Technische Universität München
Fakultät für Elektrotechnik und Informationstechnik
Lehrstuhl für Kommunikationsnetze

Design, Implementation, and Evaluation of Mechanisms for Predictable Latency in Programmable Networks

Amaury Van Bemten, M.Sc.

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender:		Prof. Dr. Andreas Herkersdorf
Prüfer der Dissertation:	1.	Prof. Dr.-Ing. Wolfgang Kellerer
	2.	Prof. Dr. Laurent Vanbever

Die Dissertation wurde am 18.05.2020 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 21.09.2020 angenommen.

Design, Implementation, and Evaluation of Mechanisms for Predictable Latency in Programmable Networks

Amaury Van Bemten, M.Sc.

21.09.2020

Abstract

Communication networks form the backbone of our digital society, connecting users to data centers, data centers to each other, and sensors and actuators to automation controllers. This high connectivity enables the industry and public authorities to provide an ever growing plethora of services with systems such as the Internet of things, cyber-physical systems, smart cities, smart grids, cloud computing, and more generally, 5G networks. Ever since the deployment of the ARPANET and the first Internet node in 1969 at the University of California, Los Angeles, the Internet and communication technologies have grown and developed at an increasing pace. Originally designed for providing a simple best-effort connectivity, our modern digital society and its new applications and services now impose additional requirements for the underlying infrastructure. The quality of the service offered to users, customers, and tenants depends on the quality of service offered by the underlying communication infrastructure.

Generally, emerging applications require this underlying infrastructure to provide *predictability*, both from a correctness and from a performance point of view. Predictability is hard to achieve given the distributed nature of communication networks. As a result, the last decade has seen a shift from statically configured networks towards *programmable networks*, where the behavior of the network, and in particular of its constituting nodes, is not anymore governed by configuration files and an immutable logic, but rather by software that operates independently from the network hardware. Technologies such as OpenFlow and P4 enable network operators to remotely program the behavior of their network, thereby reducing costs and increasing flexibility, but most importantly allowing to cope with the heterogeneity and high variability of today's applications and communications. To which extent such technologies can provide the predictability required by modern applications is an ongoing research problem. This thesis investigates the particular problem of providing *predictable latency* to applications using programmable networks. Latency is a critical quality of service metric for applications, as guaranteed latency enables services to provide response time guarantees to their users, tenants, and customers. The focus of this thesis is on providing *strict* per-packet latency guarantees to users and applications. Providing predictability and strict determinism in distributed systems raises numerous interesting challenges and research questions.

Communication networks involve many components to transport data from one endpoint to the other. In fact, even the seemingly simple local task of forwarding a data packet from one port to the other involves many components, e.g., packet buffers, memory units, and queuing disciplines. As a result, gaining a deep understanding of all the involved components, the possible sources of delay, their causes and influential factors, is a challenging problem, especially given the fact that switching hardware has not been built with predictability in mind, but rather efficiency and statistiscal perfor-

mance. This thesis provides measurement procedures and a predictability study of programmable forwarding devices. Results from this study are used to devise precise performance models of forwarding hardware. Such models are a required step towards the design of a network providing strict latency guarantees to its tenants. Our evaluations in several testbed setups show that the models we propose can provide worst-case end-to-end latency guarantees to applications without sacrificing network utilization.

With a setup involving a centralized controller configuring forwarding components, typical for programmable networks, a routing procedure is responsible for finding paths for communication flows. The goal of providing strict latency guarantees requires the routing procedure to consider not only the physical links where packets are forwarded but also how packets are scheduled at each link, e.g., at which priority level. At the same time, the procedure must ensure that resources are allocated wisely to flows, in order to increase the number of flows that can be simultaneously accommodated, and hence increase revenue for the network operator. This thesis provides the design and thorough evaluation of a routing procedure that defines both the physical links packets follow and the priority level at which these packets are queued at each link. Existing algorithms are investigated and new algorithms are designed to improve on the runtime, optimality, and completeness of state-of-the-art algorithms in different scenarios and problem settings.

In such an online setup where flows are embedded at runtime, the forwarding behavior of switches has to be updated without hindering the predictability of the forwarding operation performed in the data plane. Measurement campaigns in the literature have already shown that runtime updates in programmable networks can lead to unpredictable behavior. This thesis investigates the impact of these runtime reconfigurations on latency guarantees in the data plane. The result is the design, implementation, and evaluation of complete systems providing latency guarantees to online flow embedding requests. We design two systems, namely *Loko* and *Chameleon*, respectively focusing on small and low-capacity networks, and on data center networks. While *Loko* prevents unpredictability due to reconfigurations by including these operations in the latency modeling of the forwarding elements, *Chameleon* relies on end-host networking, and in particular source routing, to circumvent the need for reconfiguring networking devices.

Kurzfassung

Unsere digitale Gesellschaft baut stark auf Kommunikationsnetze, um Menschen mit Rechenzentren, Rechenzentren miteinander, und Sensoren und Aktuatoren mit Steuerungseinheiten zu verbinden. Diese hohe Konnektivität wird von Industrie und Behörden benutzt, um immer mehr Dienste durch Systeme wie das Internet der Dinge, cyber-physische Systeme, Smart Cities, das Smart Grid, Cloud Computing, und allgemeiner 5G Netze, zur Verfügung zu stellen. Seit dem Einsatz des ARPANETs und des ersten Internet Knoten in 1969 in der University of California, Los Angeles, sind das Internet und Technologien basierend auf Informationsaustausch stark gewachsen und entwickeln sich mit zunehmender Geschwindigkeit. Heutige Kommunikationsnetze wurden entwickelt, um einen universellen Kommunikationsdienst zu bieten, aber unsere moderne digitale Gesellschaft und die immer unterschiedlicheren Anwendungen zwingen der Kommunikationsinfrastruktur neue Anforderungen auf. Die Qualität der Dienste, welche den Nutzern und Kunden angeboten wird, ist abhängig von der Qualität der Dienste, welche die physikalische Infrastruktur bietet.

Im Allgemeinen benötigen die neue Anwendungen eine *berechenbare Vorhersagbarkeit* der Kommunikationsnetze, sowohl hinsichtlich Korrektheit als auch Leistung. Aufgrund der verteilten Natur von Kommunikationsnetzen ist Vorhersagbarkeit sehr schwer zu garantieren. Daher wurden in den letzten Jahren statisch konfigurierte Netze immer mehr von sogenannten *programmierbaren Netzen* ersetzt, in denen das Verhalten der Netze und insbesondere ihrer konstituierenden Knoten nicht mehr von Konfigurationsdateien und unveränderlicher Logik geregelt ist, sondern von Software, die unabhängig von der Hardware läuft. Mit derartigen neuen Konzepte wie OpenFlow und P4 können Netzbetreiber das Verhalten ihrer Kommunikationsnetze von außen programmieren und damit die Kosten reduzieren, die Flexibilität erhöhen, und am wichtigsten die stark variierenden und sich schnell ändernden Anforderungen von heutigen Anwendungen effizient bewältigen. In welchem Ausmaß diese neuen Technologien und Netzkonzepte die Vorhersagbarkeit, die von neuen Anwendungen und Diensten benötigt wird, bieten können ist eine offene Forschungsfrage. Diese Dissertation untersucht das Potenzial von programmierbaren Netzen um vorhersagbare *Latenz* für Anwendungen zu bieten. Latenz ist eine kritische Metrik für Anwendungen, da eine berechenbar vorhersagbare Latenz es ermöglicht, den Nutzern Garantien für Bearbeitungszeiten zu bieten. Insbesondere wird in dieser Dissertation die Unterstützung strikter pro-Paket Latenzgarantien untersucht. Vorhersagbarkeit und strikter Determinismus in verteilten Systeme zu realisieren, wirft einige interessante Herausforderungen und Forschungsfragen auf.

Kommunikationsnetze bestehen aus vielen verschiedenen Komponenten, um Daten von einem Endpunkt bis zum anderem zu übermitteln. Tatsächlich sind sogar für die anscheinend einfache Aufgabe ein Paket von einem Port zum anderen weiterzuleiten viele verschiedene Komponenten betei-

ligt; zum Beispiel Paket Puffer, Speichereinheiten, und Warteschlangendisziplinen. Infolgedessen ist es ein herausforderndes Problem, alle verschiedene Verzögerungsquellen, ihre Ursachen und Einflussfaktoren, zu verstehen und in Zahlen auszudrücken. Was dieses Problem weiter verkompliziert, ist die Tatsache, dass Weiterleitungskomponenten für Effizienz und statistische Leistung entwickelt wurden, und nicht, um deterministisch garantierte Eigenschaften bereitzustellen.

Diese Dissertation präsentiert Messmethoden und eine Vorhersagbarkeitstudie von programmierbaren Weiterleitungselementen in Netzen. Beobachtungen und Ergebnisse dieser Studie ermöglichen anschließend die Entwicklung von präzisen Leistungsmodellen für Weiterleitungselemente. Solche Modelle bilden die Grundlage, um Netzwerke mit Latenzgarantien zu entwerfen. Unsere Evaluierungen und Messanalysen zeigen, dass die Modelle die wir vorschlagen, strikte Ende zu Ende Verzögerung garantieren können, ohne die Netzauslastung zu schmälern.

Wenn ein Netz durch eine zentrale Steuerung geregelt ist, was für programmierbare Netze typisch ist, ist ein Routingalgorithmus verantwortlich für das Finden von Pfaden für die Kommunikationsflüsse. Das Ziel strikte Latenzgarantien zu bieten erzwingt spezielle Anforderungen an den Routingalgorithmus, der nicht nur physische Verbindungen auswählen muss, sondern auch entscheiden muss, wie Pakete und mit welcher Priorität sie an jedem Gerät gesteuert werden. Gleichzeitig muss der Algorithmus sicherstellen, dass Netzressourcen den Flüssen so zugewiesen werden, dass die Anzahl an Flüssen die gleichzeitig im Netz angenommen werden können, maximiert wird und dadurch Einnahmen für den Netzbetreiber so viel wie möglich erhöht werden. In dieser Dissertation wird ein derartiger Routingalgorithmus untersucht, und dabei der Entwurf und eine detaillierte Evaluierung von Routingalgorithmen vorgestellt. Vorhandene Algorithmen werden untersucht und neue Algorithmen entwickelt, um die Laufzeit, Optimalität, und Vollständigkeit vorhandener Routingalgorithmen in verschiedenen Szenarien und Situationen zu verbessern.

Um Flüsse zur Laufzeit im Netzwerk einzubetten muss das Verhalten der Weiterleitungselemente auf einen neuen Stand gebracht werden ohne die Vorhersagbarkeit der Weiterleitungsoperationen in der Datenebene zu behindern. Es wurde bereits in der Literatur gezeigt, dass Laufzeitaktualisierungen in programmierbaren Netzen zu unvorhersagbarem Verhalten führen können. In dieser Dissertation wird der Einfluss dieser Laufzeitaktualisierungen auf Vorhersagbarkeit und Verzögerung in der Datenebene untersucht. Basierend auf den Ergebnissen dieser Untersuchung werden Systemen die Latenzgarantien für Echtzeitflüsse bereitstellen entwickelt, implementiert, und evaluiert. Zwei verschiedene Systemen werden vorgestellt: *Loko*, das auf kleine Netze mit geringe Kapazität fokussiert, und *Chameleon*, das für Rechnerzentrumsnetze geeignet ist. *Loko* vermeidet Unvorhersagbarkeit durch eine mathematische Modellierung der Aktualisierungsoperationen und *Chameleon* konfiguriert das Verhalten der Netze von den End-Hosts her, um das Bedürfnis nach eine Aktualiserung der Netzelemente zu vermeiden.

Contents

Chapter 1

Introduction

Communication networks lie at the heart of our digital society, connecting users, data centers, sensors, actuators, and many other devices together. Originally designed for providing a best-effort connectivity service, the shift from human-based communications towards automation and machine-to-machine (M2M) communications imposes new important requirements for the underlying infrastructure. The ongoing fourth industrial revolution, referred to as Industry 4.0, indeed pushes towards automation of manufacturing processes and business logic, which involves data exchange between sensors, actuators, controllers, and other servers in systems such as the internet of things (IoT), cyber-physical systems (CPSs), smart cities, smart grids, cloud computing, or more generally 5G networks [Sha+17]; [ITU15]; [Agy+14]; [GH13]; [Lin+17]; [Yaq+17]; [KRR16]; [Yan+12]. Such systems offer services to their users, customers or tenants, the quality of which depends on the quality of service (QoS) offered by the underlying physical communication infrastructure. Generally, these emerging applications require that the communication network is highly *predictable*, both in terms of correctness as well as in terms of performance, so that they can in turn provide service guarantees to their users. For example, a safety-critical control loop in a manufacturing plant can require its data to be transmitted within a given strict latency bound in order to ensure a proper reaction is triggered on time in case of a particular event.

Generally speaking, network performance can be quantified through different statistical measures (e.g., average, minimum, or maximum) of different end-to-end (E2E) network properties (e.g., throughput, delay, jitter, packet loss or availability) [Sta15]. Different applications require different performance guarantees to provide their services. For example, while a video surveillance service might only require average performance guarantees in terms of throughput and jitter, the sensors and actuators involved in a safety-critical control loop such as mentioned above would rather require maximum latency and minimum availability guarantees. In this thesis, we define *predictability* as the ability of a network or network device to perform its operations with strict performance guarantees, i.e., able to provide guarantees for the absolute maximum or minimum of an E2E network property.

This thesis focuses on applications that require predictable *latency*, i.e., that require strict E2E latency guarantees. Industrial communications (e.g., M2M communications and networked control loops) are an example of such applications [SJH06]; [Dec05]. Sensors send measurement signals to a controller which is responsible for triggering appropriate and timely reactions on actuators. For example, in an electrical grid, based on the state of the grid measured by sensors, wind turbines sometimes have to reduce their electricity production to avoid instabilities in the electrical network. The breaking system, i.e., actuators, in the wind turbine nacelle must then be triggered to reduce the electricity production on time. Another example is a robot that operates in an environment with humans on a production line and equipped with sensors that monitor its environment. A controller must make sure that the robot does not damage equipment or injure humans while moving and must hence timely trigger actuators based on the data received from the sensors. These communications have strict QoS requirements, mainly in terms of E2E delay [05]. That means that packets involved in such communications must all reach their destination within a given maximum E2E latency bound. Violations of this bound are not acceptable, as they could lead to dramatic situations, from financial penalties for the wind turbine example to physical damage and human casualties in the moving robot scenario. Such flows are often referred to as *real-time flows*, and we interchangeably use *real-time*, *predictable latency*, and *strict* or *industrial-grade* to refer to the level of QoS needed by such flows.

A wide gamut of distributed proprietary solutions [Sau10] and extensions of Ethernet [Dec05]; [Li+17] have been developed for providing this strict QoS. Such solutions include Profinet, Profibus, CAN, or Modbus. However, in order to ensure predictability, these solutions typically impose restrictions on the topology that can be deployed, or require changes within the network protocol stack, which leads to expensive forwarding devices. The lack of compatibility between the different industrial Ethernet protocols further leads to vendor lock-in because network operators must deploy devices from the same manufacturer to ensure their interoperability. That slows down innovation and increases the costs and the probability of errors, as such proprietary devices usually require customized scripting tools and many hours of testing performed by highly specialized network engineers [Viz+19].

Programmable networks, and in particular software-defined networking (SDN), have been seen in the recent years as a solution to overcome such protocol openness issues and improve the automation and flexibility of network configuration and management [Kel+19]. SDN decouples the control plane (CP) of forwarding devices from their data plane (DP). Using standardized and open interfaces (e.g., OpenFlow (OF) [McK+08]), a logically centralized controller configures the forwarding behavior of switches based on the global knowledge of the network state. This centralized and direct control over network devices and the resulting management flexibilities [Kel+19] have brought the potential to greatly improve predictability and efficiency, at least from a CP logic point of view. In particular, it enables faster and more fine-grained flow-level control and management than proprietary solutions like Profibus or CAN which rely on distributed mechanisms. Also, SDN only requires simple commodity forwarding elements that can be changed and updated independently [HWJ16] and hence avoids vendor lock-in and provides solutions at a cost lower than proprietary protocols that require specialized hardware to enable deterministic guarantees [Sau10]; [GJF13b]; [Viz+19].

The goal of this thesis is the design, implementation, and evaluation of mechanisms for predictable latency in programmable networks, and in SDNs in general. In such a centrally managed network, requests for connections with predictable latency are expected to arrive over time at the northbound interface (NBI) of a logically centralized controller. While network programmability and SDN are promising technologies for the design and implementation of traffic engineering mechanisms, it opens many research challenges. We discuss these challenges in the next section.

1.1 Research Challenges

The design and implementation of solutions for predictable latency in programmable networks comprises various challenges. This section summarizes the main research challenges tackled in this thesis. The next section then reports on the main contributions of the thesis.

Measurement procedures and predictability study of programmable forwarding devices.

A big challenge in the design of mechanisms for predictable latency in SDNs is to gain a deep understanding of the behavior of programmable forwarding devices. In fact, even seemingly simple tasks, such as forwarding, involve many complex components, such as link buffers, hardware memory units, switch central processing units (CPUs), queuing disciplines, etc. A deep understanding of the behavior of all these components is necessary for guaranteeing predictable network operations. While the proprietary solutions that exist for predictable latency modify the behavior of forwarding devices to ensure predictability, a SDN solution based on commodity off-the-shelf programmable devices must build on top of hardware that was not originally designed for predictability. The first research challenge is hence to gain a better understanding of the behavior of existing forwarding devices through the design and application of measurement procedures for unveiling behavioral artifacts with respect to predictability, or to empirically confirm the predictability of some components.

Modeling of programmable forwarding devices for worst-case performance prediction.

The main cause of delays in modern networks is the presence of *microbursts* [Gho+17]; [Jos+18] that collide at the egress ports of switches, thereby generating queuing and hence delay. Providing predictable latency requires the appropriate traffic and switch models for predicting the worst-case delay a packet from a microburst can experience at any hop on its path to its destination. While proprietary solutions modify the forwarding elements to be able to model their worst-case behavior easily, a SDN-based solution for predictable latency must design worst-case latency models for commodity programmable hardware. That is a challenging task as commodity SDN hardware was not designed with predictability in mind. Models have to be designed and thoroughly verified through comprehensive benchmarking campaigns to ensure that, within a given range of scenarios and use cases, the latency experienced by packets at a given switch can be strictly upper-bounded. Besides providing correct models, network operators also strive for high utilization, as more accepted connections means more revenue. Avoiding to be too pessimistic while still providing correct latency guarantees is a major challenge, as less pessimistic predictions lead to more accepted flows and hence higher network utilization.

Fast and efficient path finding.

The practical implementation of a centralized controller is a challenging task. First, from an algorithmic point of view, a routing procedure is responsible for finding a path for arriving requests in a reasonable amount of time. This routing procedure should interact with a network model for E2E latency predictions. While models are expected to grasp complex queuing disciplines and flows multiplexing in the network, the interaction between these two components must happen fast. As the main interface to the tenants of the network, low runtime is a key feature of the routing procedure. At the same time, in order to increase the number of applications that can be accommodated and hence increase revenue for the operator, efficient network resources utilization is a second key asset for the routing procedure. Understanding and designing such a routing procedure to achieve these goals in the context of centralized management for predictable latency is an open research question.

Predictable configuration and reconfiguration of forwarding devices.
After the routing procedure finds an embedding for a new application, this embedding has to be programmed in the network. However, while recent efforts have shown the numerous benefits, e.g., in terms of flexibility [Kel+19], of programmability, the configuration of forwarding rules must be done carefully to avoid interferences in the DP. This CP to DP potential vector of interferences is a challenging research direction. In particular, reconfigurations, a major strategy for (re-)optimizing path selection and hence improving network utilization, require forwarding configurations to be changed perpetually and at runtime. The problem of finding an ordered set of reconfiguration commands to preserve the network guarantees is a challenging algorithmic issue. Besides, each reconfiguration operation must be performed, in practice, without interfering with the current operation of the network. This might have as well to be included in the E2E latency modeling. The design and implementation of such a configuration and reconfiguration strategy for programmable devices that keeps strict guarantees in the DP is another challenging research problem.

1.2 Contributions

This section summarizes the main contributions of the thesis towards predictable latency in programmable networks. Fig. 1.1 illustrates the structure of the thesis by highlighting the three main investigated research directions along with the used methodologies and concepts and the associated publications from the author where these original contributions were published.

