger, and O. Rottenstreich, *Efficient measurement on programmable switches using probabilistic recirculation,* in *2018 IEEE 26th International Conference on Network Protocols (ICNP)* (IEEE, 2018) pp. 313–323.

[105] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, *P4: Programming protocol-independent packet processors,* SIGCOMM Comput. Commun. Rev. **44**, 87 (2014).

[106] B. Turkovic, F. Kuipers, N. van Adrichem, and K. Langendoen, *Fast network congestion detection and avoidance using p4,* in *CoNEXT*, NEAT '18 (ACM, New York, NY, USA, 2018) pp. 45–51.

[107] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, *Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn,* ACM SIGCOMM CCR **43**, 99 (2013).

[108] R. Bifulco and G. Rétvári, *A survey on the programmable data plane: Abstractions architectures and open problems,* in *Proc. IEEE HPSR* (2018) pp. 1–7.

[109] About Agilio SmartNICs, *About Agilio SmartNICs,* https://www.netronome.com/products/smartnic/overview/, [Online; accessed 16-January-2020].

[110] A. Metwally, D. Agrawal, and A. El Abbadi, *Efficient computation of frequent and top-k elements in data streams,* in *International Conference on Database Theory* (Springer, 2005) pp. 398–412.

[111] G. Cormode and S. Muthukrishnan, *An improved data stream summary: The count-min sketch and its applications,* J. Algorithms **55**, 58 (2005).

[112] X. Wu and Y. Luo, *Network Measurement with P4 and C on Netronome Agilio,* Available at https://www.slideshare.net/Open-NFP/network-measurement-with-p4-and-c-on-netronome-agilio.

[113] S. Muthukrishnan *et al., Data streams: Algorithms and applications,* Foundations and Trends® in Theoretical Computer Science **1**, 117 (2005).

[114] Q. Rong, G. Zhang, G. Xie, and K. Salamatian, *Mnemonic lossy counting: An efficient and accurate heavy-hitters identification algorithm,* in *IEEE IPCCC* (2010) pp. 255–262.

[115] The CAIDA UCSD Anonymized Internet Traces - 2016, 2018, *The CAIDA UCSD anonymized internet traces - 2016, 2018,* Available at http://www.caida.org/data/passive/passive_dataset.xml.

[116] T. Benson, A. Akella, and D. A. Maltz, *Network traffic characteristics of data centers in the wild,* in *ACM IMC '10* (2010) pp. 267–280.

[117] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, *Heavykeeper: An accurate algorithm for finding top-k elephant flows,* IEEE/ACM Transactions on Networking **27**, 1845 (2019).

[118] B. Claise, *Cisco systems NetFlow services export version 9,* Tech. Rep. 2070-1721 (Internet Engineering Task Force, 2004).

[119] P. Phaal, S. Panchen, and N. McKee, *InMon corporation's sFlow: A method for monitoring traffic in switched and routed networks,* Tech. Rep. 2070-1721 (Internet Engineering Task Force, 2001).

[120] C. Estan and G. Varghese, *New directions in traffic measurement and accounting,* SIGCOMM Comput. Commun. Rev. **32**, 323 (2002).

[121] Y. Zhou, T. Yang, J. Jiang, B. Cui, M. Yu, X. Li, and S. Uhlig, *Cold filter: A meta-framework for faster and more accurate stream processing,* in *ACM SIGMOD* (2018) pp. 741–756.

[122] Z. Liu, A. Manousis, G. Vorsanger, V. Sekar, and V. Braverman, *One sketch to rule them all: Rethinking network flow monitoring with univmon,* in *Proceedings of the 2016 ACM SIGCOMM Conference* (ACM, 2016) pp. 101–114.

[123] M. Charikar, K. Chen, and M. Farach-Colton, *Finding frequent items in data streams,* in *International Colloquium on Automata, Languages, and Programming* (Springer, 2002) pp. 693–703.

[124] X. Dimitropoulos, P. Hurley, and A. Kind, *Probabilistic lossy counting: an efficient algorithm for finding heavy hitters,* SIGCOMM Comput. Commun. Rev. **38**, 5 (2008).

[125] X. Yu, H. Xu, D. Yao, H. Wang, and L. Huang, *Countmax: A lightweight and cooperative sketch measurement for software-defined networks,* IEEE/ACM Transactions on Networking (2018).

[126] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, *Elastic sketch: Adaptive and fast network-wide measurements,* in *ACM SIGCOMM* (2018) pp. 561–575.

[127] G. Einziger and R. Friedman, *Counting with tinytable: Every bit counts!* in *ICDCN* (2016).

[128] E. Assaf, R. Ben-Basat, G. Einziger, and R. Friedman, *Pay for a sliding bloom filter and get counting, distinct elements, and entropy for free,* CoRR **abs/1712.01779** (2017), arXiv:1712.01779 .

[129] R. B. Basat, G. Einziger, R. Friedman, and Y. Kassner, *Poster abstract: A sliding counting bloom filter,* in *INFOCOM WKSHPS* (2017) pp. 1012–1013.

[130] K. Polachan, B. Turkovic, T. Prabhakar, C. Singh, and F. A. Kuipers, *Dynamic network slicing for the tactile internet,* in *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)* (IEEE, 2020) pp. 129–140.

[131] A. Seam, A. Poll, R. Wright, J. Mueller, and F. Hoodbhoy, *Enabling mobile augmented and virtual reality with 5g networks,* (2017).

[132] G. Fettweis and S. Alamouti, *5g: Personal mobile internet beyond what cellular did to telephony,* IEEE Communications Magazine **52**, 140 (2014).

[133] Y. Gao, S. S. Vedula, C. E. Reiley, N. Ahmidi, B. Varadarajan, H. C. Lin, L. Tao, L. Zappella, B. Bejar, D. D. Yuh, C. C. G. Chen, R. Vidal, S. Khudanpur, and G. D. Hager, *Jhu isi gesture and skill assessment working set (jigsaws) a surgical activity dataset for human motion modeling,* Modeling and Monitoring of Computer Assisted Interventions (M2CAI) – MICCAI Workshop , 1 (2014).

[134] ITU, *The tactile internet,* (2014).

[135] J. J. LaViola, Jr., *A Discussion of Cybersickness in Virtual Environments,* SIGCHI Bull. **32**, 47 (2000).

[136] K. Polachan, T. V. Prabhakar, C. Singh, and D. Panchapakesan, *Quality of control assessment for tactile cyber-physical systems,* in *2019 IEEE International Conference on Sensing, Communication and Networking (IEEE SECON)* (2019).

[137] J. Marescaux, J. Leroy, F. Rubino, M. Smith, M. Vix, M. Simone, and D. Mutter, *Transcontinental robot-assisted remote telesurgery: feasibility and potential applications,* Annals of surgery **235**, 487 (2002).

[138] L. B. Valdez, R. R. Datta, B. Babic, D. T. Müller, C. J. Bruns, and H. F. Fuchs, *5g mobile communication applications for surgery: An overview of the latest literature,* Artificial Intelligence in Gastrointestinal Endoscopy **2**, 1 (2021).

[139] G. P. Fettweis, *The tactile internet: Applications and challenges,* IEEE Vehicular Technology Magazine **9**, 64 (2014).

[140] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, *P4: Programming protocol-independent packet processors,* SIGCOMM Comput. Commun. Rev. **44**, 87 (2014).

[141] *P4 behavioral model,* https://github.com/p4lang/behavioral-model, accessed: 19-03-2018.

[142] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, *Software-defined networking: A comprehensive survey,* Proceedings of the IEEE **103**, 14 (2015).

[143] H. Zhang, N. Liu, X. Chu, K. Long, A.-H. Aghvami, and V. C. Leung, *Network slicing based 5g and future mobile networks: mobility, resource management, and challenges,* IEEE Communications Magazine **55**, 138 (2017).

[144] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, *Network slicing in 5g: Survey and challenges,* IEEE Communications Magazine **55**, 94 (2017).

[145] S. Vassilaras, L. Gkatzikis, N. Liakopoulos, I. N. Stiakogiannakis, M. Qi, L. Shi, L. Liu, M. Debbah, and G. S. Paschos, *The algorithmic aspects of network slicing,* IEEE Communications Magazine **55**, 112 (2017).

[146] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, *Flowvisor: A network virtualization layer,* OpenFlow Switch Consortium, Tech. Rep **1**, 132 (2009).

[147] D. Drutskoy, E. Keller, and J. Rexford, *Scalable network virtualization in software-defined networks,* IEEE Internet Computing **17**, 20 (2012).

[148] T. Mizrahi and Y. Moses, *Time4: Time for sdn,* IEEE Transactions on Network and Service Management **13**, 433 (2016).