The first and minor contribution of this thesis is based on deterministic services (DetServ), an architecture and an associated latency model for the provisioning of latency guarantees that was already presented in a previous doctoral thesis from the Technical University of Munich [Guc18]. A model of traditional forwarding mechanisms is established and extended to design an E2E network model that supports access control and resource reservation. We additionally identify and clearly define the necessary logical components within this architecture. We perform packet-level simulations that show that such an architecture and its associated network model indeed can provide strict latency guarantees to applications. Through Monte Carlo simulations, we further show that this architecture also provides the means of achieving low request processing time, a key feature of a network QoS framework. An extended architecture for supporting networks with wireless hops, wireless deterministic services (WDetServ), is also designed and presented.

The second contribution of this thesis is the algorithmic design and analysis of a routing procedure for the online embedding of routing requests with strict E2E latency requirements. Working hand in hand with a network model, the routing procedure is responsible for finding an embedding (consisting of a physical path and priority levels at each hop) that fulfills the latency requirement of a new flow and that does not violate the guarantees provided to previously embedded flows. By splitting the problem in several components, we show that the routing procedure can be boiled down to a delay-constrained least-cost (DCLC) routing problem and the design of a corresponding per-

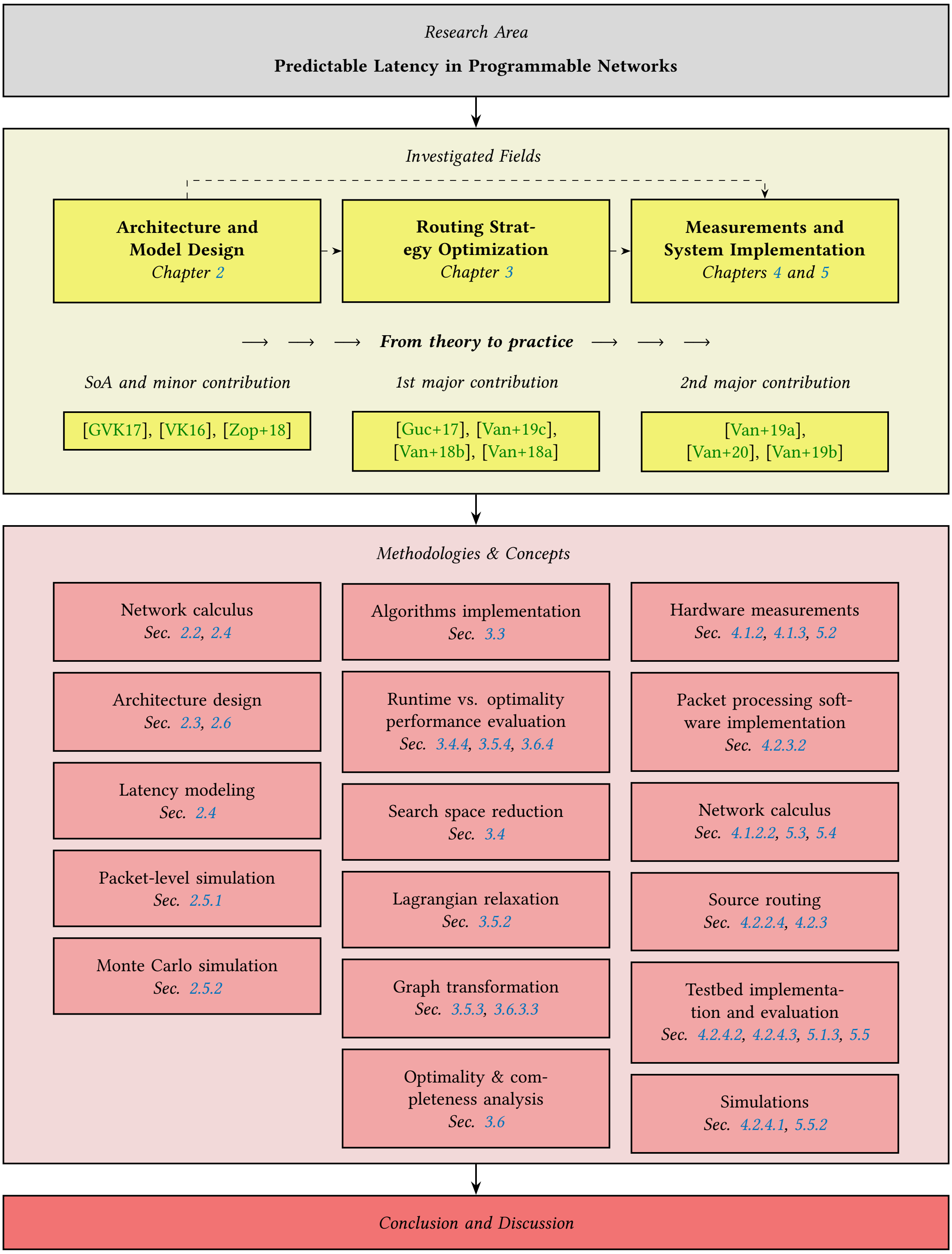

Research Area
Predictable Latency in Programmable Networks
Investigated Fields
Architecture and Model Design
Chapter 2
Routing Strategy Optimization
Chapter 3
Measurements and System Implementation
Chapters 4 and 5
⟶ ⟶ ⟶ From theory to practice ⟶ ⟶ ⟶
SoA and minor contribution
1st major contribution
2nd major contribution
[GVK17], [VK16], [Zop+18]
[Guc+17], [Van+19c], [Van+18b], [Van+18a]
[Van+19a], [Van+20], [Van+19b]
Methodologies & Concepts
Network calculus Sec. 2.2, 2.4
Architecture design Sec. 2.3, 2.6
Latency modeling Sec. 2.4
Packet-level simulation Sec. 2.5.1
Monte Carlo simulation Sec. 2.5.2
Algorithms implementation Sec. 3.3
Runtime vs. optimality performance evaluation Sec. 3.4.4, 3.5.4, 3.6.4
Search space reduction Sec. 3.4
Lagrangian relaxation Sec. 3.5.2
Graph transformation Sec. 3.5.3, 3.6.3.3
Optimality & completeness analysis Sec. 3.6
Hardware measurements Sec. 4.1.2, 4.1.3, 5.2
Packet processing software implementation Sec. 4.2.3.2
Network calculus Sec. 4.1.2.2, 5.3, 5.4
Source routing Sec. 4.2.2.4, 4.2.3
Testbed implementation and evaluation Sec. 4.2.4.2, 4.2.4.3, 5.1.3, 5.5
Simulations Sec. 4.2.4.1, 5.5.2
Conclusion and Discussion

Figure 1.1: Thesis structure. The thesis investigates three main different research directions in the area of predictable latency in programmable networks: architecture and model design, routing strategy optimization, and measurements and system implementation. Whereas the two first fields mostly focus on theoretical concepts, the last two chapters on measurements and system implementation focus on practical aspects and issues and aim at the design of a complete working prototype system for providing latency guarantees to applications.

queue cost function. We show that many algorithms have already been proposed for this problem and evaluate them thoroughly in many different scenarios. Besides identifying a few algorithms as among the best in most cases, we show that one cannot rely on an optimal algorithm for online route computations and *heuristics*, i.e., sub-optimal algorithms, must be used. We further show that the per-queue access control of the DetServ latency model influences the optimality and completeness of these algorithms. We formalize this effect by proposing the *Mn taxonomy*, a taxonomy that classifies routing metrics based on the number n of previously visited edges that are needed to compute the metric value at a given edge. For $n > 1$, we show that the optimal substructure property (OSP) that forms the basis of most routing algorithms is violated: algorithms then lose their properties. Whether optimality and/or completenesss is/are lost depends on the Mn classification of the involved routing metrics and on whether these metrics are used for path optimization or for constraining the set of valid paths. We propose several solutions, namely edge-based Dijkstra (EBD) and graph transformation algorithm (GTA), for recovering these key features in certain cases, at the price of an increased runtime. We also design Lagrange relaxation based aggregated cost for specified nodes (LARAC-SN) and mole in the hole (MITH), two routing algorithms for finding delay-constrained paths traversing service function chains (SFCs), and propose bounded Dijkstra (BD), a search space reduction method for speeding up algorithms that are based on several subsequent shortest path subroutines (e.g., Dijkstra runs). We show that this method allows to drastically reduce the runtime of such algorithms – for favorable cases, by 96% on average for some algorithms – without impacting their output and hence optimality.

Finally, the third and major contribution of this thesis is the investigation of the predictability of SDN hardware and the adaptation of the originally proposed architecture and network model for the design, implementation, and evaluation of proof-of-concept systems providing strict latency guarantees to applications. In particular, we focus on two different types of network: data center networks and small networks, which we define as networks with low-cost and low-capacity forwarding devices. For data center networks, through a thorough measurement campaign, we observe that existing hardware can present many different sources of unpredictability, mostly in the implementation of the control agent. To overcome these observed behavioral artifacts, we design, implement, and evaluate *Chameleon*, a demand-aware cloud network that combines adaptive source routing with priority queuing to meet both predictable latency *and* resource efficiency objectives. *Chameleon* dynamically reevaluates routing decisions, performing adjustments while maintaining network calculus invariants to ensure strict latency guarantees are provided and preserved. The system avoids interacting with the forwarding devices by completely relying on end-host data plane development kit (DPDK)-based tagging for defining forwarding and enqueuing decisions at each hop. Besides ensuring strict latency predictability in the network, we show that *Chameleon* allows to easily incorporate route and priority level reconfigurations and that this in turn allows to reach higher network utilization and hence reduce the flow rejection rate. For small networks, we observe that traditional assumptions taken by latency models are not valid. In particular, low-capacity forwarding devices have lower processing throughput and can generate inter-port interferences. This leads to state of the art (SoA) systems that fail at providing their guarantees. Based on this observation, we design, implement, and evaluate *Loko*, a system that provides latency guarantees with low-cost and low-capacity hardware. The system is based on a thorough switch benchmarking, based on which a tailored per-switch latency model is derived and used for global access control and resource reservation. Using testbed measurements of our proof-of-concept implementation, we show that *Loko* indeed provides latency guarantees in networks based on low-cost and low-capacity forwarding elements. This work opens several interesting avenues for future research, as it shows that tailored implementations on low-cost devices such as DPDK-based implementations of virtual network functions (VNFs) for packet processing on multi-port network interface cards (NICs) can provide predictability and performance guarantees.

In order to ensure reproducibility, and as a contribution to the research community, most of our source code, configuration files, and data sets have been made available online [Van19c]; [Van19b]; [Van19a].

1.3 Outline

The remainder of the thesis is organized as follows.

Chapter 2 describes the related work on the provisioning of QoS in programmable networks and establishes the DetServ architecture and E2E network latency model. The path followed towards the definition of this architecture and model is clearly described. The resulting DetServ system is then theoretically evaluated through packet-level and Monte Carlo simulations to prove its correctness and potential for low request processing time and high network utilization. An extensive treatment of deterministic network calculus (DNC), the mathematical concepts on which our network models are based, is also provided.

Chapter 3 investigates in detail the routing procedure of the DetServ architecture. Existing algorithms are evaluated in detail and compared in different network and routing request settings. Algorithms are proposed to improve the runtime of shortest path subroutines and to route flows through SFCs. A taxonomy is further proposed to formalize the impact the DetServ network model has on the optimality and completeness of existing algorithms. All evaluations in this chapter are conducted through simulations by evaluating the runtime, optimality gap, and completeness of algorithms in different network and request settings.

Chapter 4 brings the previous chapters into practice by benchmarking existing SDN hardware and analyzing how the previously defined theoretical model can be applied to them. Several unpredictability sources are identified and circumvented by the design of a solution based on source routing and perpetual route and priority reconfigurations. *Chameleon*, the resulting system, is designed, implemented, verified, and evaluated with a proof-of-concept implementation in a data center testbed.

Chapter 5 further analyzes how the models operate in practice by investigating the deployment of predictable latency solutions in small networks. We observe that low-cost and low-capacity hardware do not fulfill the traditional assumptions of SoA E2E latency models, including those defined in chapter 2. A thorough benchmarking of the processing time, throughput, and buffer capacity of the Zodiac FX switch [Nor19] is performed. Based on the results, an adapted model is designed and used for the design, implementation, and evaluation of access control and resource reservation routines that enable the provisioning of latency guarantees with Zodiac FX switches. Results are validated in a proof-of-concept testbed implementation.

Finally, chapter 6 concludes this thesis by summarizing the main results and messages and by giving a brief outlook for interesting future work and research directions.

Chapter 2

Architecture Design and Network Modeling for Predictable Latency

This chapter sets the scene for chapters 3, 4, and 5 by describing the architecture and modeling principles on which this thesis is based. In particular, we precisely describe DetServ, an architecture for the provisioning of *det*erministic *serv*ices, and in particular predictable latency, in programmable networks. This architecture relies on SDN principles, by which a centralized controller manages and configures the switches in the network. This is shown in Fig. 2.1. Tenants request the embedding of a flow for a given application through the NBI of a centralized controller. Such an architecture is fundamental for providing strict guarantees. Indeed, unpredicted microbursts are the main cause of delays in modern networks [Gho+17]; [Jos+18], and having such an entry point to the network where service level agreements (SLAs) are negotiated allows to control microbursts, predict the worst-case delay they can generate, and hence compute worst-case latency guarantees. A routing procedure is then responsible for finding a path that satisfies the requirements of the application.

Guaranteeing deterministic services to applications requires the definition of architectural components performing well-defined and particular tasks. Latency computations must be done entirely in the CP and must not rely on DP measurements, as this leads to unprecise and potentially stale computations [GRK15]. This is done by keeping track of the whole state of the network, i.e., of the set of applications and their current resources usage, in the CP. The main component of the DetServ architecture is the *network model*, responsible for the modeling of latency in the network and for accordingly computing E2E latency guarantees. This is a challenging problem, as the switches we consider available in programmable and SDN networks consist of simple forwarding elements based on ternary content-addressable memory (TCAM), supporting only limited protocols and features (e.g., Ethernet and priority queuing) and implementing limited or no control logic but rather relying on a centralized controller for determining their forwarding behavior. One of the goal of this modeling is also to avoid requiring additional features from the forwarding elements, as this rapidly leads to more expensive devices. The quest for predictable latency, i.e., for latency bounds that are deterministically guaranteed, requires us to adopt a deterministic modeling framework, in contrast to stochastic modeling approaches like queuing theory and stochastic network calculus (SNC). Our modeling hence relies on DNC principles, a mathematical framework and system theory for the computation of worst-case performance bounds in communication networks. Our description of the Det-

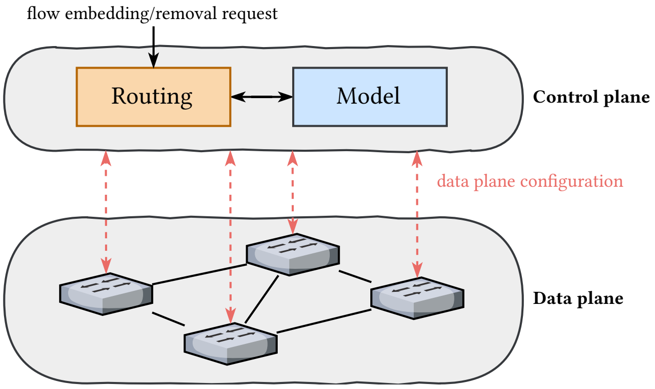


Figure 2.1: Global architecture that forms the basis of this thesis. A centralized SDN controller is responsible for handling flow embedding requests. With this aim, a *routing procedure* relies on the computations of a *network model* to guarantee latency bounds to applications. To avoid unprecise and stale information, the network model keeps track of the whole state of the network (the accepted flows and where they are embedded) in the CP and does not rely on measurements from the DP.

Serv architecture defines the components needed in such a network model to achieve all these goals. The resulting architeture is shown in Fig. 2.26. To show the generality of the DetServ architecture, we show how it can be applied to incorporate wireless links in the network. We accordingly define WDetServ (Fig. 2.33). Through packet-level and Monte Carlo simulations, we confirm that the DetServ architecture and model indeed provide correct latency bounds and with a reasonable processing time for online embedding scenarios (hundreds of milliseconds in the worst cases). Throughout this thesis, we assume that a centralized controller manages DP devices through an out-of-band interface. The consideration of a scenario where the controller is connected in-band, i.e., using links that are used for DP communications, is left out of this thesis but has been investigated in [Sak+20].

Content and outline of this chapter. Sec. 2.1 covers the related work in the area of predictable latency and QoS provisioning in SDN. Sec. 2.2 provides introduction material on DNC, the modeling framework that forms the basis of this thesis. This material essentially relies on content from [VK16]. The reader familiar with these concepts can freely skip this section. The detailed architecture design and modeling of E2E network latency are presented in Sec. 2.3 and 2.4. A proof-of-concept packet-level simulation confirming that the provided guarantees are indeed met is presented in Sec. 2.5. Additional simulations assessing the network utilization achievable by the model are also presented. Sec. 2.6 describes WDetServ, an extension of the DetServ architecture and network model for supporting networks with wireless hops. The DetServ architecture and network models are the result of a bilateral collaboration published in [GVK17]. The network model and a first version of the architecture are available in a previous doctoral thesis from the Technical University of Munich [Guc18]. In addition to presenting again these results, we provide an extended architecture description and the additional packet-level simulations and Monte Carlo simulations of Sec. 2.5. The WDetServ description in Sec. 2.6 is based on condensed material from [Zop+18]. Finally, Sec. 2.7 concludes this chapter and portrays how the next chapters build on top of it.

2.1 Related Work

Industrial applications have for a long time been a major use case for predictable latency. Initially, proprietary solutions (e.g., Profibus, Interbus or CAN) have been specifically developed for real-time industrial communications [GJF13b]; [Sau10]. However, these solutions often come with a complete proprietary communication stack which requires specialized and expensive hardware.

Later, Ethernet data transfer rates increased and Ethernet became ubiquitous in local area networks (LANs) and the Internet. Therefore, it attracted a lot of attention for industrial deployments. However, because of its non-deterministic medium access control (MAC) scheme, Ethernet was initially not considered as a suitable solution. The usage of full duplex point-to-point links along with Ethernet switches instead of shared buses and hubs allowed to avoid collisions and hence the negative impact of the Ethernet MAC protocol [Dec05]. Nevertheless, this introduces buffering and possibly overflows, which were still considered to be a source of non-determinism [Dec05]. Despite this, using Ethernet has major benefits, including simple and cheap deployment, easy connectivity towards office networks, the Internet or more generally any Internet protocol (IP) traffic, and usage of off-the-shelf communication hardware. Hence, many industrial control systems manufacturers decided to develop proprietary extensions of Ethernet to achieve determinism [JN04]; [GJF13b]. A broad overview of Ethernet-based real-time technologies, including deterministic Ethernet standards, was provided by Decotignie [Dec05]. Unfortunately, these solutions require changes within the network protocol stack or impose topology restrictions or both, which leads to more expensive forwarding devices than with standard Ethernet.

The emergence of SDN and network programmability provided a new opportunity for traffic engineering in Ethernet networks, where work focusing on QoS for data center applications have become prominent. An overview of the most important existing works and their respective features in shown in Tab. 2.1.