[149] P. Rost, C. Mannweiler, D. S. Michalopoulos, C. Sartori, V. Sciancalepore, N. Sastry, O. Holland, S. Tayade, B. Han, D. Bega, D. Aziz, and H. Bakker, *Network slicing to enable scalability and flexibility in 5g mobile networks,* IEEE Communications Magazine **55**, 72 (2017).

[150] M. Dohler, T. Mahmoodi, M. A. Lema, M. Condoluci, F. Sardis, K. Antonakoglou, and H. Aghvami, *Internet of skills, where robotics meets ai, 5g and the tactile internet,* in *2017 European Conference on Networks and Communications (EuCNC)* (2017) pp. 1–5.

[151] M. Simsek, A. Aijaz, M. Dohler, J. Sachs, and G. Fettweis, *5g-enabled tactile internet,* IEEE Journal on Selected Areas in Communications **34**, 460 (2016).

[152] Z. Hou, C. She, Y. Li, T. Q. S. Quek, and B. Vucetic, *Burstiness-aware bandwidth reservation for ultra-reliable and low-latency communications in tactile internet,* IEEE Journal on Selected Areas in Communications **36**, 2401 (2018).

[153] M. Condoluci, T. Mahmoodi, E. Steinbach, and M. Dohler, *Soft resource reservation for low-delayed teleoperation over mobile networks,* IEEE Access **5**, 10445 (2017).

[154] F. Zhou, F. D. l. Torre, and J. K. Hodgins, *Hierarchical aligned cluster analysis for temporal clustering of human motion,* IEEE Transactions on Pattern Analysis and Machine Intelligence **35**, 582 (2013).

[155] L. Li and B. A. Prakash, *Time series clustering: Complex is simpler!* in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (2011) pp. 185–192.

[156] S. Li, K. Li, and Y. Fu, *Temporal subspace clustering for human motion segmentation,* in *Proceedings of the IEEE International Conference on Computer Vision* (2015) pp. 4453–4461.

[157] D. Baumann, F. Mager, R. Jacob, L. Thiele, M. Zimmerling, and S. Trimpe, *Fast feedback control over multi-hop wireless networks with mode changes and stability guarantees,* (2019), arXiv:1909.10873 [eess.SY] .

[158] B. Turkovic, S. Nijhuis, and F. Kuipers, *Elastic slicing in programmable networks,* in *NetSoft 2021-IEEE International Conference on Network Softwarization* (IEEE, 2021).

[159] A. Gember, P. Prabhu, Z. Ghadiyali, and A. Akella, *Toward software-defined middlebox networking,* in *Proceedings of the 11th ACM Workshop on Hot Topics in Networks* (2012) pp. 7–12.

[160] J. Ordonez-Lucena, P. Ameigeiras, D. Lopez, J. J. Ramos-Munoz, J. Lorca, and J. Folgueira, *Network slicing for 5g with sdn/nfv: Concepts, architectures, and challenges,* IEEE Communications Magazine **55**, 80 (2017).

[161] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, *Paving the way for {NFV}: Simplifying middlebox modifications using statealyzr,* in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)* (2016) pp. 239–253.

[162] S. Luo, H. Yu, and L. Vanbever, *Swing state: Consistent updates for stateful and programmable data planes,* (2017).

[163] C. Zhang, J. Bi, Y. Zhou, and J. Wu, *Hypervdp: High-performance virtualization of the programmable data plane,* IEEE Journal on Selected Areas in Communications **37**, 556 (2019).

[164] D. Hancock and J. Van der Merwe, *Hyper4: Using p4 to virtualize the programmable data plane,* in *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies* (ACM, 2016) pp. 35–49.

[165] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja, *Virtp4: An architecture for p4 virtualization,* in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (IEEE, 2019) pp. 75–78.

[166] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, *E2: a framework for nfv applications,* in *Proceedings of the 25th Symposium on Operating Systems Principles* (2015) pp. 121–136.

[167] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, *Flowtags: Enforcing network-wide policies in the presence of dynamic middlebox actions,* in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (2013).

[168] A. Gember, S. S. J. Anand Krishnamurthy, R. Grandl, X. Gao, A. Anand, T. Benson, A. Akella, and V. Sekar, *Stratos: A network-aware orchestration layer for middleboxes in the cloud. corr (2013),* (2013).

[169] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu, *Simple-fying middlebox policy enforcement using sdn,* in *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM* (2013) pp. 27–38.