Prior to the 2010s, tenants of cloud networks were offered the possibility of paying for compute resources, i.e., for virtual machines (VMs). However, the network resources connecting these VMs, which are shared among all tenants, were not included in pricing. Cloud offers had no abstraction or mechanism for allocating bandwidth between VMs. This led to bad performance and unpredictability for applications running in the cloud [Guo+10]; [Bal+11]. As a result, many efforts have been trying to provide bandwidth guarantees, work conservation, inter-tenant fairness, and isolation, or a combination of these, in cloud networks. SecondNet [Guo+10] and Oktopus [Bal+11], the first works in this direction, proposed to add network resources into the cloud pricing and provided bandwidth guarantees through traffic shaping and rate-limiting at the end-host hypervisors. These approaches are not work-conserving, i.e., the bandwidth reserved for a flow is left unused if this flow does not use it. That leads to loss of revenue for the operator, as this bandwidth could be used to finish flows from other tenants. With this aim, Seawall [Shi+11] proposed to use congestion-controlled tunnels implemented in a shim layer that intercepts all packets entering and leaving a server. This layer, responsible for congestion control feedback, ensures that bandwidth is used by a flow if available. However, Seawall provided no bandwidth guarantees to tenants. Gatekeeper [Rod+11] proposed an approach to provide both bandwidth guarantees and work-conservation by using a distributed mechanism at the virtualization layer of each server that controls the usage of the network access link of each server

Name	Guarantees				Constraints		
	Pkt. lat.	**BW**	**Burst**	**WC**	**Switches req.**	**OS/App. changes**	**Other**
SoA focusing on providing bandwidth (BW) guarantees and/or being work-conserving (WC).							
SecondNet [Guo+10]	✗	✓	✗	✗	PQ, MPLS	-	-
Oktopus [Bal+11]	✗	✓	✗	✗	PQ	-	-
Seawall [Shi+11]	✗	✗	✗	✓	-	-	-
Gatekeeper [Rod+11]	✗	✓	✗	✓	-	-	Non-congested core
TIVC / Proteus [Xie+12]	✗	✓	✗	●	-	-	-
NetShare [Lam+12]	✗	✗	✗	✓	WFQ	-	-
FairCloud PS-L/PS-N [Pop+12]	✗	✗	✗	✓	WFQ	-	-
FairCloud PS-P [Pop+12]	✗	✓	✗	✓	WFQ	-	Tree topology
EyeQ [Jey+13]	✗	✓	✗	✓	ECN	-	Non-congested core
Elasticswitch [Pop+13]	✗	✓	✗	✓	-	-	-
Hadrian [Bal+13]	✗	✓	✗	✓	Custom protocol	-	-
Trinity [Hu+16]	✗	✓	✗	✓	PQ, ECN	-	-
HUG [Cho+16]	✗	✓	✗	✓	-	-	-
eBA [Liu+16]	✗	✓	✗	✓	Custom protocol	-	-
QShare [Liu+18]	✗	✓	✗	✓	WFQ	-	-
SoA optimizing the transport protocol.							
DCTCP [Ali+11]	✗	✗	✗	✓	ECN	OS	-
D^3 [Wil+11]	✗	✗	✗	✓	Custom protocol	OS, App.	-
PDQ [HCG12]	✗	✗	✗	✓	Custom protocol	OS, App.	-
D^2TCP [VHV12]	✗	✗	✗	✓	ECN	OS, App.	-
HULL [Ali+12]	✗	✗	✗	✓	Custom feature	OS	-
DeTail [Zat+12]	✗	✗	✗	✓	Custom protocol	OS, App.	-
pFabric [Ali+13]	✗	✗	✗	✓	Custom protocol	OS, App.	-
NDP [Han+17]	✗	✗	✗	✓	Custom protocol	OS	-
Homa [Mon+18]	✗	✗	✗	✓	PQ	OS	-
HPCC [Li+19]	✗	✗	✗	✓	Custom protocol	OS	-
SoA providing per-packet latency guarantees.							
TDMA Ethernet [Vat+12]	✓	✓	✓	✗	-	-	Millisecond timescale
Fastpass [Per+14]	✓	✓	✓	✗	-	OS	End-hosts synchronization
QJump [Gro+15]	✓	✓	✗	✗	-	-	-
Silo [Jan+15]	✓	✓	✓	✗	-	-	Multi-rooted tree topology
LaaS [Zah+19]	✓	✓	✓	✗	-	-	Tenants on diff. phys. links
Contributions in this thesis							
DetServ (chapter 2)	✓	✓	✓	✗	PQ	-	-
Chameleon (chapter 4)	✓	✓	✓	✗	PQ, VLAN	-	-
Loko (chapter 5)	✓	✓	✓	✗	-	-	Tailored to low-cost devices

Table 2.1: Overview of the main modern solutions for the provisioning of QoS in programmable networks. While many approaches focus on providing BW guarantees or optimize the transport protocol to reduce FCT and/or tail latency, only a few approaches provide strict latency guarantees, the focus of this thesis. These approaches suffer from limitations such as the need for synchronization or the support of only particular topologies. Our contributions in this thesis provide strict latency guarantees with precise BW and burst allowance in any general network without any particular requirements. Furthermore, while Loko (chapter 5) is tailored to low-cost and low-capacity networks for which none of the existing solutions work, we show in chapter 4 that DetServ and Chameleon reach higher network utilization than existing approaches.
Note that *App.* change does not include the usage of another transport library but only the requirement for providing more information to this library (e.g., deadline).

and accordingly provides per-virtual network interface card (vNIC) link bandwidth guarantees. Each vNIC can exceed its guaranteed allocation when extra bandwidth is available at both transmitting and receiving endpoints. Gatekeeper however assumes that the core of the network is not congested. In order to achieve work-conservation, Proteus [Xie+12] proposed temporally interleaved virtual clusters (TIVCs). The requirements of applications are expressed in terms of time windows of given width (time) and height (bandwidth). An algorithm then allocates demands both spatially and temporally. That allows for work-conservation, but only if the time dimension of applications is profiled correctly, what remains a very complicated task. NetShare [Lam+12] proposed to achieve work-conservation by using weighted fair queuing (WFQ) for allocating bandwidth predictably across services based on weights and automatically sharing unused bandwidth among services. Because this does not scale to many tenants, they reduced the number of queues needed by stochastically grouping services in the same queues, changing the grouping at each switch and periodically. Unfortunately, this can potentially lead to bandwidth requirements violations. FairCloud [Pop+12] then highlighted the obvious trade-off between bandwidth allocation and work-conservation and accordingly proposed three policies based on WFQ, similarly to NetShare. Bandwidth guarantees and work-conservation can be achieved by assuming a tree topology and allocating spare capacity on the fly. Based on that, EyeQ [Jey+13] proposed a similar approach, but based on early congestion notification (ECN) and assuming a non-congested core, with per-destination rate limiters and feedback from receivers. Elasticswitch [Pop+13] followed by proposing a similar solution implemented in the hypervisor: guarantees are carefully divided among VMs and work-conservation is provided by allocating spare capacity based on traffic estimation. Several similar proposals then followed. Hadrian [Bal+13] relies on a resource reservation protocol (RSVP)-like protocol to distributedly reserve bandwidth on routers, Trinity [Hu+16] separates bandwidth guarantees and work-conservation in different priority classes, High utilization with guarantees (HUG) [Cho+16] focuses on correlated and elastic demands, eBA [Liu+16] uses explicit bandwidth information from physical links to enforce accurate rate control, and QShare [Liu+18] uses WFQ to automatically provide work-conservation and uses dynamical tenants-queue bindings to solve the queue scarcity problem. Other approaches not shown in the table and focusing on data rate allocation include [Dua14]; [Sha+14]; [TPR14]; [Lee+14]; [Kum+15]; [She+16]; [Duf+17]; [MMB17]; [PBS17]. While these approaches provide the scalability and QoS level needed for bandwidth-hungry data center applications, they do not provide strict buffer management as necessary for providing strict latency guarantees.

Another category of works focused on data center applications tries to adapt the layer-4 (L4) transport protocol used in order to reduce and/or minimize (tail) latency and/or flow completion time (FCT). Data center TCP (DCTCP) [Ali+11], one of the first proposals in this direction, uses ECN marking on switches to provide feedback to end hosts and reduce buffer usage. Deadline-driven delivery (D^3) [Wil+11] allows flows to ask for a given FCT and end hosts use this information to request rate from routers along the data path. Preemptive distributive quick (PDQ) [HCG12] approximates a range of scheduling disciplines like earliest deadline first (EDF) or shortest job first (SJF). To solve the deadline-unawareness of DCTCP and the greediness and switch customization needed by D^3, Deadline-aware datacenter TCP (D^2TCP) [VHV12] was proposed as a distributed and reactive (can update reservations) protocol that uses a novel congestion avoidance algorithm based on ECN and deadline information. The key idea behind the algorithm is that far-deadline flows back off aggressively and near-deadline back off only a little or not at all. Other proposals with the same goals but using different approaches appeared. High-bandwidth ultra-low latency (HULL) [Ali+12] marks congestion based on utilization of links rather than on buffering, DeTail [Zat+12] uses a cross-layer stack that detects congestion at lower layers to load-balance at higher layers, pFabric [Ali+13], instead of using rate control, relies on end hosts setting a priority number to each packet dependent on the flow deadline and remaining size, NDP [Han+17] trims packets when congestion happens to send the header to the receiver so that it is aware of the demand, Homa [Mon+18] relies on a receiver-driven control mechanism that tells the sender when to send, and high precision congestion control (HPCC) [Li+19] uses information from in-network telemetry to compute accurate flow rates. [Bai+16]; [ZBC19] are other similar approaches based on ECN. While these algorithms are by definition work-conserving, they minimize average latency or reduce its tail based on the set of flows in the network, rather than providing delay or even bandwidth guarantees through appropriate admission control. These protocols also require modifications to the operating system (OS) running, sometimes to the application or custom protocol or features to be implemented on the networking hardware. Also, it has been argued that L4-based optimizations do not provide robust performance isolation because these techniques rely on cooperation among the flows, and there is always someone that can misbehave [Shi+11]. Similarly to these transport protocols which provide QoS through feedback received from the DP, a wide range of other proposals, not shown in the table, build the network state by retrieving it from the DP and accordingly provide QoS guarantees [Kim+10]; [Egi+12]; [Bar+13]; [AX14]; [Ada+15]; [An+16]; [She+16]. This step adds a non-negligible delay to the flow request processing time and suffers from possible measurement errors which impedes the provisioning of hard latency guarantees. On the contrary, the approach taken in this thesis is to keep a model of the resources usage in the CP [GRK15] (see Sec. 2.3 and 2.4). The state of the network can then be retrieved from the model itself, avoiding the request processing loop to go through the DP, thereby reducing the request processing time. The model only has to communicate with the DP at topology change events and must ensure to precisely compute latency bounds through appropriate mathematical modeling (see Sec. 2.2).

Finally, a few recent efforts attempt to provide predictable latency and delay guarantees in shared network environments. These are the works closest to the focus of this thesis, as they can provide predictable latency guarantees. Time-division multiple access (TDMA) Ethernet [Vat+12] relies on an external entity to schedule transmissions (through Ethernet flow control pause and unpause frames) from the end hosts. Such an architecture that implements TDMA without the need for synchronization, suffers from low granularity (in the order of milliseconds) or imprecision that could lead to latency violations or the inability to fulfill latency requirements in the order of tens or hundreds of microseconds. Fastpass [Per+14] also proposes a TDMA approach where a centralized controller assigns timeslots to end hosts. It however requires synchronization among hosts. [Sch+16] is another example not shown in the table providing latency guarantees through TDMA. These solutions can potentially lead to an optimal utilization of resources. However, because of the need for synchronization, changes in the protocol stack of endpoints might be needed, thereby leading to expensive solutions in terms of cost and effort and potentially to delay violations if the synchronization is not accurate enough. Links as a service (LaaS) [Zah+19] suggests that existing network isolation solutions are either impractical or cannot really guarantee performance. The authors accordingly argue that most demanding tenants should be provided with exclusive access to a subset of the data network physical links. While this can provide strict latency guarantees without the need for synchronization or OS stack changes, having only one tenant per physical link drastically reduces the number of tenants that can be accommodated and, hence, the revenue for the operator. Also avoiding end hosts synchronization, QJump [Gro+15] computes latency guarantees by ensuring that each flow has at most one packet in transit in the network at any given time. Unfortunately, this prevents applications from sending bursts of data. We will see in chapter 4 that this leads to a high rejection rate and low network utilization. Silo [Jan+15], the closest work related to our contributions, applies DNC to compute guarantees. Compared to Silo, our first contribution, DetServ (Sec. 2.3 and 2.4), introduces priority queuing. The main benefit is that the several queues offer greater delay diversity, which we show in chapter 4 greatly increases the number of flows that can be accepted, i.e., network utilization. Also, Silo focuses on multi-rooted tree topologies, while we introduce routing (chapter 3) to accommodate (and leverage) any topological structure. To further increase network utilization, the *Chameleon* system proposed in chapter 4 adds the possibility of reconfiguring flows at runtime. To achieve this, source routing based on virtual local area network (VLAN) tagging is used. In chapter 5, we focus on low-cost low-capacity networks. All these systems, including DetServ and *Chameleon*, rely on assumptions that turn out to be invalid for low-cost devices and hence, as we show for Silo and QJump in Sec. 5.1, fall short to provide latency guarantees in such scenarios. Our proposed *Loko* system avoids these assumptions to provide strict latency guarantees in such networks.

Not shown in the table, several works focus on architectural issues such as interface design and requirements analysis [Kas+12]; [Sha+13]; [OD+13]; [GHG14]. In particular, the institute of electrical and electronics engineers (IEEE) time-sensitive networking (TSN) task group works on standardizing approaches and concepts for providing "*deterministic services through IEEE 802 networks, i.e., guaranteed packet transport with bounded latency, low packet delay variation, and low packet loss*" [IEE]. These works mention the need for access control and resource reservation as we describe in Sec. 2.3.4 and for a path computation unit as we investigate in chapter 3.

To conclude, we also mention that several works focus on load balancing traffic flows on different paths to improve QoS [He+15c]; [Gen+16]; [Che+16]; [Gho+17]; [Kat+17b]; [Ara+18]. We do not consider such an approach in this thesis but that is an interesting direction for future work.

2.2 Modeling Background: Deterministic Network Calculus

Network calculus is a system theory for communication networks. Based on the *min-plus* algebra, deterministic network calculus (DNC) provides a theoretical framework for analyzing performance guarantees in computer networks. Specifically, worst-case bounds on delay and buffer requirements in a network can be computed. Network calculus is said to be part of *exotic* or *tropical* algebras. These are a set of mathematical results giving insight into man-made systems (such as communication networks).

As this thesis is focused on the provisioning of strict latency guarantees, the results presented in this section focus on the computation of *deterministic* bounds, i.e., this section only deals with DNC. Another branch of DNC, namely stochastic network calculus (SNC), allows to compute worst-case performance bounds which follow probabilistic distributions. [Cha00], [JL08], [CBL05], [Fid06], [Jia06] and [Bac+92] are references covering this other branch of DNC.

The technical content of this section is mostly based on results from [LT12]. Additional developments and modifications of results for this thesis are accompanied by a [⋆] sign and, obviously, by a proof or explanation.

2.2.1 Introduction

DNC is the *system theory* that applies to computer networks. The main difference with the traditional system theory is that DNC is based on min-plus algebra, while traditional system theory is based on classical algebra. Roughly speaking, this means that the addition becomes the computation of the minimum and the multiplication becomes the addition.

Among the similarities between both theories is the usage of the *convolution* operator. In classical system theory, the convolution of an input signal by the impulse response of a system gives the output of the system. Also, the impulse response of the concatenation of a series of systems is given by the convolution of the impulse responses of all the systems. Similarly, in DNC, the so-called min-plus convolution is used to compute the output of a system or to merge nodes in series into one single node.

Nevertheless, both theories present some differences. A major one is the response of a linear system to the sum of two inputs. In classical system theory, the response to the sum of two inputs is the sum of the individual responses to each signal. In min-plus algebra, the addition corresponds to the multiplication in classical algebra and is therefore a non-linear operation. As a result, little is known on the aggregation of multiplexed flows. On the other hand, the computation of the minimum corresponds to the addition in classical algebra. Therefore, the operation is linear and the response to the minimum of two inputs is the minimum of the responses taken separately. Another difference is how non-linear systems are handled. In classical system theory, non-linear systems are linearized around their operating point and the input signals are restricted around this operating point. In nework calculus, a non-linear system is replaced by a linear system that is a lower bound for the non-linear system. This is how worst-case performance measures can be computed.

In order to provide worst-case performance bounds for flows in a network, DNC requires a model of these flows and of the network. Once this modeling is done, the derivation of the bounds is an easy task. That is the bright side of DNC. It gives easily *deterministic bounds*. However, the modeling of the flows and of the network is a complex first step. In particular, while we will see in Sec. 2.2.4 that modeling flows is usually easy, the modeling of network nodes and the extension to multiples nodes, i.e., to a network, is more complex. First, though a node is usually modeled by a simple mathematical curve or function, ensuring that this curve is a correct model for this node is not trivial. Second, the extension to multiple nodes requires to apply some complex min-plus algebra operations. Third, DNC results are usually developed for bit-by-bit systems and therefore yield bounds which are not valid for packet-based systems. The step of converting the bounds to be valid for packet-based systems is often forgotten or neglected.

The goal of this section is to overcome these three obstacles, thereby making the remainder of this thesis, which heavily relies on DNC principles, easy to understand and follow. In Sec. 2.2.2, we

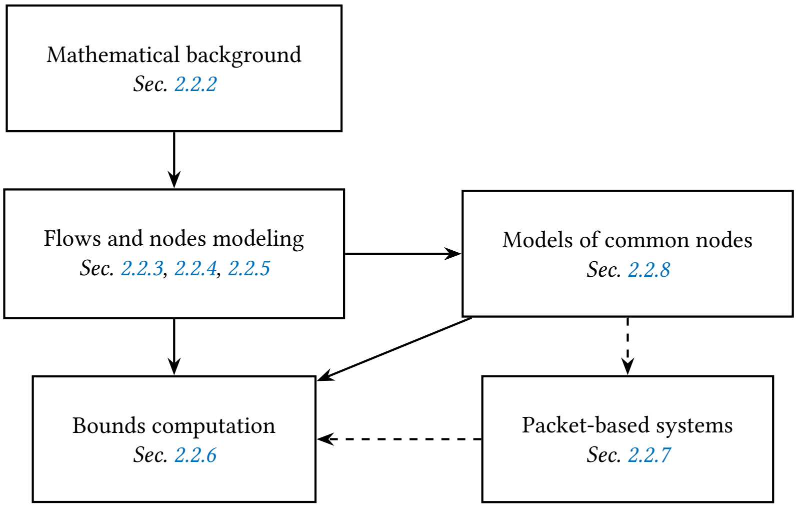


Figure 2.2: Structure of Sec. 2.2.

give a comprehensive, though classical, overview of the mathematical background. Sec. 2.2.3, 2.2.4 and 2.2.5 are then devoted to the description and explanation of how flows and network nodes are modeled. In Sec. 2.2.6, we then introduce how bounds can be derived from these models. Then, while Sec. 2.2.7 explains how the transition from bit-by-bit systems to packet-based systems can be done, Sec. 2.2.8 presents models of commonly encountered networking nodes. Thus, in combination with Sec. 2.2.7, Sec. 2.2.8 describes how to obtain models of packet-based systems. Then, with flows models as described in Sec. 2.2.4, bounds on network performance, and in particular latency, can be computed as described in Sec. 2.2.6. This structure is illustrated in Fig. 2.2.

2.2.2 Mathematical Background: Min-Plus Algebra

Before developing the results of DNC theory, we first introduce the important underlying mathematical concepts. DNC is built on top of both the min-plus and max-plus algebras. This section focuses on min-plus algebra, as the developments in this thesis only rely on min-plus algebra. [Bac+92] provides a detailed treatment of both min- and max-plus algebras. [Cha00] is an additional reference. The reader can freely skip this section and come back to it when corresponding mathematical results are used.

In conventional algebra, one usually works with the algebraic structure $(\mathbb{R}, +, \times)$ (or $(\mathbb{Z}, +, \times)$), i.e., with the set of reals (or integers) endowed with the addition and multiplication operators. The particular properties of the two operators make $(\mathbb{R}, +, \times)$ a *commutative field* and $(\mathbb{Z}, +, \times)$ a *commutative ring* [LT12, pp. 103-104].

In min-plus algebra, the addition operator is replaced by the infimum (or the minimum if it exists)[1] operator ($\wedge$) and the multiplication operator is replaced by the addition operator. $+\infty$ is also included in the set of elements on which min-plus operators can be applied. This defines another algebraic structure, $(\mathbb{R} \cup \{+\infty\}, \wedge, +)$, which verifies the following properties [LT12, p. 105].

[1] The *infimum* $\inf\{\mathcal{S}\}$ of a set $\mathcal{S}$ is defined as the greatest lower bound of $\mathcal{S}$. For example $\inf\{]0,5]\} = 0$. By convention, $\inf\{\varnothing\} = +\infty$ [LT12, pp. 103-104].
The *minimum* $\min\{\mathcal{S}\}$ of a set $\mathcal{S}$ is the element of $\mathcal{S}$ which is smaller than all its other elements. It does not always exist. For example, $]0,5]$ has no minimum [LT12, p. 103]. However, we have $\min\{[0,5]\} = \inf\{[0,5]\} = 0$.

- **Closure of $\wedge$.**
  $\forall a, b \in \mathbb{R} \cup \{+\infty\}, \quad a \wedge b \in \mathbb{R} \cup \{+\infty\}$
- **Associativity of $\wedge$.**
  $\forall a, b, c \in \mathbb{R} \cup \{+\infty\}, \quad (a \wedge b) \wedge c = a \wedge (b \wedge c)$
- **Neutral element for $\wedge$.**
  $\exists e \in \mathbb{R} \cup \{+\infty\} : \quad \forall a \in \mathbb{R} \cup \{+\infty\}, \quad a \wedge e = a \qquad (e = +\infty)$
- **Idempotency of $\wedge$.**
  $\forall a \in \mathbb{R} \cup \{+\infty\}, \quad a \wedge a = a$
- **Commutativity of $\wedge$.**
  $\forall a, b \in \mathbb{R} \cup \{+\infty\}, \quad a \wedge b = b \wedge a$
- **Closure of $+$.**
  $\forall a, b \in \mathbb{R} \cup \{+\infty\}, \quad a + b \in \mathbb{R} \cup \{+\infty\}$
- **Associativity of $+$.**
  $\forall a, b, c \in \mathbb{R} \cup \{+\infty\}, \quad (a + b) + c = a + (b + c)$
- **Neutral element for $+$.**
  $\exists u \in \mathbb{R} \cup \{+\infty\} : \quad \forall a \in \mathbb{R} \cup \{+\infty\}, \quad a + u = a = u + a \qquad (u = 0)$
- **Commutativity of $+$.**
  $\forall a, b \in \mathbb{R} \cup \{+\infty\}, \quad a + b = b + a$
- **The neutral element for $\wedge$ is absorbing for $+$.**
  $\forall a \in \mathbb{R} \cup \{+\infty\}, \quad a + e = e = e + a$
- **Distributivity of $+$ with respect to $\wedge$.**
  $\forall a, b, c \in \mathbb{R} \cup \{+\infty\}, \quad (a \wedge b) + c = (a + c) \wedge (b + c) = c + (a \wedge b)$

These axioms define a *commutative dioid*[2] [LT12, p. 105]. It is not a (commutative) ring because the $\wedge$ operator is idempotent but not cancelable [LT12, p. 105]. For example, $(\mathbb{Z}, +, \times)$ is a (commutative) ring because the $+$ operator is cancelable. It is not a (commutative) field because it does not contain a multiplicative inverse for every non-zero element. $(\mathbb{R}, +, \times)$ is therefore a (commutative) field.

2.2.2.1 Wide-Sense Increasing Functions

A function or sequence[3] f is said *wide-sense increasing* if and only if [LT12, p. 105]

$$\boxed{\forall s \geq t, \quad f(s) \geq f(t)}. \qquad \text{(Wide-Sense Increasing Function)}$$

We adopt the following notations [LT12, p. 105]. $\mathcal{G}$ is the set of non-negative wide-sense increasing functions or sequences and $\mathcal{F}$ is the set of non-negative wide-sense increasing functions or sequences such that $f(t) = 0$ if $t < 0$.

[2] It would have been a simple *dioid* if the $+$ operator was not commutative.

[3] $f(t)$ is called a *function* when its parameter t is continuous and a *sequence* when t is discrete.

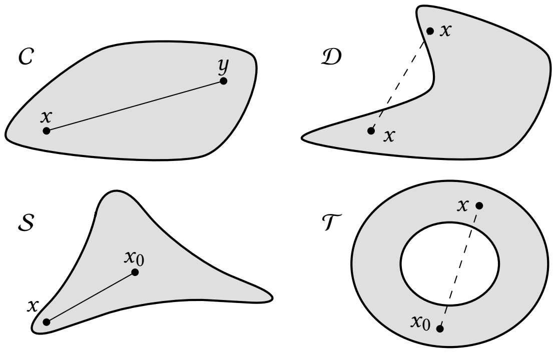


Figure 2.3: $C \subset \mathbb{R}^2$ is a convex set because the line connecting any two points of C is entirely in C. In contrast, $\mathcal{D} \subset \mathbb{R}^2$ is not a convex set. $\mathcal{S} \subset \mathbb{R}^2$ is a star-shaped set because there exist a point $x_0 \in \mathcal{S}$ such that the line segment connecting x_0 to any other point in $\mathcal{S}$ is entirely in $\mathcal{S}$. In contrast, $\mathcal{T} \subset \mathbb{R}^2$ is not a star-shaped set.

2.2.2.2 Concave, Convex and Star-Shaped Functions

Concave, convex and star-shaped functions are of particular interest in DNC. They are defined as follows.

A function $f: \mathbb{R} \longrightarrow \mathbb{R}$ is *convex* if and only if [LT12, p. 109]

$$\boxed{\forall x, y \in \mathbb{R}, u \in [0,1], \quad f(ux + (1-u)y) \leq uf(x) + (1-u)f(y)}. \qquad \text{(Convex Function)}$$

A function $f: \mathbb{R} \longrightarrow \mathbb{R}$ is *concave* if and only if $-f$ is convex or, alternatively, if and only if [LT12, p. 109]

$$\boxed{\forall x, y \in \mathbb{R}, u \in [0,1], \quad f(ux + (1-u)y) \geq uf(x) + (1-u)f(y)}. \qquad \text{(Concave Function)}$$

A function $f: [0, +\infty[\longrightarrow \mathbb{R}$ is *star-shaped* if and only if [BO62, pp. 1203-1204]-[MOA11, p. 650]

$$\boxed{\forall x \geq 0, \alpha \in [0,1], \quad f(\alpha x) \leq \alpha f(x)}, \qquad \text{(Star-Shaped Function)}$$

or, alternatively, if $f(0) \leq 0$ and $\forall x > 0, \;\; f(x)/x$ is wide-sense increasing [MOA11, p. 650]-[BO62, p. 1205].

These mathematical definitions are based on the definition of convex and star-shaped domains. A domain $\mathcal{S} \subseteq \mathbb{R}^n$ is *convex* if and only if [LT12, p. 109]

$$\boxed{\forall x, y \in \mathcal{S}, u \in [0,1], \quad ux + (1-u)y \in \mathcal{S}}. \qquad \text{(Convex Set)}$$

Intuitively, a set $\mathcal{S}$ is convex if the line segment connecting any two points of $\mathcal{S}$ is entirely in $\mathcal{S}$.

A domain $\mathcal{S} \subseteq \mathbb{R}^n$ is *star-shaped* (or is a *star-domain*) if and only if [ST83, pp. 141-142]

$$\boxed{\exists x_0 \in \mathcal{S}: \quad \forall x \in \mathcal{S}, u \in [0,1], \quad ux_0 + (1-u)x \in \mathcal{S}}. \qquad \text{(Star-Shaped Set)}$$

Intuitively, a set $\mathcal{S}$ is star-shaped if there is a point x_0 in $\mathcal{S}$ such that the line segment connecting x_0 to any other point in $\mathcal{S}$ is entirely in $\mathcal{S}$. The set is then also said *star-shaped with respect to* x_0.

Examples of such domains in $\mathbb{R}^2$ are shown in Fig. 2.3. The set C and $\mathcal{S}$ are respectively convex and star-shaped. The set $\mathcal{D}$ and $\mathcal{T}$ are respectively not convex and not star-shaped because of the

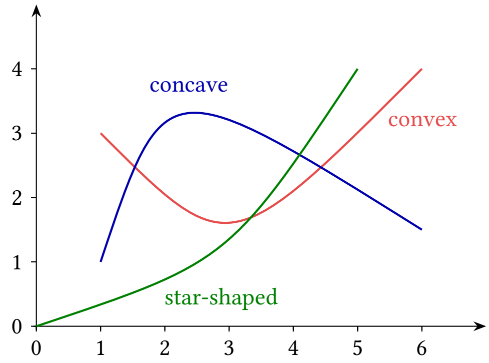


Figure 2.4: Example of concave, convex and star-shaped functions.

existence of the dashed line segment violating the definitions here above. Note that $\mathcal{D}$ and C are also star-shaped.

From this, we can give more intuitive definitions of convex, concave and star-shaped functions. A function $f : \mathbb{R} \longrightarrow \mathbb{R}$ is *convex* if and only if its epigraph[4] is a convex set [MOA11, p. 646]. A function $f : \mathbb{R} \longrightarrow \mathbb{R}$ is *concave* if and only if its opposite $-f$ is convex or, alternatively, if the set of points lying below its graph is a convex set. A function $f : [0, +\infty[\longrightarrow \mathbb{R}$ is *star-shaped* if and only if the set of points lying above its graph is star-shaped with respect to the origin [BO62, p. 1203]. Note that the star-shaped property of a function is only defined for functions whose domain is $[0, +\infty[$. We will however use it for functions defined on the whole real line by considering only the values of the function in $[0, +\infty[$.

Fig. 2.4 shows examples of such functions. These functions enjoy some interesting properties.

- The maximum (resp. minimum) of any number of convex (resp. concave) functions is a convex (resp. concave) function [LT12, p. 109].
- If $f(0) \leq 0$ and f is convex, then f is star-shaped [MOA11, p. 650]-[BO62, p. 1207]. Similarly, as can be seen in Fig. 2.3, a convex set is star-shaped.
- The maximum of two star-shaped functions is star-shaped [LT12, p. 110].

Note that in [LT12], the authors consider that a function $f \in \mathcal{F}$ is star-shaped if the set of points lying *below* its graph is star-shaped with respect to the origin. As we did not find any naming for such functions in the literature, we call them *lower-star-shaped* functions [★]. Obviously, a function f is lower-star-shaped if its opposite $-f$ is star-shaped and lower-star-shaped functions therefore enjoy properties similar to star-shaped functions:

- if $f(0) \geq 0$ and f is concave, then f is lower-star-shaped;
- the minimum of two lower-star-shaped functions is lower-star-shaped.

[4] The epigraph of a function is the set of points lying above the graph of the function.

2.2.2.3 Min-Plus Convolution

In classical system theory using classical algebra, the *convolution* of functions $f(t)$ and $g(t)$ (that are zero for $t < 0$) is defined as

$$(f * g)(t) = \int_0^t f(t-s)g(s)\, ds. \qquad \text{(Convolution)}$$

Transforming the addition into infimum and the multiplication into addition, we obtain the definition of the *min-plus convolution* for $f, g \in \mathcal{F}$.

$$\boxed{(f \otimes g)(t) = \inf_{0 \le s \le t} \{f(t-s) + g(s)\}} \qquad \text{(Min-Plus Convolution)}$$

(If $t < 0$, $(f \otimes g)(t) = 0$.)

The min-plus convolution enjoys the following properties in $\mathcal{F}$ [LT12, pp. 111-115].

- **Closure.**
  $\forall f, g \in \mathcal{F}, \quad f \otimes g \in \mathcal{F}$

- **Associativity.**
  $\forall f, g, h \in \mathcal{F}, \quad (f \otimes g) \otimes h = f \otimes (g \otimes h)$

- **Neutral element.**
  $\exists \delta_0 \in \mathcal{F}: \quad \forall f \in \mathcal{F}, \quad f \otimes \delta_0 = f \qquad (\delta_0 = +\infty$ if $t > 0$, 0 otherwise)

- **Commutativity.**
  $\forall f, g \in \mathcal{F}, \quad f \otimes g = g \otimes f$

- **Distributivity with respect to $\wedge$.**
  $\forall f, g, h \in \mathcal{F}, \quad (f \wedge g) \otimes h = (f \otimes h) \wedge (g \otimes h)$

- **Addition of a constant.**
  $\forall f, g \in \mathcal{F}, K \in \mathbb{R}^+, \quad (f + K) \otimes g = (f \otimes g) + K$

- **Isotonicity.**
  $\forall f, g, f', h' \in \mathcal{F}, \quad f \le g, f' \le g' \quad \Rightarrow \quad f \otimes f' \le g \otimes g'$

- $\forall f, g \in \mathcal{F}, \quad f(0) = g(0) = 0 \quad \Rightarrow \quad f \otimes g \le f \wedge g$

- $\forall f, g \in \mathcal{F}, \quad f(0) = g(0) = 0$ and f, g lower-star-shaped $\quad \Rightarrow \quad f \otimes g = f \wedge g$

- $\forall f, g \in \mathcal{F}, \quad f, g$ convex $\Rightarrow f \otimes g$ convex. In particular, if f and g are convex and piecewise linear, their convolution is obtained by putting E2E the different linear pieces of f and g, sorted by increasing slopes.

Graphically, the min-plus convolution of two functions $f(t) \in \mathcal{F}$ and $g(t) \in \mathcal{F}$ can be computed as shown in Fig. 2.5. Let us first consider the case where $f(0) = g(0) = 0$. In this case, the min-plus convolution is obtained by placing g at each point of f and keeping the minimum of all these functions and of f. If the functions are not zero at the origin, the *Addition of a constant* property here above shows that we can apply the construction explained above on $f(t) - f(0)$ and $g(t) - g(0)$ and add $g(0) + f(0)$ to the obtained result.

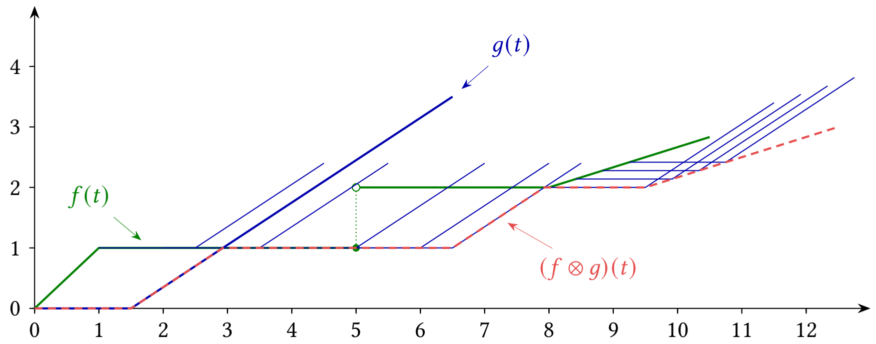


Figure 2.5: The min-plus convolution of two functions passing through the origin can be obtained by placing one of the two functions at each point of the other function and taking the minimum of all the resulting functions.

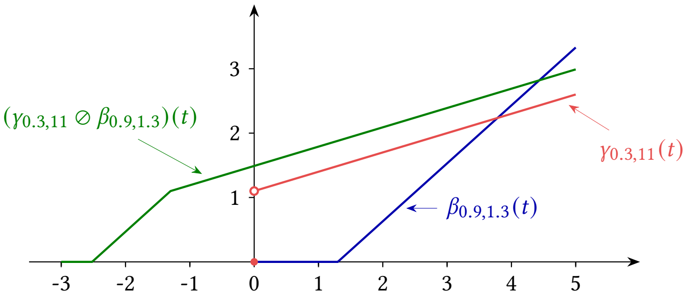


Figure 2.6: Min-plus deconvolution of $\gamma_{r,b}$ by $\beta_{R,T}$.

2.2.2.4 Min-Plus Deconvolution

The dual operation of the min-plus convolution is the *min-plus deconvolution* defined as follows for $f, g \in \mathcal{F}$[5].

$$\boxed{(f \oslash g)(t) = \sup_{u \geq 0}\{f(t+u) - g(u)\}} \qquad \text{(Min-Plus Deconvolution)}$$

If one of the function is infinite for some t, the min-plus deconvolution is undefined.

In contrast to the min-plus convolution, the min-plus deconvolution is not closed in $\mathcal{F}$ (the result is not necessarily 0 for $t \leq 0$), not commutative and not associative [LT12, p. 122]. The min-plus deconvolution of $\gamma_{r,b}$ by $\beta_{R,T}$[6] is shown in Fig. 2.6.

The min-plus deconvolution enjoys the following properties [LT12, pp. 123, 129].

- **Isotonicity.**
  $$\forall f, g, h \in \mathcal{F}, \quad f \leq g \;\Rightarrow\; f \oslash h \leq g \oslash h, \;\; h \oslash f \geq h \oslash g$$

- **Composition.**
  $$\forall f, g, h \in \mathcal{F}, \quad (f \oslash g) \oslash h = f \oslash (g \otimes h)$$

[5] The *supremum* $\sup\{\mathcal{S}\}$ of a set $\mathcal{S}$ is defined as the smallest upper bound of $\mathcal{S}$. For example $\sup\{[0,5[\} = 5$. By convention, $\sup\{\varnothing\} = -\infty$. The *maximum* $\max\{\mathcal{S}\}$ of a set $\mathcal{S}$ is the element of $\mathcal{S}$ which is bigger than all its other elements. It does not always exist. For example, $[0,5[$ has no maximum. However, we have $\sup\{[0,5]\} = \max\{[0,5]\} = 5$.

[6] These functions are defined respectively in Sec. 2.2.4.2 and 2.2.5.2.

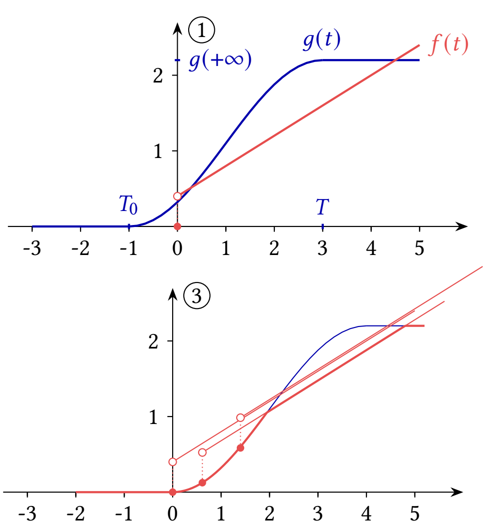


Figure 2.7: Representation of the min-plus deconvolution $g \oslash f$ by time-inversion (adapted from [LT12, p. 127]). g is first rotated around the point $\left(\frac{T}{2}, \frac{g(+\infty)}{2}\right)$ (2). Then, the result is convolved with f (3). Finally, the result of the convolution is rotated back around $\left(\frac{T}{2}, \frac{g(+\infty)}{2}\right)$ to give the final result (4).

- **Composition with $\otimes$.**

  $\forall f, g \in \mathcal{F}, \quad (f \otimes g) \oslash g \leq f \otimes (g \oslash g)$

- **Addition of a constant.**

  $\forall f, g \in \mathcal{F}, K \in \mathbb{R}^+, \quad (f + K) \oslash g = (f \oslash g) + K$

The min-plus convolution and min-plus deconvolution are said *dual* from each other because they satisfy [LT12, p. 123]

$$f \oslash g \leq h \quad \Leftrightarrow \quad f \leq h \otimes g. \tag{2.1}$$

The min-plus deconvolution of a function $g \in \mathcal{G}$ with finite lifetime by $f \in \mathcal{F}$ can be easily graphically computed. A function g is said to have a finite lifetime if there exist some finite T_0 and T such that $g(t) = 0$ for $t \leq T_0$ and $g(t) = g(T)$ for $t \geq T$. If we call $\hat{\mathcal{G}}$ the subset of functions of $\mathcal{G}$ with finite lifetime, we can show that, for $f \in \mathcal{F}$ such that $\lim_{t \to +\infty} f(t) = +\infty$ and $g \in \hat{\mathcal{G}}$, we can compute $g \oslash f$ by rotating g by 180° around $\left(\frac{T}{2}, \frac{g(+\infty)}{2}\right)$, computing the min-plus convolution with f and then rotating back again around $\left(\frac{T}{2}, \frac{g(+\infty)}{2}\right)$ [LT12, pp. 125-126]. This is called the representation of the min-plus deconvolution by *time-inversion*. The process is illustrated in Fig. 2.7.

2.2.2.5 Vertical and Horizontal Deviations

The *vertical deviation* and *horizontal deviation* between two curves in $\mathcal{F}$ are two quantities that are very often used in DNC. They are defined as follows [LT12, p. 128].

$$\boxed{h(f, g) = \sup_{t \geq 0}\{\inf_{d \geq 0}\{d : f(t) \leq g(t + d)\}\}} \tag{Horizontal Deviation}$$

$$\boxed{v(f, g) = \sup_{t \geq 0}\{f(t) - g(t)\}} \tag{Vertical Deviation}$$

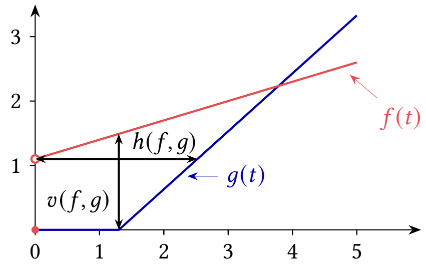


Figure 2.8: Horizontal and vertical deviation between two curves in $\mathcal{F}$.

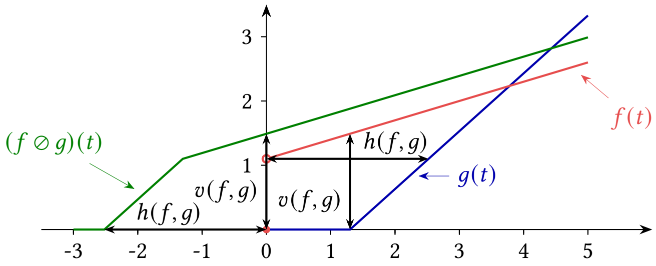


Figure 2.9: Relation between $(f \oslash g)(t)$ and $h(f,g)$ and $v(f,g)$.