[170] V. Sekar, N. Egi, S. Ratnasamy, M. K. Reiter, and G. Shi, *Design and implementation of a consolidated middlebox architecture,* in *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)* (2012) pp. 323–336.

[171] M. Kablan, A. Alsudais, E. Keller, and F. Le, *Stateless network functions: Breaking the tight coupling of state and processing,* in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)* (2017) pp. 97–112.

[172] A. Gember-Jacobson, R. Viswanathan, C. Prakash, R. Grandl, J. Khalid, S. Das, and A. Akella, *Opennf: Enabling innovation in network function control,* ACM SIGCOMM Computer Communication Review **44**, 163 (2014).

[173] S. Rajagopalan, D. Williams, H. Jamjoom, and A. Warfield, *Split/merge: System support for elastic execution in virtual middleboxes,* in *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)* (2013) pp. 227–240.

[174] S. Woo, J. Sherry, S. Han, S. Moon, S. Ratnasamy, and S. Shenker, *Elastic scaling of stateful network functions,* in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)* (2018) pp. 299–312.

[175] M. He, A. Basta, A. Blenk, N. Deric, and W. Kellerer, *P4nfv: An nfv architecture with flexible data plane reconfiguration,* in *2018 14th International Conference on Network and Service Management (CNSM)* (IEEE, 2018) pp. 90–98.

[176] G. Sviridov, M. Bonola, A. Tulumello, P. Giaccone, A. Bianco, and G. Bianchi, *Lodge: Local decisions on global states in prograananaable data planes,* in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)* (IEEE, 2018) pp. 257–261.

[177] M. T. Arashloo, Y. Koral, M. Greenberg, J. Rexford, and D. Walker, *Snap: Stateful network-wide abstractions for packet processing,* in *Proceedings of the 2016 ACM SIGCOMM Conference* (2016) pp. 29–43.

[178] L. Liu, H. Xu, Z. Niu, P. Wang, and D. Han, *U-haul: Efficient state migration in nfv,* in *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems* (2016) pp. 1–8.

[179] B. Turkovic, F. A. Kuipers, and S. Uhlig, *Interactions between congestion control algorithms,* in *2019 Network Traffic Measurement and Analysis Conference (TMA)* (IEEE, 2019) pp. 161–168.

[180] B. Turkovic, F. A. Kuipers, and S. Uhlig, *Fifty shades of congestion control: A performance and interactions evaluation,* arXiv preprint arXiv:1903.03852 (2019).

[181] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, *BBR: Congestion-Based Congestion Control,* Queue **14**, 20 (2016).

[182] D. Scholz, B. Jaeger, L. Schwaighofer, D. Raumer, F. Geyer, and G. Carle, *Towards a Deeper Understanding of TCP BBR Congestion Control,* in *IFIP Networking 2018* (Zurich, Switzerland, 2018).

[183] B. Turkovic, F. A. Kuipers, and S. Uhlig, *Fifty Shades of Congestion Control: A Performance and Interactions Evaluation,* ArXiv (2019).

[184] A. Afanasyev, N. Tilley, P. Reiher, and L. Kleinrock, *Host-to-host congestion control for TCP,* IEEE Communications surveys & tutorials **12**, 304 (2010).

[185] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, *TCP Vegas: New techniques for congestion detection and avoidance*, Vol. 24 (ACM, 1994).

[186] G. Hasegawa, K. Kurata, and M. Murata, *Analysis and improvement of fairness between TCP Reno and Vegas for deployment of TCP Vegas to the Internet,* in *Proceedings 2000 International Conference on Network Protocols* (2000) pp. 177–186.

[187] K. Srijith, L. Jacob, and A. L. Ananda, *Tcp vegas-a: Improving the performance of tcp vegas,* Computer communications **28**, 429 (2005).

[188] C. Jin, D. Wei, S. H. Low, J. Bunn, H. D. Choe, J. C. Doylle, H. Newman, S. Ravot, S. Singh, F. Paganini, G. Buhrmaster, L. Cottrell, O. Martin, and W. chun Feng, *FAST TCP: from theory to experiments,* IEEE Network **19**, 4 (2005).

[189] S. Belhaj and M. Tagina, *VFAST TCP: An improvement of FAST TCP,* in *Computer Modeling and Simulation, 2008. UKSIM 2008. Tenth International Conference on* (IEEE, 2008) pp. 88–93.

[190] J. Sing and B. Soh, *TCP New Vegas: improving the performance of TCP Vegas over high latency links,* in *Network Computing and Applications, Fourth IEEE International Symposium on* (IEEE, 2005) pp. 73–82.