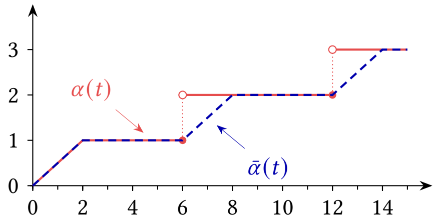


Figure 2.10: Example of an α curve and its sub-additive closure $\bar{\alpha}$ (adapted from [LT12, p. 14]).

Both quantities are shown in Fig. 2.8. It is important to note that these quantities are *not* symmetric with respect to f and g. Graphically, $h(f,g)$ corresponds to the greatest horizontal distance between the graphs of f and g *when* f is left to g and $v(f,g)$ corresponds to the greatest vertical distance between the graphs of f and g *when* f is greater than g.

The min-plus deconvolution allows to easily express these two quantities. Indeed, we can show that [LT12, p. 128]

$$v(f,g) = (f \oslash g)(0), \tag{2.2}$$

$$h(f,g) = \inf_{d \geq 0}\{d : (f \oslash g)(-d) \leq 0\}. \tag{2.3}$$

This is illustrated in Fig. 2.9.

2.2.2.6 Sub-Additivity

Sub-additive functions are another class of functions that are important in DNC. A function $f \in \mathcal{F}$ is said sub-additive if and only if [LT12, p. 116]-[Ros50, p. 227]

$$\boxed{\forall s,t \geq 0, \quad f(t+s) \leq f(t) + f(s)}. \tag{Sub-Additivity}$$

The concept is illustrated in Fig. 2.10. The α curve is not sub-additive but $\bar{\alpha}$ is.

Letting $t = t' - s$, one can easily see that an equivalent definition is $f \leq f \otimes f$. If $f(0) = 0$, we know that $f \geq f \otimes f$ and it is hence equivalent to imposing that $f = f \otimes f$.

It can be shown that the sum and min-plus convolution of two sub-additive functions are also sub-additive [LT12, pp. 117-118].

A lower-star-shaped function passing through the origin is sub-additive [LT12, p. 117]. Hence, if $f(0) = 0$, we have

$$f \text{ is concave } \Rightarrow f \text{ is lower-star-shaped } \Rightarrow f \text{ is sub-additive,} \tag{2.4}$$

but none of the reverse implications holds [BO62, p. 1207].

If $f \in \mathcal{F}$, $f \oslash f$ is a sub-additive function of $\mathcal{F}$ passing through the origin [LT12, p. 123].

Defining the *sub-additive closure* $\bar{f}$ of a function $f \in \mathcal{F}$ as[7] [LT12, p. 118]

$$\boxed{\bar{f} = \delta_0 \wedge f \wedge (f \otimes f) \wedge (f \otimes f \otimes f) \wedge \ldots \triangleq \inf_{n \geq 0}\{f^{(n)}\}}, \tag{Sub-additive Closure}$$

it can be shown that $\bar{f}$ is the greatest sub-additive function smaller than f and passing through the origin [LT12, pp. 119-120]. From this, we can easily show that, for $f \in \mathcal{F}$, [LT12, pp. 120, 125]

$$\boxed{f(0) = 0 \text{ and } f \text{ is sub-additive} \quad \Leftrightarrow \quad f \otimes f = f \quad \Leftrightarrow \quad \bar{f} = f \quad \Leftrightarrow \quad f \oslash f = f}. \tag{2.5}$$

Fig. 2.10 shows a curve α and its sub-additive closure $\bar{\alpha}$.

The sub-additive closure also enjoys the following properties [LT12, p. 120].

- **Isotonicity.**
  $\forall f, g \in \mathcal{F}, \quad f \leq g \;\Rightarrow\; \bar{f} \leq \bar{g}$
- $\forall f, g \in \mathcal{F}, \quad \overline{f \wedge g} = \bar{f} \otimes \bar{g}$
- $\forall f, g \in \mathcal{F}, \quad \overline{f \otimes g} \geq \bar{f} \otimes \bar{g}$
- $\forall f, g \in \mathcal{F} : f(0) = g(0) = 0, \quad \overline{f \otimes g} = \bar{f} \otimes \bar{g}$

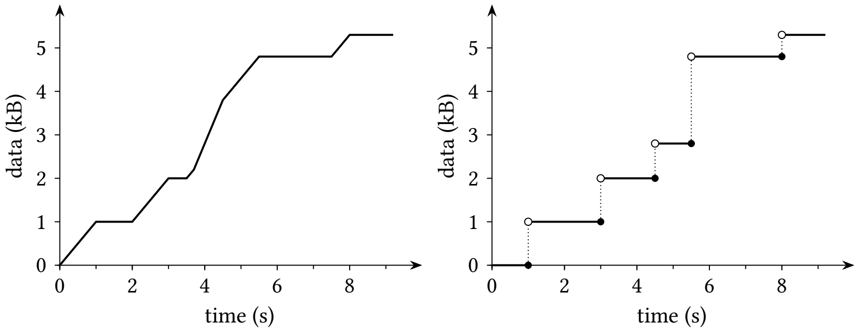
2.2.3 Data Modeling

2.2.3.1 Data Flow

A data flow is represented by a *cumulative function* $R(t)$ representing the number of bits seen on the flow during the time interval $[0, t]$. $R(t)$ is wide-sense increasing and we assume that $R(t) = 0 \;\; \forall t \leq 0$. Hence, $R(t) \in \mathcal{F}$.

Examples of $R(t)$ curves are shown in Fig. 2.11. Different models can be used. Both figures represent the same flow arriving at a given node. The flow is an aggregation of two flows coming from two different input links. From the first link, packets of length 1 kB, 0.8 kB and 0.5 kB are sent respectively after 0, 3.7 and 7.5 seconds. From the second link, packets of length 1 kB and 2 kB are sent respectively after 2 and 3.5 seconds. Both links have a transmission rate of 1 kBps. In Fig. 2.11a,

[7] δ_T is defined on page 31.



(a) Fluid model.

(b) General continuous time model.

Figure 2.11: Example of cumulative functions $R(t)$ representing data arriving at an aggregation node. From a first link, packets of length 1 kB, 0.8 kB and 0.5 kB are sent respectively after 0, 3.7 and 7.5 seconds. From a second link, packets of length 1 kB and 2 kB are sent respectively after 2 and 3.5 seconds. Both links have a capacity of 1 kBps. In the fluid model, data is observed bit-by-bit. In the general continuous time model, data is observed packet-by-packet.

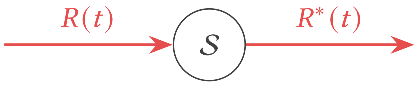


Figure 2.12: A DNC system takes data $R(t)$ as input and delivers the data $R^*(t)$ at its output.

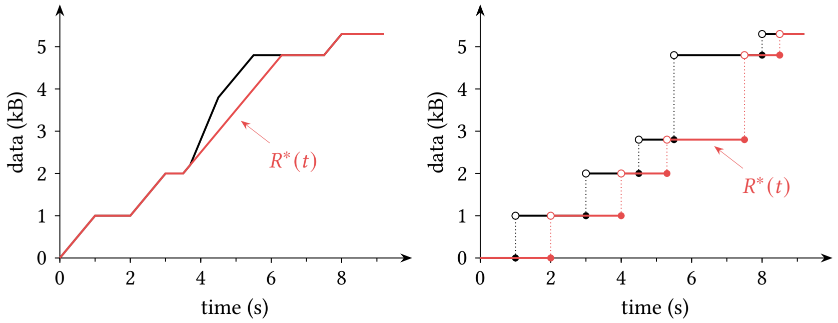
both data and time are continuous. This is called the *fluid model.* Packets are transmitted bit-by-bit. In Fig. 2.11b, time is continuous but data is discrete. Packets are considered seen only when fully received. This is the *general continuous time model.* We have to take the convention that R is either left- or right-continuous[8]. Here, we represented a left-continuous function. We will keep this convention throughout the thesis.

2.2.3.2 System

A *system* $\mathcal{S}$ (as shown in Fig. 2.12) is defined as blackbox taking data $R(t)$ (the *input function*) as input and delivering the data $R^*(t)$ (the *output function*) at its output after a variable delay. A system might represent a single buffer, a complex communication node or even a complete network. The analyzed communication network can then be represented as the interconnection of these systems, also called *network elements* [Cru91, p. 115].

For a system which serves data at a rate of 1 kBps, Fig. 2.13 shows the output functions $R^*(t)$ corresponding to the input functions shown in Fig. 2.11. In the fluid model (Fig. 2.13a), this results in sending bits as soon as they arrive at a rate of 1 kBps. In the general continuous time model (Fig. 2.13b) this results in serving a packet when it has been fully received and considering it out of the system when it has been fully sent.

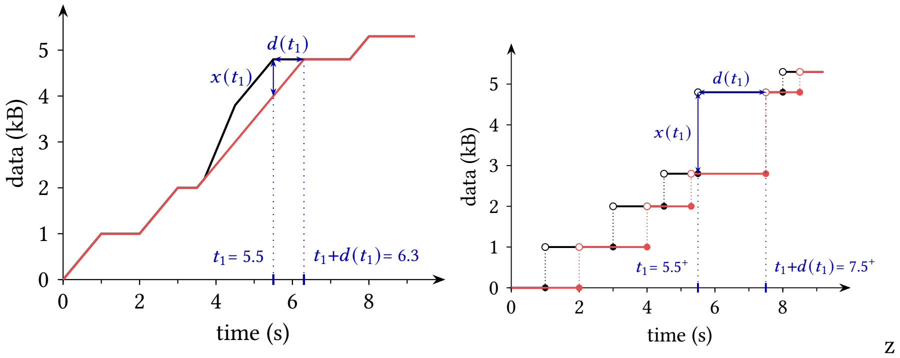
[8] Informally, a function is *left-continuous* (resp. *right-continuous*) if no jump occurs when the limit point is approached from the left (resp. right) [AE10, p. 79].



(a) Fluid model.

(b) General continuous time model.

Figure 2.13: Example of input and output functions for a system serving input data at a rate of 1 kBps.



(a) Fluid model.

(b) General continuous time model.

Figure 2.14: Graphical representation of the backlog and delay experienced by the last bit of the packet sent after 3.5 seconds. In the fluid model, the backlog and delay are smaller because the node does not have to wait for the entire packet to be received to start sending it.

2.2.3.3 Backlog and Virtual Delay

Two interesting quantities can be derived from the input and output functions [LT12, p. 5]. The *backlog* $x(t)$ corresponds to the amount of data held inside the system at time t.

$$\boxed{x(t) = R(t) - R^*(t)} \qquad \text{(Backlog)}$$

The *virtual delay* $d(t)$ corresponds to the delay that would be experienced by a bit at time t if all bits received before it are served before it[9].

$$\boxed{d(t) = \inf_{\tau \geq 0}\{\tau : R(t) \leq R^*(t+\tau)\}} \qquad \text{(Virtual Delay)}$$

Fig. 2.14 shows the graphical interpretation of these two quantities. The virtual delay (resp. the backlog) at time t corresponds to the horizontal deviation (resp. vertical deviation) between the input and output curves starting from the point $(t, R(t))$.

Interestingly, the figure shows that the backlog and virtual delay can differ from one model to the other. Indeed, the delay experienced by the last bit of the 2 kB packet is different. For the fluid model, in Fig. 2.14a, the last bit of the 2 kB packet enters the system at time $t_1 = 5.5$ and leaves it at time

[9] That is the reason why it is called *virtual* delay.

$t_1 + d(t_1) = 6.3$. The delay experienced by this bit[10] is hence $d(t_1) = 0.8$. For the general continuous time model, in Fig. 2.14b, the last bit of the 2 kB packet enters the system at time[11] $t_1 = 5.5^+$. With similar computations, we obtain a virtual delay (equal to the delay) of $d(t_1) = 2$. This is of course in accordance with the assumptions of each model. In the general continuous time model, the node must wait for the entire packet to be received to start sending it, which induces a delay. This can also be seen for all other packets. This delay corresponds to the time needed to wait for the complete packet to be fully received and fully sent.

Since most of the communication networks are nowadays packet-based, we are only interested in working with the general continuous time model. Indeed, store-and-forward switches forward a packet to an output link only when this packet is fully received and cut-through switches do so only when the destination address (or the entire header) is fully received. However, it is often easier to use the fluid model. This is not a problem. We will see in Sec. 2.2.7 how it is possible to switch from the fluid model (bit-by-bit) to the general continuous time model (packet-by-packet).

2.2.4 Arrival Curves

In order to provide performance bounds for a flow, we must have a lower bound of the service the network can offer and an upper bound of the flow characteristics. In other words, we need to know the minimum service a network guarantees to offer and the maximum amount of data a flow will send. These are respectively given by *service curves* (defined in Sec. 2.2.5), which model the network and its nodes, and by *arrival curves* (defined in this section), which model flows.

2.2.4.1 Definition

A wide-sense increasing function α is an *arrival curve* for a flow R if and only if [LT12, p. 7]-[Cru91, p. 116]

$$\boxed{\forall s \leq t, \quad R(t) - R(s) \leq \alpha(t-s)}. \qquad \text{(Arrival Curve)}$$

We also say that the flow R is α*-smooth*.

Fig. 2.15 illustrates the concept. The arrival curve constraint means that, during any time window of width τ, the amount of additional data sent by the flow is limited by $\alpha(\tau)$. Graphically, if one draws instances of the arrival curve starting at any point of the $R(t)$ curve, $R(t)$ must always remain smaller than all these instances.

From the definition of the min-plus convolution, we can show that the definition of an arrival curve is equivalent to [LT12, p. 15] $R \leq R \otimes \alpha$. From this other definition and from the isotonicity and associativity of $\otimes$, we can also show that if α_1 and α_2 are arrival curves for a flow, then so is $\alpha_1 \otimes \alpha_2$ [LT12, p. 15].

It can be shown that we can always reduce an arrival curve to be left-continuous. More precisely, an arrival curve $\alpha(t)$ can always be reduced to $\alpha_l(t) = \sup_{s<t} \alpha(s)$ without changing the set of flows for which it is an arrival curve [LT12, p. 9]. $\alpha_l(t)$ corresponds to the limit to the left of $\alpha(t)$ and is left-continuous. Since we have $\alpha_l(t) \leq \alpha(t)$, this is always a better bound for the flow.

[10] The virtual delay corresponds to the delay because bits/packets leave the system in the same order as they entered it.
[11] $t^+ = \inf_{x\in\mathbb{R}}\{x > t\}$.

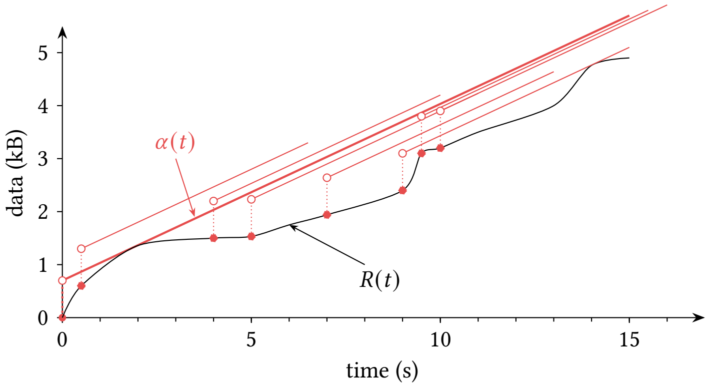


Figure 2.15: Graphical illustration of the arrival curve concept (adapted from [LT12, p. 7]). If we draw instances of the arrival curve starting at any point of the $R(t)$ curve, $R(t)$ must always remain smaller than all these instances.

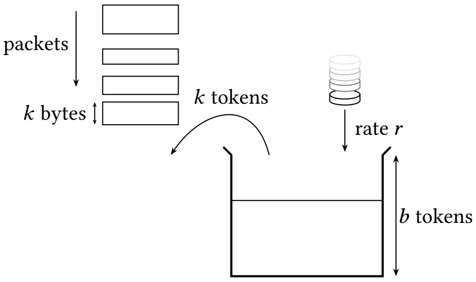


Figure 2.16: Illustration of the token bucket algorithm. When a packet of size k has to be transmitted, it removes k tokens from the token bucket. The packet is declared non-conformant if there are not enough tokens in the bucket.

If two flows R_1 and R_2 have respectively α_1 and α_2 as arrival curves, then their aggregate $R_1 + R_2$ has $(\alpha_1 + \alpha_2)$ as arrival curve [Cru91, p. 116].

2.2.4.2 Affine and Token Bucket Arrival Curves

Affine arrival curves are the most commonly used arrival curves.

$$\boxed{\gamma_{r,b}(t) = \begin{cases} rt + b & \text{if } t > 0 \\ 0 & \text{otherwise} \end{cases}} \qquad \text{(Affine Arrival Curve)}$$

b is called the *burst* and r the *rate*. This kind of arrival curve allows a source to send b bits at once, but not more than r bps over the long run. The arrival curve shown in Fig. 2.15 belongs to this family.

An affine arrival curve is closely related to the concept of *token bucket*, which defines an algorithm for determining if some data is conformant or non-conformant to some traffic policy. For this reason, arrival curves belonging to this family are also sometimes called token bucket arrival curves.

The *token bucket* algorithm [TW11, pp. 407-411] is illustrated in Fig. 2.16. A bucket of size b tokens, initially full, is filled at a rate of r tokens per second. If the bucket is full, no more tokens are added. When a packet of size k has to be transmitted into the network, it must remove k tokens from the bucket. If there are not enough tokens, the packet is declared non-conformant. Non-conformant

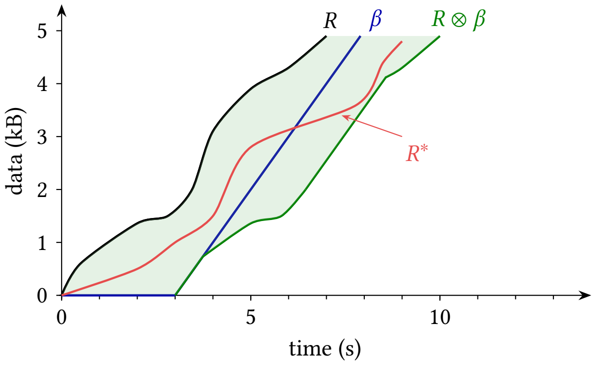


Figure 2.17: Graphical illustration of the service curve concept (adapted from [LT12, p. 19]). The output of the system must always be greater than the convolution of the input with the service curve of the system.

data can be either queued or dropped. If there are enough tokens, k tokens are removed and the packet can be transmitted.

This algorithm limits the long-term rate of a flow to r bps but allow bursts of up to b bits. The analogy with affine arrival curves is then obvious. A token bucket with filling rate r and bucket size b forces a flow to be constrained by the arrival curve $\gamma_{r,b}$ [LT12, p. 11].

2.2.5 Service Curves

After having formalized an upper bound of the flow characteristics in the previous section, we now define a lower bound of the service the network can offer. The details of how packets are handled by a node or network are abstracted using the concept of *service curve*, which we define in the current section. The combination of arrival and service curves will then be used in the next sections to compute performance bounds.

The service curve concept defined in Sec. 2.2.5.1 is the classical service curve concept used in DNC. Other types of service curves can be defined. We also introduce strict service curves in Sec. 2.2.5.3. In case of ambiguity, the concept defined in Sec. 2.2.5.1 is alternatively referred to as *classical service curve*. [BJT09] provides a more detailed treatment of service curves in DNC.

2.2.5.1 Definition

A system $\mathcal{S}$, with input and output functions R and R^*, offers a service curve β to R if and only if β is wide-sense increasing, $\beta(0) = 0$ and [LT12, p. 19]

$$\boxed{R^* \geq R \otimes \beta}, \tag{Service Curve}$$

or, alternatively,

$$R^*(t) \geq \inf_{s \leq t}\{R(s) + \beta(t-s)\}. \tag{2.6}$$

Fig. 2.17 illustrates this definition. Given R and the service curve β offered by a system $\mathcal{S}$, we can compute $R \otimes \beta$. The output R^* of the system $\mathcal{S}$ will lie in the area (shaded in the figure) between[12] R and $R \otimes \beta$. An example of possible output is also shown in the figure.

[12] For causality reasons, we must also have $R^* \leq R$.

This service curve concept indeed provides a lower bound on the service a system $\mathcal{S}$ can offer. In particular, if a system $\mathcal{S}$ pretends to offer a service curve β, we can be sure that, if we provide R at the input of $\mathcal{S}$, its output will be at least $R \otimes \beta$.

2.2.5.2 Common Service Curves

Burst-delay and *rate-latency* functions are the functions most commonly used as service curves. They are defined as follows.

$$\boxed{\delta_T(t) = \begin{cases} +\infty & \text{if } t > T \\ 0 & \text{otherwise} \end{cases}} \tag{Burst-Delay Function}$$

$T > 0$ is called the *delay*. As convolving a function with δ_T shifts it by T to the right, this kind of service curve is used to model a first-in first-out (FIFO) node imposing a delay $d \leq T$ to bits passing through it.

$$\boxed{\beta_{R,T}(t) = R[t-T]^+ = \begin{cases} R(t-T) & \text{if } t > T \\ 0 & \text{otherwise} \end{cases}} \tag{Rate-Latency Function}$$

$T \geq 0$ is called the *delay* and $R \geq 0$ the *rate*. This type of service curve is very commonly used for modeling scheduling algorithms and will be main service curve type used in this thesis. Bits might have to wait up to T before being served with a rate greater or equal to R. The service curve shown in Fig. 2.17 is a rate-latency service curve.

2.2.5.3 Strict Service Curve

A system $\mathcal{S}$ offers a *strict service curve* β to a flow if, during any backlogged period[13] of duration τ, the output of the system is at least equal to $\beta(\tau)$ [LT12, p. 21]. Mathematically[14] [BJT09, p. 6]

$$\boxed{\forall \text{ backlogged period }]s,t], \quad R^*(t) - R^*(s) \geq \beta(t-s)}. \tag{Strict Service Curve}$$

The strict service curve property allows to guarantee service during any backlogged interval, which is not possible with the classical service curve property [LT12, p. 196]. Indeed, if for some interval of time, a server gave a service higher than announced by its service curve, the classical service curve property allows the server to be lazy afterwards. This is illustrated in Fig. 2.18. The server gave a high service between times 0 and 2 and can hence be lazy (nearly delivering no service) between times 2 and 6.5 though there is some backlogged data in the system (since $R > R^*$). On the other hand, a node offering a strict service curve β guarantees that, for any backlogged period of length τ, it will output at least $\beta(\tau)$ amount of data.

As its name suggests, the strict service curve property is more strict than the service curve property. This means that if a node offers β as a strict service curve to a flow, then it also offers the same curve β as a *classical* service curve to the flow [LT12, p. 22].

[13] A backlogged period is an interval of time I (can be closed, semi-closed or open) during which the backlog is non-null, i.e., $\forall u \in I: \ R(u) - R^*(u) > 0$ [BJT09, p. 5].

[14] If β is assumed left-continuous, changing $]s,t]$ to $]s,t[$ in the definition does not change the meaning of it [BJT09, pp. 6-7].

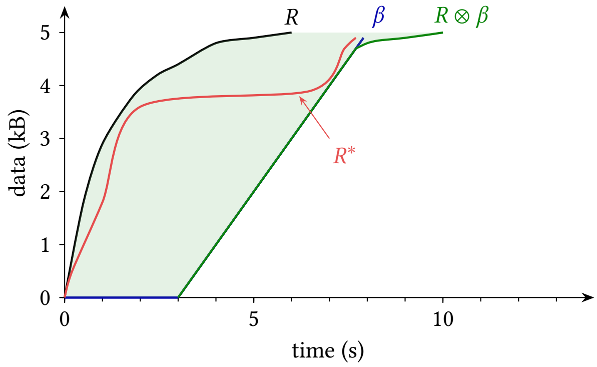


Figure 2.18: The service curve property does not provide guarantees over any interval of time (adapted from [LT12, p. 196]).

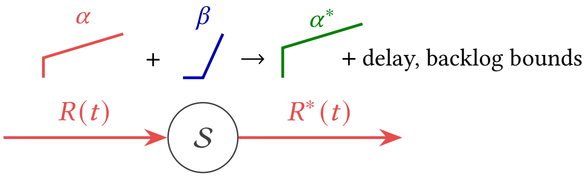


Figure 2.19: From arrival and service curves, DNC allows to compute bounds on the delay a flow will experience, on the backlog a flow will cause and on the new arrival curve of the flow at the output of the system.

2.2.5.4 Concatenation

Assuming that a flow traverses systems $\mathcal{S}_1$ and $\mathcal{S}_2$ in sequence, if the systems offer respectively the service curves β_1 and β_2 to the flow, then the concatenation of the two systems offers a service curve $\beta_1 \otimes \beta_2$ to the flow [LT12, p. 28]. Hence, if we note $\beta(\mathcal{S})$ the service curve offered by a system $\mathcal{S}$ and use $\circ$ to denote the concatenation of two systems, we have

$$\boxed{\beta(\mathcal{S}_1 \circ \mathcal{S}_2) = \beta(\mathcal{S}_1) \otimes \beta(\mathcal{S}_2)}. \qquad \text{(Concatenation)}$$

Note that if β_1 and β_2 are strict service curves, $\beta_1 \otimes \beta_2$ is not necessarily strict [BJT09, pp. 13-14].

In particular, the concatenation of two rate-latency servers β_{R_1,T_1} and β_{R_2,T_2} results in a rate-latency server $\beta_{\min\{R_1,R_2\},T_1+T_2}$ [LT12, p. 28]. This property can easily be generalized to any number n of rate-latency servers β_{R_i,T_i}: their concatenation is a rate-latency server $\beta_{\min\{R_1,\ldots,R_n\},T_1+\ldots+T_n}$.

2.2.6 Bounds

Armed with the concepts of arrival and service curves, we are now able to develop the main results of DNC. After having introduced the mechanisms to compute the worst-case bounds in Sec. 2.2.6.1–2.2.6.3, we present in Sec. 2.2.6.4 the particular values in commonly encountered cases.

From the arrival curve α of a flow and the service curve β of a node, DNC theory allows to compute an upper bound of the backlog generated by the flow at this node (Sec. 2.2.6.1), the virtual delay the flow will experience at the node (Sec. 2.2.6.2), and the new arrival curve α^* of the flow at the output of the node (Sec. 2.2.6.3). This is illustrated in Fig. 2.19.

2.2.6.1 Backlog Bound

From the definitions of service and arrival curves, it can be shown that the backlog $x(t)$ at a node offering a service curve β to a flow with arrival curve α is such that [LT12, pp. 22-23]

$$\boxed{x(t) \leq v(\alpha, \beta)}, \tag{Backlog Bound}$$

i.e., is bounded by the vertical deviation between the arrival and service curves.

2.2.6.2 Delay Bound

Similarly, it can be shown that the virtual delay $d(t)$ experienced by a flow with arrival curve α at a node offering a service curve β is such that [LT12, p. 23]

$$\boxed{d(t) \leq h(\alpha, \beta)}, \tag{Delay Bound}$$

i.e., is bounded by the horizontal deviation between the arrival and service curves.

2.2.6.3 Output Flow Bound

After having traversed a node, a flow R is transformed into R^*. One might wonder then what could be an arrival curve for R^*. As data of the flow can be buffered at the node before being served, we expect the burst of the flow to increase when traversing a node. It can be shown that an α-smooth flow traversing a node with service curve β gets out of the node with an arrival curve α^* given by [LT12, pp. 23, 35–36]

$$\boxed{\alpha^* = \alpha \oslash \beta.}. \tag{Output Flow}$$

Since traffic exiting a system might be input traffic to a second system, the knowledge of this output bound allows to analyze this second system and hence to perform a network-wide worst-case analysis.

$\alpha^*(0^+)$ corresponds to the maximum amount of data of the flow the node will output in an infinitely small amount of time, i.e., to the new maximum burst of the flow. Since $(\alpha \oslash \beta)(0) = v(\alpha, \beta)$ (see Sec. 2.2.2.5), we see that the new maximum burst of the flow corresponds to the maximum backlog that can be observed at the node. This is intuitive. Indeed, all data that accumulates in the node can be released instantly as a burst if the node eventually receives an infinite service, which the service curve concept allows.

Recall that the min-plus deconvolution is not closed in $\mathcal{F}$. Therefore $\alpha^*(t)$ is not necessarily in $\mathcal{F}$. However, a bound in $\mathcal{F}$ for the output flow can hence be obtained by setting $\alpha^*(t) = 0 \ \ \forall t \leq 0$.

A major drawback of the output bound is that the deconvolution operation is not easy to perform in the general case. [Cru91, p. 117] shows that an output bound α^* can be obtained from the delay bound $\bar{D}$ at a node and is given by

$$\alpha^*(t) = \alpha(t + \bar{D}). \tag{2.7}$$

This means that the arrival curve of a flow after a node can be obtained by shifting to the left its initial arrival curve by the maximum delay the flow could experience at this node. This result is useful when either *(i)* the arrival and service curves combination leads to intractable or complicated computations, or *(ii)* the maximum delay is known but not the service curve.

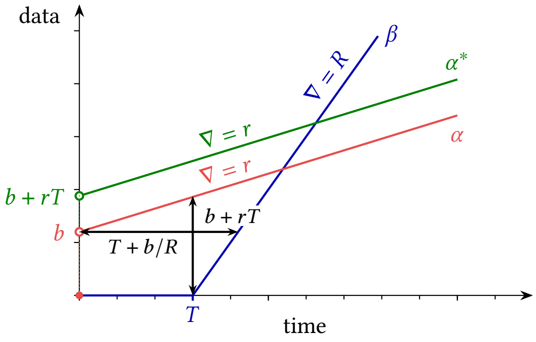


Figure 2.20: Graphical illustration of the computation of the delay, backlog and output bounds for a token bucket flow traversing a rate latency server (adapted from [LT12, p. 24]).

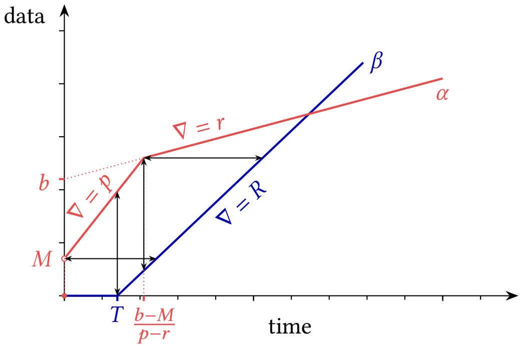


Figure 2.21: Graphical illustration of the computation of the delay and backlog bounds for a variable bit rate (VBR) flow traversing a rate latency server (adapted from [LT12, p. 25]).

2.2.6.4 Common Bounds Results

In this section, we present the bounds values for two commonly encountered arrival-service curves combinations.

Token bucket flow through a rate latency server. Considering a flow constrained by a token bucket arrival curve $\gamma_{r,b}$ traversing a node offering a service curve $\beta_{R,T}$, we obtain the following bounds (for $r \leq R$), shown in Fig. 2.20 [LT12, p. 24].

$$x(t) \leq b + rT \tag{2.8}$$

$$d(t) \leq T + \frac{b}{R} \tag{2.9}$$

$$\alpha^*(t) = \gamma_{r,b+rT} \tag{2.10}$$

VBR Flow through rate latency server. We call a constant bit rate (CBR) connection a flow constrained by a token bucket arrival curve. We then call a VBR connection a flow constrained by two token buckets in series [LT12, p. 13], i.e., by an arrival curve of the type

$$\alpha(t) = \alpha_1(t) \otimes \alpha_2(t) = (M + pt) \otimes (b + rt) = \min\{M + pt, b + rt\}. \tag{2.11}$$

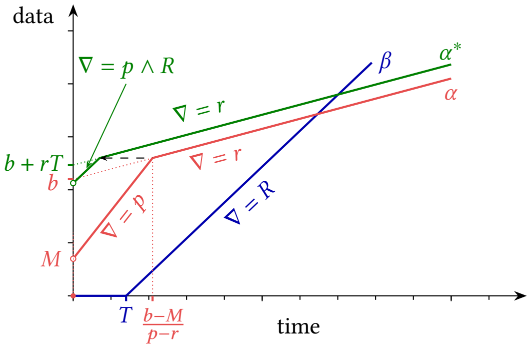


Figure 2.22: Graphical illustration of the computation of the output bound for a VBR flow traversing a rate latency server.

We assume that $r \leq R$, $M \leq b$ and $p \geq r$. Such an arrival curve is shown in Fig. 2.21. The Internet integrated services (IntServ) framework [BCS94] uses this family of arrival curve. Considering such a flow traversing a node offering a service curve $\beta_{R,T}$, we obtain the following bounds, shown in Fig. 2.21 and 2.22 [LT12, pp. 24-25].

$$x(t) \leq b + rT + \left(\frac{b-M}{p-r} - T\right)^+ ((p-R)^+ - p + r) \tag{2.12}$$

$$d(t) \leq T + \frac{M + \frac{b-M}{p-r}(p-R)^+}{R} \tag{2.13}$$

$$\alpha^*(t) = \begin{cases} \text{if } \frac{b-M}{p-r} \leq T & b + r(T+t) \\ \text{otherwise} & \min\{b + r(T+t),\ \ (t+T)(p \wedge R) + M + \frac{b-M}{p-r}(p-R)^+\} \end{cases} \tag{2.14}$$

From the convexity and linearity of the region between α and β, the maximum horizontal and vertical deviations can only be reached at angular points of either α or β. From this, only two values are possible for both the delay and backlog bounds [LT12, p. 24]. These are shown in Fig. 2.21. Some algebra then leads to the formulas given above.

2.2.7 Packet-Based Systems

The developments so far considered continuous data flows. However, packet-switched systems send data per-packet, rather than bit-by-bit. We show in this section how it is possible to quantify the irregularities introduced by packetization in order to still use the concepts introduced in the previous sections.

2.2.7.1 The Packetizer

Let us consider $L(n)$ ($n \in \mathbb{N}$), the wide-sense increasing sequence of cumulative packet lengths. We then define the following building block [LT12, p. 41]

$$\boxed{P^L(x) = \sup_{n \in \mathbb{N}}\{L(n) : \ \ L(n) \leq x\}}, \tag{Function P^L}$$

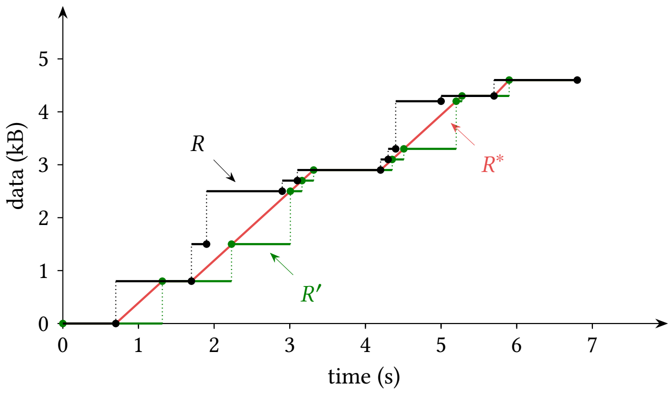


Figure 2.23: Modeling of a real variable length packet trunk with constant bit rate as a $\beta = c$ system (input R, output R^*) followed by a packetizer (input R^*, output R') (adapted from [LT12, p. 40]).

which can be alternatively defined by [LT12, p. 41]

$$P^L(x) = L(n) \quad \Leftrightarrow \quad L(n) \leq x < L(n+1), \tag{2.15}$$

i.e., $P^L(x)$ is the largest cumulative packet length that is entirely contained in x. can easily be seen that the function is right-continuous [LT12, p. 41]. An *L-packetizer* is defined as the system that transforms $R(t)$ into $P^L(R(t))$ [LT12, p. 41]. A flow R is then said *L-packetized* if $P^L(R(t)) = R(t)\ \forall t$ [LT12, p. 41]. The packetizer enjoys the following properties [LT12, p. 42].

- **Isotonicity.**
  $\forall x, y \in \mathbb{R}, \quad x \leq y \quad \Rightarrow \quad P^L(x) \leq P^L(y)$
- **Idempotency.**
  $\forall x \in \mathbb{R}, \quad P^L(P^L(x)) = P^L(x)$
- **Optimality.**
  Among all flows such that
  $$\begin{cases} x \text{ is L-packetized} \\ x \leq R \end{cases}$$
  $P^L(R(t))$ is the upper-bound.

2.2.7.2 Impact of the Packetizer

Fig. 2.23 shows an example of L-packetized input flow R along with the corresponding bit-by-bit output R and L-packetized output R' if the flow traverses a $\beta = c$ system. This corresponds to the correct modeling of a real variable length packet trunk with constant bit rate. We see that the delay and backlog bounds are bigger than those that we would have obtained considering only R and R^*. The following quantifies this deviation.

Consider a bit-by-bit system with L-packetized input R, bit-by-bit output R^* and with service curve β. The output R^* is then L-packetized to produce the final output R'. If the systems are FIFO and lossless, we have the following results[15] [LT12, pp. 42-44].

[15] $l_{max} = \sup_n\{L(n+1) - L(n)\}$.

- The per-packet delay[16] for the combined system is the maximum virtual delay for the bit-by-bit system.
- The service curve β' (from which the maximum backlog for the combined system can be computed) is given by
$$\beta'(t) = [\beta(t) - l_{max}]^+, \tag{2.16}$$
- If a flow S has $\alpha(t)$ as an arrival curve, then $P^L(S(t))$ has $\alpha(t) + l_{max} 1_{\{t>0\}}$ as an arrival curve.

The second point is consistent with Fig. 2.23 while the third one is consistent with the observation made in the introduction of this section. The first point can be interpreted as follows. The packetizer waits for the last bit of a packet to consider the first bits transmitted. Therefore, the packet itself is not delayed, since it is fully received at the same time. However, downstream nodes will have to wait for the entire packet to be received before being able to process it. The processing of the packet is then delayed. Packetizers hence do not increase the maximum delay at the node where they are appended but they, however, generally increase the E2E delay [LT12, p. 45].

That is why, for E2E delay bound calculations, the packetizer at the last hop can be neglected. Consider for example [LT12, p. 44] the concatenation of m generalized processor sharing (GPS) nodes with rate R, each node being followed by an L-packetizer. Each GPS node followed by its associated L-packetizer offers (from the result hereabove) a service curve $\beta_{R,\frac{l_{max}}{R}}$. The complete system hence offers a service curve $\beta_{R,m\frac{l_{max}}{R}}$. However, to compute the E2E delay bound, we can neglect the last packetizer and consider the service curve $\beta_{R,(m-1)\frac{l_{max}}{R}}$. If the flow is originally constrained by $\gamma_{r,b}$, the E2E delay bound is hence
$$\frac{b + (m-1)l_{max}}{R}.$$

2.2.8 Service Curves for Common Nodes

In the previous sections, we have seen what bounds DNC allows us to compute based on arrival and service curves. Most flows can easily be modeled by an affine arrival curve. We now need to know how to obtain the service curve corresponding to a physical node. This issue is addressed in this section. We will then be able to model a flow with its arrival curve, a node (and a network) with its service curve, and hence to compute bounds in real scenarios.

2.2.8.1 Constant Delay Line

A *constant delay line* is a network element that outputs all data which arrives on its single input on its single output stream exactly T seconds later [Cru91, p. 117]. This corresponds to a node with a service curve given by δ_T. The maximum delay experienced at this node is given by T. The output arrival curve α^* is equal to the initial arrival curve α [Cru91, p. 117] and the backlog is bounded by $\alpha(T)$. This is shown in Fig. 2.24.

This type of element can be used to model propagation and processing delays [Cru91, p. 117]. As these elements do not modify the arrival curve of flows, they can usually be omitted in the modeling.

[16] For a system with L-packetized input and output, the per-packet delay is $\sup_i\{T_i' - T_i\}$ where T_i' and T_i are the arrival and departure times for the ith packet.

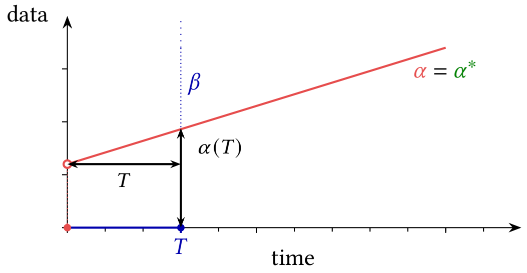


Figure 2.24: Bounds at a constant delay line network element.

The E2E delay bound of a flow can then be obtained by computing the delay bound without these elements and adding their delay contribution to the result.

Since $\delta_T - l = \delta_T \;\; \forall l$, this model is valid both for bit-by-bit and packet-by-packet systems [★].

2.2.8.2 First-In First-Out (FIFO)

The simplest form of packet scheduling is first-in first-out (FIFO). Packets are served in order of arrival. The different flows are not isolated. If the node is serving the flows at a rate r, it offers a service curve $\beta = rt$ to the aggregate flow. The virtual delay and buffer bounds are the same for all the flows and must be computed with the arrival curve of the aggregate flow [LT12, pp. 67-68].