[191] A. Kuzmanovic and E. W. Knightly, *Tcp-lp: low-priority service via end-point congestion control,* IEEE/ACM Transactions on Networking **14**, 739 (2006).

[192] A. Kuzmanovic and E. W. Knightly, *Tcp-lp: A distributed algorithm for low priority data transfer,* in *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, Vol. 3 (IEEE, 2003) pp. 1691–1701.

[193] S. Shalunov, G. Hazel, J. Iyengar, M. Kuehlewind, *et al.*, *Low extra delay background transport (ledbat),* IETF draft (2012).

[194] C. P. Fu and S. C. Liew, *TCP Veno: TCP enhancement for transmission over wireless access networks,* IEEE Journal on selected areas in communications **21**, 216 (2003).

[195] R. King, R. Baraniuk, and R. Riedi, *TCP-Africa: An adaptive and fair rapid increase rule for scalable TCP,* in *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, Vol. 3 (IEEE, 2005) pp. 1838–1848.

[196] K. Tan, J. Song, Q. Zhang, and M. Sridharan, *A Compound TCP Approach for High-Speed and Long Distance Networks,* in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications* (2006) pp. 1–12.

[197] A. Baiocchi, A. P. Castellani, and F. Vacirca, *YeAH-TCP: yet another highspeed TCP,* in *Proc. PFLDnet*, Vol. 7 (2007) pp. 37–42.

[198] S. Liu, T. Başar, and R. Srikant, *TCP-Illinois: A loss-and delay-based congestion control algorithm for high-speed networks,* Performance Evaluation **65**, 417 (2008).

[199] H. Shimonishi and T. Murase, *Improving efficiency-friendliness tradeoffs of TCP congestion control algorithm,* in *Global Telecommunications Conference, 2005. GLOBECOM'05. IEEE*, Vol. 1 (IEEE, 2005) pp. 5–pp.

[200] K. Kaneko, T. Fujikawa, Z. Su, and J. Katto, *TCP-Fusion: a hybrid congestion control algorithm for high-speed networks,* in *Proc. PFLDnet*, Vol. 7 (2007) pp. 31–36.

[201] H. Shimonishi, T. Hama, and T. Murase, *TCP-adaptive reno for improving efficiency-friendliness tradeoffs of TCP congestion control algorithm,* in *Proc. PFLDnet* (Citeseer, 2006) pp. 87–91.

[202] G. Marfia, C. Palazzi, G. Pau, M. Gerla, M. Sanadidi, and M. Roccetti, *Tcp libra: Exploring rtt-fairness for tcp,* in *International Conference on Research in Networking* (Springer, 2007) pp. 1005–1013.

[203] M. Hock, R. Bless, and M. Zitterbart, *Experimental evaluation of BBR congestion control,* in *2017 IEEE 25th International Conference on Network Protocols (ICNP)* (2017) pp. 1–10.

[204] S. Ma, J. Jiang, W. Wang, and B. Li, *Towards RTT Fairness of Congestion-Based Congestion Control,* CoRR **abs/1706.09115** (2017), arXiv:1706.09115 .

[205] R. K. Jain, D.-M. W. Chiu, and W. R. Hawe, *A Quantitative Measure of Fairness and Discrimination,* Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA (1984).

[206] *The chromium projects: Chromium,* https://www.chromium.org/Home, accessed: 04-03-2019.

[207] M. Hock, R. Bless, and M. Zitterbart, *Experimental evaluation of bbr congestion control,* in *2017 IEEE 25th International Conference on Network Protocols (ICNP)* (IEEE, 2017) pp. 1–10.

[208] T. Kozu, Y. Akiyama, and S. Yamaguchi, *Improving rtt fairness on cubic tcp,* in *2013 First International Symposium on Computing and Networking* (2013) pp. 162–167.

[209] R. Al-Saadi, G. Armitage, J. But, and P. Branch, *A survey of delay-based and hybrid tcp congestion control algorithms,* IEEE Communications Surveys & Tutorials **21**, 3609 (2019).

[210] D. Scholz, B. Jaeger, L. Schwaighofer, D. Raumer, F. Geyer, and G. Carle, *Towards a deeper understanding of tcp bbr congestion control,* in *2018 IFIP Networking Conference (IFIP Networking) and Workshops* (IEEE, 2018) pp. 1–9.

[211] N. Cardwell, Y. Cheng, S. H. Yeganeh, and V. Jacobson, *BBR congestion control,* Working Draft, IETF Secretariat, Internet-Draft draft-card-well-iccrg-bbr-congestion-control-00 (2017).

[212] Y. Cao, A. Jain, K. Sharma, A. Balasubramanian, and A. Gandhi, *When to use and when not to use bbr: An empirical analysis and evaluation study,* in *Proceedings of the Internet Measurement Conference* (2019) pp. 130–136.

[213] S. Ma, J. Jiang, W. Wang, and B. Li, *Towards rtt fairness of congestion-based congestion control,* CoRR (2017).

[214] G. Hasegawa, K. Kurata, and M. Murata, *Analysis and improvement of fairness between tcp reno and vegas for deployment of tcp vegas to the internet,* in *Proceedings 2000 International Conference on Network Protocols* (IEEE, 2000) pp. 177–186.

[215] J. Lin, L. Cui, Y. Zhang, F. P. Tso, and Q. Guan, *Extensive evaluation on the performance and behaviour of tcp congestion control protocols under varied network scenarios,* Computer Networks **163**, 106872 (2019).

[216] G. White and D. Rice, *Active queue management in docsis 3. x cable modems,* Technical report, CableLabs (2014).

[217] I. Järvinen and M. Kojo, *Evaluating codel, pie, and hred aqm techniques with load transients,* in *39th Annual IEEE Conference on Local Computer Networks* (IEEE, 2014) pp. 159–167.

[218] N. Yamanaka, *High-Performance Backbone Network Technology* (CRC Press, 2004).

[219] F. Schwarzkopf, S. Veith, and M. Menth, *Performance analysis of codel and pie for saturated tcp sources,* in *2016 28th International Teletraffic Congress (ITC 28)*, Vol. 1 (IEEE, 2016) pp. 175–183.

[220] T. B. community, *Feature Rich Flow Monitoring with. P4,* Available at https://www.netronome.com/media/documents/WBN-2017-11-1-Penn-Feature-Rich-Flow-Monitoring-OpenNFP_.pdf.

[221] J. Postel, L. Garlick, and R. Rom, *Transmission control protocol specification,* DARPA Internet Request for Comments **793** (1981).

[222] S. Ha, I. Rhee, and L. Xu, *Cubic: a new tcp-friendly high-speed tcp variant,* ACM SIGOPS operating systems review **42**, 64 (2008).

[223] G. Montenegro and D. Havey, *Top 10 networking features in windows server 2019,* https://techcommunity.microsoft.com/t5/networking-blog/top-10-networking-features-in-windows-server-2019-8-a-faster/ba-p/339749, accessed: 23-3-2020.

[224] S. Ma, J. Jiang, W. Wang, and B. Li, *Fairness of congestion-based congestion control: Experimental evaluation and analysis,* arXiv preprint arXiv:1706.09115 (2017).

[225] B. Turkovic, F. A. Kuipers, and S. Uhlig, *Interactions between congestion control algorithms,* in *2019 Network Traffic Measurement and Analysis Conference (TMA)* (IEEE, 2019) pp. 161–168.

[226] K. Dooley and I. Brown, *Cisco IOS cookbook: Field-tested solutions to Cisco router problems* (" O'Reilly Media, Inc.", 2006).

[227] X. Chen, H. Kim, J. M. Aman, W. Chang, M. Lee, and J. Rexford, *Measuring tcp round-trip time in the data plane,* in *Proceedings of the Workshop on Secure Programmable Network Infrastructure* (2020) pp. 35–41.

[228] V. Firoiu and M. Borden, *A study of active queue management for congestion control,* 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 (IEEE, 2000) pp. 1435–1444.

[229] W.-C. Feng, D. D. Kandlur, D. Saha, and K. G. Shin, *A self-configuring red gateway,* in *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, Vol. 3 (IEEE, 1999) pp. 1320–1328.

[230] S. H. Low, F. Paganini, and J. C. Doyle, *Internet congestion control,* IEEE control systems magazine **22**, 28 (2002).

[231] B. Cronkite-Ratcliff, A. Bergman, S. Vargaftik, M. Ravi, N. McKeown, I. Abraham, and I. Keslassy, *Virtualized congestion control,* in *Proceedings of the 2016 ACM SIGCOMM Conference* (2016) pp. 230–243.

[232] G. Fettweis, *The tactile internet: Applications & challenges,* IEEE Vehic. Tech. Mag. **9**, 64 (2014).