The virtual delay computed in such a way does not correspond to the real delay because a FIFO node is not necessarily FIFO per bit. Indeed, since packets are considered only when fully received, a bit belonging to a small packet might be sent before a bit arrived earlier but belonging to a big packet. For the virtual delay bound to correspond to the per-packet delay bound, we must have an L-packetized input [★]. Indeed, in such a case, bits belonging to the same packet are seen all at the same time and both packets and bits are hence transmitted FIFO. Then, to reflect packetization at the output, the service curve has to be changed to $\beta = [rt - l_{max}]^+$.

2.2.8.3 Priority Queuing (PQ)

Under priority queuing (PQ), packets arriving at the output link are classified into priority classes. When a packet is to be transmitted, a packet from the highest priority class with the non-empty queue is chosen (the choice among packets in the same priority queue is generally done FIFO). Under *non-preemptive* priority queuing (PQ) scheduling, the transmission of a packet is not interrupted once it has begun. In constrast, a *preemptive* PQ scheduler would stop the transmission of a packet if a packet of higher priority arrives at the node and immediately sent the latter [KR13, pp. 642-643].

Consider a non-preemptive priority scheduler serving two flows H and L. The node offers a *strict* service curve β to the aggregate of the two flows. H has priority over L. If H is α_H-smooth and l^L_{max} is the maximum packet size of L, the service curves guaranteed to H and L are [LT12, p. 176]

$$\boxed{\beta_H = [\beta - l^L_{max}]^+} \tag{PQ – High Priority}$$

and

$$\boxed{\beta_L = [\beta - \alpha_H]^+} \tag{PQ – Low Priority}$$

if these are wide-sense increasing. β_H is also a strict service curve [BJT09, pp. 19-20]. β_L is not strict but $\beta'_L = [\beta - \alpha_H - l^L_{max}]^+$ is [BJT09, pp. 19-20].

For example, if the node serves the aggregate at a rate C and if H is $\gamma_{r,b}$-smooth ($r < C$), then [LT12, p. 21]

$$\beta_H = \beta_{C,\frac{l^L_{max}}{C}}, \tag{2.17}$$

$$\beta_L = \beta_{C-r,\frac{b}{C-r}}. \tag{2.18}$$

The latency $\frac{l^L_{max}}{C}$ of β_H accounts for the fact that, since the scheduler is non preemptive, H might have to wait for a complete L packet to be sent. The latency $\frac{b}{C-r}$ of β_L is the time needed to empty the buffer of the high priority queue. Indeed, the buffer might be instantly containing b and then filled at rate r. If emptied at a rate C, the buffer will finally be empty at time $t^* : \quad b + rt^* - Ct^* = 0$, i.e., at time $\frac{b}{C-r}$. After this time, L can be served at a rate $C - r$ since r is still used to serve H. In this example, both service curves are strict [LT12, p. 22].

Generalizing to n classes, if C is the overall capacity of the server and α_i the arrival curve for class i (class 1 is the highest priority) then the service curve for class i is [Sch+03, p. 4170]

$$\beta_i(t) = \left(Ct - \sum_{j=1}^{i-1} \alpha_j(t) - \max_{i+1 \leq j \leq n} \{l^j_{max}\} \right)^+. \tag{2.19}$$

From the results above, taking the maximum over $i \leq j \leq n$ makes it strict [BJT09, p. 20].

In the particular case where $\alpha_i = \gamma_{r_i,b_i}$ (i.e., classes are token bucket flows) the service curve for class i is given by [Sch+03, p. 4171] $\beta_i = \beta_{R_i,T_i}$ where

$$R_i = C - \sum_{j=1}^{i-1} r_j, \tag{2.20}$$

$$T_i = \frac{\sum_{j=1}^{i-1} b_j + \max_{i+1 \leq j \leq n}\{l^j_{max}\}}{C - \sum_{j=1}^{i-1} r_j}. \tag{2.21}$$

From the results of Sec. 2.2.6, the delay and backlog experienced by class i are then bounded by [Sch+03, pp. 4171-4172]-[Cru91, pp. 122-123]

$$d_i(t) \leq d_i = \frac{\sum_{j=1}^{i} b_j + \max_{i+1 \leq j \leq n}\{l^j_{max}\}}{C - \sum_{j=1}^{i-1} r_j}, \tag{2.22}$$

$$x_i(t) \leq x_i = b_i + r_i \left(\frac{\sum_{j=1}^{i-1} b_j + \max_{i+1 \leq j \leq n}\{l^j_{max}\}}{C - \sum_{j=1}^{i-1} r_j} \right), \tag{2.23}$$

and the new burst of the class after having traversed the scheduler is given by

$$b^*_i = b_i + x_i \tag{2.24}$$

while its rate is unchanged.

Note that these formulas do not consider the packetization of the output of the scheduler. To do so, each obtained service curve β must be transformed as explained in Sec. 2.2.7.2.

2.2.8.4 Flows Aggregation With Strict Service Curve Element

In the previous section, we have shown how to compute the service curve offered to different flow classes at a scheduler. However, we know that several flows can be classified into the same class. Hence, the service curves computed might be offered to an aggregate of flows. In order to be able to obtain bounds for the individual flows and not only for the aggregate, we try in this section to derive the service curve offered to the individual flows of an aggregate. Unfortunately, the SoA dealing with aggregate multiplexing is not very rich [LT12, p. 175].

Without loss of generality, we only consider two flows.

Consider two flows being served, with some unknown arbitration between the two flows, by a node guaranteeing a *strict* service curve β. If flow 2 is α_2-smooth, then, if it is wide-sense increasing,

$$\boxed{\beta_1 = [\beta - \alpha_2]^+} \qquad \text{(Residual Service Curve – Strict Service Curve Node)}$$

is a service curve for flow 1 [LT12, p. 176]-[BJT09, pp. 17-18]. This is when we do not know anything about the scheduling between flows (also called *blind multiplexing*). If flow 2 has priority over flow 1, the service curve is strict [BJT09, pp. 17-19]. This shows that there is a difference between blind multiplexing and fixed priorities, though the worst departure process for blind multiplexing is fixed priorities. [BJT09, p. 19] provides a nice example to show how this is possible.

For example, considering a node with strict service curve $\beta_{R,T}$ serving two token bucket flows with parameters (r_i, b_i), we have that [LT12, pp. 176-177]-[Cru91, pp. 121-122][17], if $r_1 + r_2 \leq R$, the output of flow 1 is a token bucket with parameters $r_1^* = r_1$ and

$$b_1^* = b_1 + r_1 T + r_1 \frac{b_2 + r_2 T}{R - r_2}.$$

2.3 DetServ: Architecture Design

In this section, we present the architecture that forms the basis of this thesis. First, in Sec. 2.3.1, we define precisely the E2E QoS metric considered: delay. Then, in Sec. 2.3.2, we define the scenario and problem we aim to solve. Sec. 2.3.3 then defines an augmented network topology on which we perform path finding and Sec. 2.3.4 defines the different architectural components needed for a routing algorithm to be able to solve embedding requests. Finally, Sec. 2.3.5 then accordingly summarizes the interface that these components provide to the routing procedure.

Sec. 2.4 then describes how the interface of the network model is implemented. Sec. 2.6 briefly discusses how this architecture can be extended to include wireless hops. Chapter 3 deeply investigates the routing procedure. The resource allocation problem, for its part, is not extensively studied in this thesis. We will use simple algorithms. The investigation of this problem is an interesting future research direction that we discuss.

[17] This reference considers $T = 0$ and adds an additional parameter V which is 0 in our case.

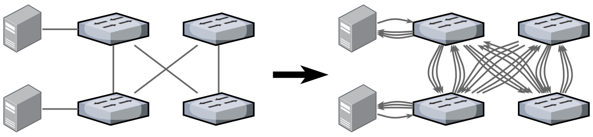


Figure 2.25: Example of queue-level topology for a network with 3 priority queues per link. Modeling each individual priority queue leads to more embedding opportunities for the routing procedure.

2.3.1 Parameter Considered: End-to-End Delay

There are several E2E QoS parameters that can be considered for predictability, e.g., packet loss, jitter, delay, throughput or availability [ÅGB11]; [Gun+11]; [KK13]; [Sta15]. As the main focus of this thesis, this architecture considers E2E delay.

Along its path, a packet suffers from different types of delays: processing, queuing, transmission, and propagation delays [KR13]. The propagation delay for each link is known from the characteristics of the link (e.g., its length and physical medium). The processing delay depends on the properties of the switching hardware and can usually be bounded by a constant. We will investigate this component in chapter 4. Finally, upper bounds on the queuing and transmission delays can be computed using DNC. The sum of all these components along the route of a flow makes up the E2E worst-case delay bound for the flow. We do not consider delay introduced at end hosts.

2.3.2 Problem Formulation: Online Flow Embedding

We consider an online setting where users or applications arrive one at a time and request a path with predictable latency. The offline counterpart, consisting in finding an embedding for a set of flows, has been shown to be intractable [GRK15]; [GRK16]; [Guc18]. Each flow should be embedded such that its latency requirement is fulfilled and such that its consumption of resources is minimized, so as to maximize the probability of acceptance of future requests. However, of course, when processing a flow request, the amount and characteristics of flow requests that will follow is unknown. As such, our problem is the following.

For a given flow f, find a route through the topology from its source s_f to its destination d_f such that (i) *the E2E delay requirement t_f of the flow is satisfied,* (ii) *the E2E guarantees provided to previously embedded flows are still guaranteed, and* (iii) *the probability of future flow requests acceptance is maximized.*

QoS routing is initiated by a query of the DP. For example, this can be done by contacting the NBI of the SDN controller. The query should at least contain the application characteristics (e.g., source, destination, burst, rate and maximum packet size) and maximum E2E delay requirement. Based on this input and on the current state of the network, routing can then be performed. Once an embedding is found, the corresponding forwarding rules are pushed to the DP by the controller.

2.3.3 Routing Topology: Queue-Level Topology

The queuing delay a packet experiences on its way to its destination does not only depend on the physical path followed by the packet but also on how the packet is scheduled at each link. Because of its simplicity and ubiquity, we assume that non-preemptive strict priority scheduling is used at each switch.

From this, the route selection process for a flow must consider both the physical links the flow will traverse and the queues at which the flow will be buffered at each output link. As a consequence, Guck et al. [GRK15]; [GRK16] introduced what we will call the *queue-level* topology. From the physical network topology, each directed physical link (u, v) is replaced by $Q_{u,v}$ queue links, where u and v are the source and destination nodes of the link and $Q_{u,v}$ is the number of priority queues at the scheduler of the link (see Fig. 2.25). Each edge in the queue-level topology hence represents a physical link and a given queue at the ingress of this physical link, i.e., a different QoS level transmission over this physical link. Route selection on this queue-level topology thus determines both the path that a flow takes through the physical network as well as, for each physical link, the queue in which the flow will be buffered.

From a set of flows and the paths they follow in the queue-level topology, the DNC results presented in Sec. 2.2 allow to compute E2E delay bounds for each flow.

Performing route selection on the queue-level topology allows a flow to be assigned different priorities at each node, thereby increasing flexibility compared to legacy approaches (e.g., [05]; [Guo+10]; [Bal+11]; [AX14]; [Sha+14]; [TPR14]; [Gro+15]; [Hu+16]) which usually assign fixed priorities to flows along their complete path. We will detail and investigate this benefit further in chapter 4.

2.3.4 Architecture Components: Routing, Resource Allocation and Reservation, Access Control and Cost Function

The embedding of a new flow must not violate the delay guarantees provided to previously embedded flows. Indeed, as shown by Eqn. 2.19, embedding a new flow at a link updates the service offered to other flows traversing that link, which in turn updates the delay bounds for these flows and might hence potentially cause the violation of the E2E delay guarantees provided to these flows.

As a result, resources usage has to be taken into account while routing. The approach here is to rely on a *resource allocation* algorithm that defines the amount of resources to allocate to the different queues at each link of the network. The *routing* algorithm is then responsible for finding a path in the queue-level topology for which the delay of the new flow is guaranteed and that only uses resources that are still available, thereby ensuring that the guarantees of previously embedded flows are not violated. The availability of resources is provided by an *access control* logic. This logic relies on a *resource reservation* component that keeps track of the resources already used with respect to the total resources allocated by the resource allocation algorithm. Access is rejected if the new flow requires more resources than what remains from the allocated resources. The three constraints of the problem formulated in Sec. 2.3.2 are then satisfied as follows.

(i) The routing algorithm gets the worst-case E2E delay of a route by summing up the per-queue worst-case delay values (obtained from the resource allocation component) of the queues on the given route.

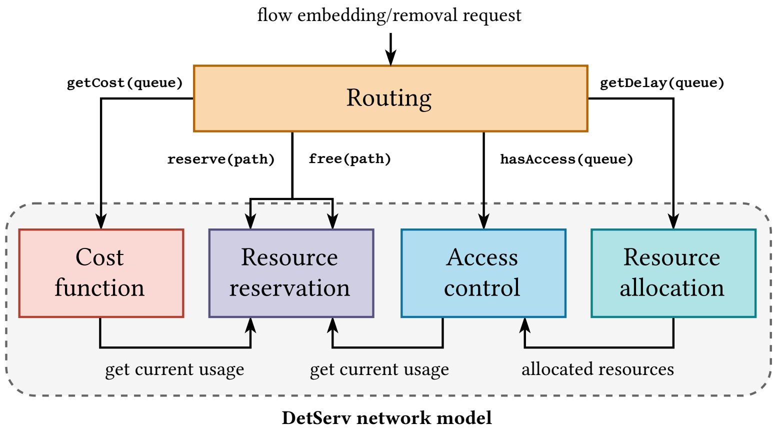


Figure 2.26: The DetServ CP architecture composed of a *routing* procedure and a *network model* that consists of a *cost function*, an *access control* logic, a *resource reservation* component and a *resource allocation* algorithm. The interface of a generic DetServ network model is shown in underlined fixed-width font. A flow request is handled by the routing procedure. Its task is to find a suitable route in the queue-level topology for the corresponding flow. While routing, the getDelay and hasAccess methods of the network model are used for the computation of worst-case delays and for access control. The reserve and free functions are for their part used to update the state of the network model to reflect the embedding or removal of a flow.

(ii) The access control logic ensures that, if the routing algorithm uses only queues for which access has been granted, the worst-case delay values defined by the resource allocation algorithm are not exceeded, and hence that the worst-case E2E delay computed for previous flows is still valid.

(iii) We introduce a *cost function*, the minimization of which maximizes the probability of future request acceptance. This function is used by the routing algorithm to choose among all the paths satisfying the E2E delay constraint of the new flow. The cost function transforms the network state, characterized for instance by data rates, buffer consumption, and already embedded flows, into a cost metric for each edge. The cost metric should maximize the number of flows that the network can serve.

The interaction between these five components is illustrated in Fig. 2.26. We refer to the combination of the *cost function*, *access control*, *resource reservation* and *resource allocation* components as the *network model*. This constitutes the DetServ architecture for providing predictable latency in programmable networks.

2.3.5 Model Functions: Interface of the Network Model

The components described in Sec. 2.3.4 and Fig. 2.26 highlight the interface that the network model has to expose to the routing algorithm. This interface consists of the following five so-called *model functions*.

- getDelay. Computes the worst-case delay of a given queue. This worst-case delay must be valid for the whole lifetime of the network.

- hasAccess. Checks whether or not there are still enough resources available for a given flow at a given queue. This method ensures that the delays returned by getDelay are never violated.
- reserve. Updates the model state to reflect the embedding of a new flow.
- free. Updates the model state to reflect the removal of a previously embedded flow.
- getCost. Computes a cost value for a given queue. Queues that should preferentially be used to maximize the probability of acceptance of future flow requests should have lower cost values.

The processing of a flow request is then illustrated in Fig. 2.26. Upon receipt of a flow embedding request, the routing algorithm searches for a solution to the problem defined in Sec. 2.3.2. While searching, the algorithm uses the getDelay and hasAccess methods to obtain the delay of an edge and to check if enough resources are available at an edge. Once a path has been found, the reserve method is used to update the state of the model in order to reflect the embedding of the new flow. Similarly, the free method is used upon receipt of a flow removal request in order to reflect the removal of the corresponding flow.

How these methods are implemented depends on how and which resources are allocated and managed at each queue. In Sec. 2.4, we present in detail a model implementing this interface by allocating a *maximum delay* resource at each queue. Another model, presented in [GVK17]; [Guc18] but not shown here, allocates *maximum burst* and *maximum rate* resources at each queue. [GVK17]; [Guc18] show that the latter provides less potential for reaching high network utilization. In Sec. 2.6, we shortly discuss how the presented DetServ architecture can be used to provide predictable latency in networks with wireless hops. We discuss the routing procedure in detail in chapter 3.

2.4 DetServ: End-to-End Network Latency Model

As elaborated in Sec. 2.3, a system for predictable latency requires a network model for the computation of worst-case delays and for access control. This network model must implement the interface defined in Sec. 2.3.5. We presented two such models based on DNC theory in [GVK17], both of which were published in a previous doctoral thesis from the Technical University of Munich [Guc18]. The first model, the multi-hop model (MHM), assigns a rate and a buffer budget to each queue in the network. This allows to compute worst-case delays for any path in the network. We showed that the MHM requires an *a priori* choice regarding the characteristics of flows that are to be embedded based on the trade-off between rate, buffer capacity, and delay. The second model, the threshold-based model (TBM), simplifies this trade-off by only fixing a maximum delay for each queue in the network, thereby avoiding the *a priori* assignment of rate and buffer budgets. In this section, we present in detail the TBM, as it showed more potential in terms of number of accepted flows and as it is the main part of our contribution in [GVK17]. The performance of the TBM, in terms of induced runtime and number of accepted flows is investigated in Sec. 2.5. We further confirm through packet-level simulations that the model is indeed correct: the delay guarantees it provides are never violated. Comparable evaluations for the MHM are available in [GVK17] and the doctoral thesis [Guc18]. Additionally to the elaborations published in this previous thesis [Guc18], we provide a detailed example of the operations of the model.

The power of the proposed model resides in the fact that it can be used with off-the-shelf switches supporting PQ and any SDN protocol providing standard enqueuing and forwarding programmable primitives, e.g., OF 1.0 [Ope09].

2.4.1 Notations

$\mathcal{P}$ and $\mathcal{G}$ respectively denote the physical and queue-level graphs (see Sec. 2.3.3). We use the indices E and N to refer to the set of edges and nodes of these graphs. For example, $\mathcal{P}_E$ represents the set of edges in the physical graph. The capacity of a physical link $(u, v) \in \mathcal{P}_E$ is denoted by $R_{(u,v)}$. We assume a non-preemptive strict PQ scheduler with $Q_{(u,v)}$ queues at the physical link $(u, v) \in \mathcal{P}_E$. Edges in the queue-level network are denoted by (u, v, p), where (u, v) is the corresponding physical link and $p \in \{1, \ldots, Q_{(u,v)}\}$ is the priority of the corresponding queue at the physical link, 1 being the highest priority.

$\mathcal{F}$ denotes the set of flows already embedded in the network. For a given embedded flow $f \in \mathcal{F}$ or for a given flow f to be embedded,

- r_f denotes the rate (as defined in Sec. 2.2.4.2) of the flow,
- $b_f^{(u,v,p)}$ denotes the burst size (as defined in Sec. 2.2.4.2) of the flow at queue $(u, v, p) \in \mathcal{G}_E$ (as we have seen in Sec. 2.2.6 that the burst of a flow changes at each hop),
- t_f denotes the E2E latency requirement of the flow,
- l_f^{max} denotes the maximum packet size of the flow, and
- $P_f \subseteq \mathcal{G}_E$ denotes the set of queues through which the flow is routed (empty set if the flow is not embedded yet).

The maximum packet size in the network is denoted by L^{max}. If it is not known, the maximum Ethernet frame size of 1542 bytes (including the preamble, a VLAN tag and the inter-frame gap).

For a given queue $(u, v, p) \in \mathcal{G}_E$,

- $\mathcal{F}_{(u,v,p)} \subseteq \mathcal{F}$ denotes the set of embedded flows traversing the queue,
- $\mathbf{U}_r^{(u,v,p)}$ denotes the sum of the rates of the flows traversing the queue, i.e.,

$$\mathbf{U}_r^{(u,v,p)} \triangleq \sum_{f \in \mathcal{F}_{(u,v,p)}} r_f, \tag{2.25}$$

- $\mathbf{U}_b^{(u,v,p)}$ denotes the sum of the bursts of the flows traversing the queue, i.e.,

$$\mathbf{U}_b^{(u,v,p)} \triangleq \sum_{f \in \mathcal{F}_{(u,v,p)}} b_f^{(u,v,p)}, \tag{2.26}$$

- $l^{max}_{(u,v,p)}$ denotes the maximum packet size among all the flows traversing the queue, i.e.,

$$l^{max}_{(u,v,p)} \triangleq \max_{f \in \mathcal{F}_{(u,v,p)}} \{l^{max}_f\}, \tag{2.27}$$

- $\mathbf{M}_t^{(u,v,p)}$ denotes the worst-case maximum latency of the queue,
- $\mathbf{M}_b^{(u,v,p)}$ denotes the worst-case maximum backlog at the queue, and
- $B_{(u,v,p)}$ denotes the buffer capacity of the queue.