[233] D. Maxim and Y.-Q. Song, *Delay analysis of avb traffic in time-sensitive networks (tsn),* in *Proceedings of the 25th International Conference on Real-Time Networks and Systems* (ACM, 2017) pp. 18–27.

[234] *P4 14 language specification,* https://p4.org/p4-spec/p4-14/v1.0.4/tex/p4.pdf (), accessed: 19-03-2018.

[235] *P4 16 language specification,* https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html (), accessed: 19-03-2018.

ACKNOWLEDGEMENTS

I still remember the first time I read the vacancy for a Ph.D. position on the Tactile Internet and thought: "How great would it be if such applications would exist in the future! It would be awesome to contribute to this." Five years later, and looking back, it feels like an eternity has passed, and I am always surprised at how much I have changed. However, this journey would have never been possible without the support of many people.

My first and foremost gratitude goes to my promotors, Fernando and Koen. Fernando, you were always calm and, in my opinion, a perfect counter to my constant switching of discussion topics and ideas. Moreover, you have encouraged me to become more focused over time and taught me how to be a good researcher, question things, identify good problems and look at them from different perspectives. In addition, with your relaxed nature and pragmatic approaches to potential problems, I always felt that no problem is too difficult and, indeed, we were always able to find a suitable solution/approach. I would not be where I am without you. Thank you for recognizing the potential in me, but also for all the great discussions over the years and your patience. Likewise, Koen, precisely because your primary research focus was not networking, you always gave me priceless feedback and made me question the things that I otherwise would take for granted. I very much enjoyed all our monthly discussions, which helped me frame problems better and present solutions more clearly. Moreover, thank you for always being approachable and for the many fun off-topic conversation we had over the years. Your unique sense of humor always managed to break the divide between students and professors and, in my opinion, made it even more enjoyable to do a Ph.D. in your group.

Next, I would like to thank my colleagues from the ENS group. Adrian, Antonia, Anup, Amjad, Chenxing, Georgios, Fernando, Qing, Ioannis, James, Jasper, Jeroen, Jorik, Keyarash, Kees, Koen, Miguel, Minaksie, Mitra, Nikos, Przmek, Renan, Rens, Sezen, Stef, Talia, Vijay, Vineet, Vito, VP, Weizheng thank you all for making the ENS group a great place to spend the last few years in. My special thanks go to my fellow office mates: Jorik, Antonia, and Renan. Our office was always by far one of the loudest and best-informed ones about things ranging from the big political issues to the best frikandel. From the ENS group, I would also like to offer special thanks to my bouldering/gaming group: Jorik, Kees, Nikos, Vito, Jasper, Renan and Eric. We had many great discussions over the years: from critical and relevant Ph.D. discussions (e.g., tips and tricks on how to squeeze your paper to fit the page limit) to teaching me how to correctly pronounce "Hugo" and surprising discussions with the bar owners about their family history. Thank you for always being an extremely enjoyable group to hang out with.

Throughout my Ph.D., I had the pleasure of collaborating with many great researchers (Fernando, Isaac, Ariel, Jorik, Niels, Steve, Koen, Kurian, Antonia, Soovam, Abhishek, TV Prabhakar, Chandramani Singh), and I would like to thank them all for their contributions to this thesis. I always appreciated our discussions and feel that this is what re-

search is really about: Challenging each other's ideas and evolving them together to find a suitable approach to a problem. Your contributions are visible throughout this thesis and have improved the quality of my papers and helped me become a better researcher.

Next, I would like to thank Lolke Boonstra. Lolke, you not only provided me with a testbed for my experiments but also offered continuous support. Thank you for always finding the time, being interested in what I was trying to achieve, and always finding innovative out-of-the-box solutions to circumvent limitations.

I would also like to thank my 'new' colleagues, the Networks group at TNO. Working there during the past year made me realize that there will always be new, fun, and interesting problems to tackle and new things to learn. Especially, I would like to thank my managers, Annemieke and Dick. You always made sure I felt welcome and part of the team, even during the challenging corona times, and over the last year, you always encouraged me to finish my thesis.

Here, I would also like to mention my friends. Amra, Bojana, Dalila, Daniel, Ema, Naida, I always felt I could rely on you in times of need, even though some of you lived miles away. Medina, Merima and Elida, I appreciate all the moments we had in the Netherlands, and I am looking forward to the many new ones. I am truly privileged to have you all as friends.

My utmost gratitude goes to my family. Throughout my life, you always supported me, made me believe in myself, and encouraged me to try new things. Although it resulted in me moving far away across Europe, I believe our relationship stayed strong and will only grow stronger in the future. Mom and dad, I wouldn't be here without your continuous support and motivation. I would like to thank you for everything you have done for me, from my early years to now. Nedim, you are a great friend, and although we are opposites (although twins) in many ways, I believe we complement each other greatly. Mirza, thank you for being there when I needed you, especially with your relaxed and very often comic comments. With the two of you, I always feel that there is a solution to every problem and that things are never as bleak as they seem. Irena and Irma, I would like to thank you for all the moments we spent together and hope there will be many new ones in the future.

Finally, I would like to thank Vito for all his love, care, and support. Words can never be enough to express all my gratitude to you. You always found time for my problems (however minor) and made me feel heard and seen. I cannot imagine a more sincere, supporting, and loving person than you. I hope our love stays as strong as it is today in many years to come.

CURRICULUM VITÆ

Belma TURKOVIĆ

Belma Turkovic was born in Zenica, Bosnia and Herzegovina on 27th December 1991. After finishing high school, she started her Bachelor at the Faculty of Electrical Engineering, University of Sarajevo. There she developed an interest in communication networks and decided to pursue this direction further during her studies. During her Master's studies, she applied for an internship at Ericsson, where she gained further knowledge in network protocols and their design. She graduated her Master's and Bachelor as the best student in her year. Upon receiving her Master's degree at the Faculty of Electrical Engineering, University of Sarajevo, in September 2015, she worked in the telecom sector for a year. However, as a person always looking for a new challenge, she decided to pursue a Ph.D. in the area of networking at the Embedded and Networked Systems Group at the Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS), TU Delft.

Throughout her Ph.D., she was a member of TU Delft's Lab on Internet Science (LOIS). Her research mainly focused on developing mechanisms for programmable networks to enable support for low-latency application domains, which resulted in his thesis. Afterward, she joined TNO, Netherlands Organisation for Applied Scientific Research, where she is continuing her research in programmable networks, 5G, and cloud computing.

LIST OF PUBLICATIONS

10. **B. Turkovic**, S. Nijhus, F.A. Kuipers, *Elastic Network Slicing*, NetSoft 2021-IEEE International Conference on Network Softwarization(2021).

9. **B. Turkovic**, S. Biswal, A. Vijay, A. Hüfner F.A. Kuipers, *P4QoS: QoS-based Packet Processing with P4*, NetSoft 2021-IEEE International Conference on Network Softwarization (2021).

8. **B. Turkovic**, F.A. Kuipers, *P4air: Increasing Fairness among Competing Congestion Control Algorithms*, 28th IEEE International Conference on Network Protocols (ICNP), (2020).

7. **B. Turkovic**, J. Oostenbrink, F.A. Kuipers, I. Keslassy, A. Orda, *Sequential Zeroing: Online Heavy-Hitter Detection on Programmable Hardware*, 2020 IFIP Networking Conference (Networking), 422-430, (2020).

6. K. Polachan, **B. Turkovic**, T.V. Prabhakar, C. Singh, F.A. Kuipers, *Dynamic Network Slicing for the Tactile Internet*, 2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS), 129-140, (2020).

5. **B. Turkovic**, F.A. Kuipers, S. Uhlig, *Interactions between Congestion Control Algorithms*, 2019 Network Traffic Measurement and Analysis Conference (TMA), 161-168 (2019).

4. **B. Turkovic**, F.A. Kuipers, S. Uhlig, *Fifty shades of congestion control: A performance and interactions evaluation*, arXiv preprint arXiv:1903.03852, (2019).

3. **B Turkovic**, J. Oostenbrink, F.A. Kuipers, *Detecting heavy hitters in the data-plane*, arXiv preprint arXiv:1902.06993, (2019).

2. **B. Turkovic**, F.A. Kuipers, N. van Adrichem, K. Langendoen, *Ḟast network congestion detection and avoidance using P4*, Proceedings of the 2018 Workshop on Networking for Emerging Applications and Technologies, 45-51 (2018).

1. D. van den Berg, R. Glans, D. de Koning, F.A. Kuipers, J. Lugtenburg, K. Polachan, T.V. Prabhakar, C. Singh, **B. Turkovic**, B. van Wijk, *Challenges in haptic communications over the tactile internet*, IEEE Access Vol. 5, 23502-23518, (2017).