Using these notations and applying the changes described in Sec. 2.2.7 for packet-based systems, Eqn. 2.20, 2.21, 2.22 and 2.23 can be respectively rewritten as

$$R_{(u,v,p)} = R_{(u,v)} - \sum_{j=1}^{p-1} \mathbf{U}_r^{(u,v,j)}, \tag{2.28}$$

$$T_{(u,v,p)} = \frac{\sum_{j=1}^{p-1} \mathbf{U}_b^{(u,v,j)} + \max_{p+1 \le j \le Q_{(u,v)}} \{l^{max}_{(u,v,j)}\} + l^{max}_{(u,v,p)}}{R_{(u,v)} - \sum_{j=1}^{p-1} \mathbf{U}_r^{(u,v,j)}}, \tag{2.29}$$

$$\mathbf{M}_t^{(u,v,p)} = \frac{\sum_{j=1}^{p} \mathbf{U}_b^{(u,v,j)} + \max_{p+1 \le j \le Q_{(u,v)}} \{l^{max}_{(u,v,j)}\} + l^{max}_{(u,v,p)}}{R_{(u,v)} - \sum_{j=1}^{p-1} \mathbf{U}_r^{(u,v,j)}}, \tag{2.30}$$

and

$$\mathbf{M}_b^{(u,v,p)} = \mathbf{U}_b^{(u,v,p)} + \mathbf{U}_r^{(u,v,p)} T_{(u,v,p)}, \tag{2.31}$$

where $\beta_{R_{(u,v,p)},T_{(u,v,p)}}$ is the rate-latency service curve (see Sec. 2.2.5.2) offered by a queue $(u,v,p) \in \mathcal{G}_E$.

2.4.2 Mathematical Formulation of Latency Requirements

First, in order to respect the latency requirements of embedded flows, we must have,

$$\sum_{(u,v,p) \in P_f} \mathbf{M}_t^{(u,v,p)} \le t_f \qquad \forall f \in \mathcal{F}. \tag{2.32}$$

Second, in order to avoid any buffer overflow (and hence any packet loss), we must have

$$\mathbf{M}_b^{(u,v,p)} \le B_{(u,v,p)} \qquad \forall\ (u,v,p) \in \mathcal{G}_E. \tag{2.33}$$

2.4.3 Model Functions Implementation: The Threshold-based Model

Both bounds in Eqn. 2.30 and 2.31 depend on $\mathbf{U}_b^{(u,v,j)}$, $\mathbf{U}_r^{(u,v,j)}$ and $l^{max}_{(u,v,j)}$ for some j, i.e., on the burst size, rate and maximum packet size of other flows embedded on the same physical link. This means that, if a new flow is embedded on a link $(u,v) \in \mathcal{P}_E$, the worst-case delay (Eqn. 2.30) and buffer consumption (Eqn. 2.31) of some of the queues at the link will be updated, thereby possibly violating requirements of some previously embedded flows (Eqn. 2.32 and 2.33). As explained in Sec. 2.3.4, we do not want to check that the latency requirements of the already embedded flows are still satisfied (i.e., check Eqn. 2.32) after a new flow embedding. That means that the worst-case bounds $\mathbf{M}_t^{(u,v,p)}$

and $\mathbf{M}_b^{(u,v,p)}$ have to be bounded independently of the state of the network. In such a way, if Eqn. 2.32 for a given flow f was satisfied when the flow was embedded, it will be kept satisfied for the whole lifetime of the network.

The two models we presented in [GVK17] differ in the way they fix the $\mathbf{M}_t^{(u,v,p)}$ bounds. While the MHM upper-bounds the variable parts of Eqn. 2.30 (i.e., $\mathbf{U}_r^{(u,v,p)}$ and $\mathbf{U}_b^{(u,v,p)}$), the TBM fixes $\mathbf{M}_t^{(u,v,p)}$ itself and lets the variables vary until the fixed threshold is reached. The MHM requires an *a priori* decision on how to allocate the rate and buffer resources at each queue. That is problematic, as the type of future flow requests, would it be bursty or bandwidth-hungry, is unknown. The TBM solves this drawback by automatically chosing between buffer capacity and data rate as flows are added to the network, thereby allocating the rate and buffer capacity resources only when needed rather than pre-allocating them without knowing future flow requests.

After fixing the worst-case delay of each queue (Eqn. 2.30) by defining the thresholds $\mathbf{T}_t^{(u,v,p)}$, flows are accepted in a queue as long as the worst-case delay of the queues at the same link do not exceed their respective thresholds.

This approach has two main benefits. First, as mentioned, the data rate and buffer space resources are allocated only when needed, rather than *a priori*, thereby leading to a better utilization of the resources. Second, the resource allocation algorithm is now simplified since it only has to optimize with respect to one variable (the time) rather than two (buffer space and data rate). In other words, the TBM replaces the three data rate, buffer space and delay resources by a *single* one: delay.

Unfortunately, when adding a flow in a queue (u, v, p), besides checking that $\mathbf{M}_t^{(u,v,p)} \leq \mathbf{T}_t^{(u,v,p)}$ for this queue, the access control mechanism has to check that the thresholds of lower priority queues are also not exceeded, what is not necessary for the MHM. That is, the access control mechanism has to check that

$$\mathbf{M}_t^{(u,v,j)} \leq \mathbf{T}_t^{(u,v,j)} \quad \forall\, j : p \leq j \leq Q_{(u,v)}. \tag{2.34}$$

Besides, the access control scheme has to make sure that no buffer overflow can be caused by the embedding of the new flow, i.e.,

$$\mathbf{M}_b^{(u,v,j)} \leq B_{(u,v,j)} \quad \forall\, j : p \leq j \leq Q_{(u,v)}. \tag{2.35}$$

Note that Eqn. 2.30 and 2.31 require the knowledge of the maximum packet size in lower priority queues. This means that, strictly speaking, when embedding a flow in a queue, higher priority queues also have to be checked since the maximum packet size might have changed. To avoid this, we replace this term by L^{max}. From this, we have

$$\mathbf{M}_t^{(u,v,p)} \leq \frac{\sum_{j=1}^{p} \mathbf{U}_r^{(u,v,j)} + L^{max} + l_{(u,v,p)}^{max}}{R_{(u,v)} - \sum_{j=1}^{p-1} \mathbf{U}_r^{(u,v,j}}, \tag{2.36}$$

and

$$\begin{aligned} \mathbf{M}_b^{(u,v,p)} \leq \mathbf{U}_b^{(u,v,p)} &+ \\ \mathbf{U}_r^{(u,v,p)} &\frac{\sum_{j=1}^{p-1} \mathbf{U}_b^{(u,v,j)} + L^{max} + l_{(u,v,p)}^{max}}{R_{(u,v)} - \sum_{j=1}^{p-1} \mathbf{U}_r^{(u,v,j)}}, \end{aligned} \tag{2.37}$$

```
function GETDELAY($(u, v, p)$)
    return $\mathbf{T}_t^{(u,v,p)}$

function HASACCESS($f$, $(u, v, p)$)
    for $j \in \{p, \dots, Q_{(u,v)}\}$ do
        $\mathbf{M}_t^{(u,v,j)} \leftarrow$ Eqn. 2.36 including new flow
        $\mathbf{M}_b^{(u,v,j)} \leftarrow$ Eqn. 2.37 including new flow
        if $\mathbf{M}_t^{(u,v,j)} > \mathbf{T}_t^{(u,v,j)}$ or $\mathbf{M}_b^{(u,v,j)} > B_{(u,v,j)}$ then
            return false
    return true

function RESERVE($f$, $P$)
    for $(u, v, p) \in P$ do
        $\mathbf{U}_b^{(u,v,p)} \leftarrow \mathbf{U}_b^{(u,v,p)} + b_f^{(u,v,p)}$
        $\mathbf{U}_r^{(u,v,p)} \leftarrow \mathbf{U}_r^{(u,v,p)} + r_f$
        Update $l_{(u,v,p)}^{max}$

function FREE($f$, $P$)
    for $(u, v, p) \in P$ do
        $\mathbf{U}_b^{(u,v,p)} \leftarrow \mathbf{U}_b^{(u,v,p)} - b_f^{(u,v,p)}$
        $\mathbf{U}_r^{(u,v,p)} \leftarrow \mathbf{U}_r^{(u,v,p)} - r_f$
        Update $l_{(u,v,p)}^{max}$
```

Figure 2.27: The model functions for the TBM (excluding GETCOST, the implementation of which is free). The threshold for the latency of a queue is chosen by the resource allocation algorithm. Access to a queue $(u, v, p) \in \mathcal{G}_E$ is then controlled by checking that the new worst-case bound does not exceed its threshold value. Besides, as the state of a queue influences the state of lower priority queues, the access control mechanism also has to check that the worst-case bounds of lower priority queues do not exceed their respective thresholds. Finally, the buffer capacity also has to be checked for the different queues.

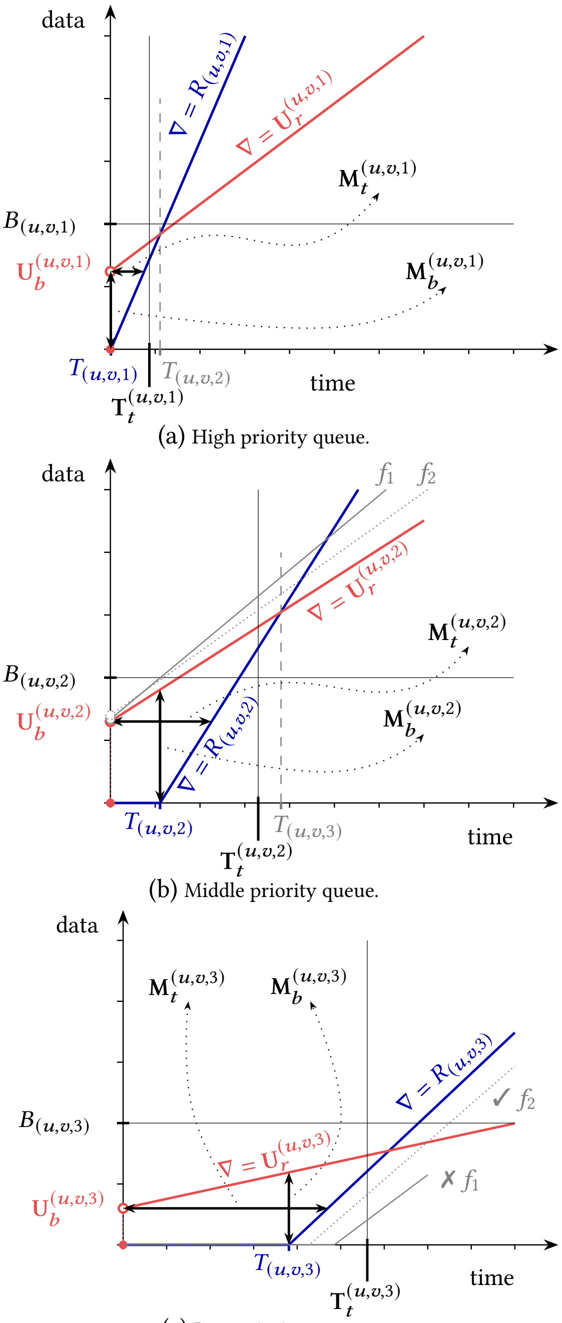
which only depend on the state of higher priority queues. As a result, it is sufficient to only check lower priority queues when embedding a new flow.

The pseudo-code of the TBM model functions implementation is given in Fig. 2.27. The reservation and freeing methods simply consist in updating the state variables. The delay of a queue link edge is simply the one fixed by the resource allocation algorithm and the access control scheme verifies that Eqn. 2.34 and 2.35 are still verified for the subject queue and the lower priority queues if the flow is embedded.

2.4.4 Threshold-based Model: Example

In this section, we present a detailed example of the operation of the TBM at a given physical link. We consider a given physical link $(u, v) \in \mathcal{P}_E$ of capacity $R_{u,v} = 1$ Gbps and with three priority queues scheduled by a non-preemptive strict PQ scheduler. The buffer capacity at each queue is 300 KB, i.e., $B_{(u,v,p)} = 300$ KB $\forall p \in \{1, 2, 3\}$. $L^{max} = 1530$ bytes. The example is illustrated in Fig. 2.28.

Let us assume that the resource allocation algorithm assigned $\mathbf{T}_t^{(u,v,1)} = 1.74$ ms, $\mathbf{T}_t^{(u,v,2)} = 6.6$ ms and $\mathbf{T}_t^{(u,v,3)} = 11.22$ ms as limit worst-case delays for the three queues. Let us further consider that the queues are in the following state. The high priority queue is traversed by an aggregate flow with a



(a) High priority queue.

(b) Middle priority queue.

(c) Low priority queue.

Figure 2.28: Example of service and arrival curves for a non-preemptive strict PQ scheduler with three queues using the TBM. All the queues are constrained by two parameters. The first one, $B_{(u,v,p)}$, corresponds to the buffer space at the queue. The second one, $\mathbf{M}_t^{(u,v,p)}$, assigned by the resource allocation algorithm, corresponds to the maximum worst-case delay of the queue. The access control has to reject any traffic that makes the worst-case backlog or delay at this queue grow bigger than these bounds. If bounds are not exceeded, bounds of lower priorities queues also have to be checked. A flow can then be embedded only if bounds of all lower priority queues are also not exceeded. Fig. 2.28b and 2.28c show the updated arrival and services curves if a flow f_1 is added to the middle priority queue. From the point of view of the middle priority queue, none of its two bounds would be violated and the flow can be added. Nevertheless, f_1 cannot be accepted because the updated service curve of the lower priority queue would lead to the violation of its worst-case delay threshold. In dashed lines, Fig. 2.28b and 2.28c also show the updated arrival and services curves if another flow f_2 is added to the middle priority queue. In this case, none of the bounds in both queues will be violated and the flow can hence be accepted.

burst of $\mathbf{U}_b^{(u,v,1)} = 186$ KB, a rate of $\mathbf{U}_r^{(u,v,1)} = 322$ Mbps and a maximum packet size $l_{(u,v,1)}^{max} = 700$ bytes. The middle and low priority queues are traversed by aggregate flows with bursts of $\mathbf{U}_b^{(u,v,2)} = 195$ KB and $\mathbf{U}_b^{(u,v,3)} = 90$ KB, rates of $\mathbf{U}_r^{(u,v,2)} = 275$ Mbps and $\mathbf{U}_r^{(u,v,3)} = 93$ Mbps, and maximum packet sizes of $l_{(u,v,2)}^{max} = 400$ bytes and $l_{(u,v,3)}^{max} = 1200$ bytes, respectively. The corresponding service and arrival curves for the three different priority queues are shown in Fig. 2.28. Note that, graphically, $T_{(u,v,j)}$ corresponds to the abscissa at which the arrival and service curves at queue $(u, v, j-1)$ intersect[18]. This can be intuitively understood. Indeed, a queue has to wait for the directly higher priority queue to empty its backlog before being served. The service curve parameters for all the priority queues are directly given by Eqn. 2.29 and 2.28. The current buffer and delay usage $\mathbf{M}_b^{(u,v,p)}$ and $\mathbf{M}_t^{(u,v,p)}$ are shown along with their limits $B_{(u,v,p)}$ and $\mathbf{T}_t^{(u,v,p)}$.

Let us consider that the routing algorithm then requests access to the middle priority queue for a flow f_1 with burst $b_f^{(u,v,2)} = 5500$ bytes and rate $r_f = 82$ Mbps. We assume the maximum packet size of the flow is smaller than the current maximum packet size of the aggregate, thereby leaving $l_{(u,v,2)}^{max}$ unchanged. The high priority queue is not concerned by this request. The updated arrival curve for the middle priority queue is shown in Fig. 2.28b (thin full line). The delay and backlog thresholds of this queue are not exceeded. From the point of view of the middle priority queue, the flow can hence be embedded. The low priority queue state also has to be checked. The updated service curve offered by the low priority queue is shown in Fig. 2.28c (thin full line). Unfortunately, we can see that the worst-case delay limit $\mathbf{T}_t^{(u,v,3)}$ would now be exceeded. As a result, f_1 has to be rejected from the middle priority queue because it would violate the delay threshold of the low priority queue.

The routing algorithm then requests access to the middle priority queue for a flow f_2 with burst $b_f^{(u,v,2)} = 15$ KB and rate $r_f = 30$ Mbps. We once more assume that the maximum packet size of the flow is smaller than the current maximum packet size of the aggregate. The high priority queue is still not concerned by the request. The updated arrival curve for the middle priority queue is shown in Fig. 2.28b (thin dotted line). As for f_1, we can see that the delay and backlog thresholds of this queue are not exceeded. Before allowing the embedding of the flow to this queue, the updated state of the low priority queue also has to be checked. The updated service curve offered by the low priority queue is shown in Fig. 2.28c (thin dotted line). We can see that the the worst-case delay limit $\mathbf{T}_t^{(u,v,3)}$ would, in this case, not be exceeded. As a result, since all the worst-case limits are still respected, f_2 can be embedded in the middle priority queue and the arrival and service curves can be updated to the thin dotted lines.

2.4.5 Threshold-based Model: The Blocking Problem

The TBM presents an inherent blocking problem where the state in a queue can prevent flows to be added in other queues. Let us consider a low priority queue that reached its delay threshold. No flows can be added anymore to this queue, as that would violate its delay threshold. The problem is that this will also block further embeddings in higher priority queues, even if these are still far from their own delay threshold. Indeed, adding a flow to a high priority queue will lower the service curve of

[18] Strictly speaking, this is only the case if we neglect the $l_{(u,v,p)}^{max}$ term in Eqn. 2.29, i.e., if the store-and-forward behavior of switches is neglected. Nevertheless, this term only *slightly* shifts the $T_{(u,v,p)}$ values, so we can consider, for understanding purposes, that the statement is true. Note that, in the figures, the real values are shown. It can be seen that $T_{(u,v,p)}$ is indeed always *very* close to the intersection of the two curves.

the low priority queue, thereby increasing its worst-case latency and exceeding its delay threshold. As a result, one might think that it is wiser to fill high priority queues first in order to avoid this problem. However, let us imagine we fill high priority queues and reach their delay threshold. Such high usage in the high priority queues could lead to very low service offered to lower priority queues. As a result of such low service curves, the lower priority queues would have a high worst-case latency even for low traffic, thereby potentially exceeding their delay threshold even for a single flow and hence preventing the usage of these low priority queues.

We observe that the usage of high priority queues blocks the usage of low priority queues, and, more surprisingly, also vice-versa. Consequently, when finding a route in the queue-level graph, the routing algorithm must operate cautiously in order to avoid such blocking situations which would inevitably cause resource waste. This problem actually corresponds to the problem of designing a good cost function (i.e., implement the getCost method) that would avoid this problem and lead to high network utilization and few rejections. We will see in chapter 4 that this problem of finding a good routing objective is very hard and we will see how we can tackle this problem to reach high network utilization.

2.4.6 Computation of the Burst Increase

We mentioned that the burst of a flow changes at each hop. However, we did not explain how this change can be computed on a *per-flow* basis and how this impacts delay computations. From Sec. 2.2.6.4, we know that an aggregate flow with arrival curve $\gamma_{\mathbf{U}_r^{(u,v,p)},\mathbf{U}_b^{(u,v,p)}}$ traversing a queue offering a service curve $\beta_{R_{(u,v,p)},T_{(u,v,p)}}$ will see its burst $\mathbf{U}_b^{(u,v,p)}$ increased by $\mathbf{U}_r^{(u,v,p)}T_{(u,v,p)}$, i.e.,

$$\mathbf{U}_b^{*(u,v,p)} = \mathbf{U}_b^{(u,v,p)} + \mathbf{U}_r^{(u,v,p)}T_{(u,v,p)}. \tag{2.38}$$

$\mathbf{U}_b^{*(u,v,p)}$ is the new burst of the entire aggregate after having traverse queue (u, v, p). The flows composing this aggregate might take different routes at the next hop and the individual burst increases of the individual flows composing the aggregate must be computed. From Eqn. 2.25 and 2.26, Eqn. 2.38 can be rewritten as

$$\mathbf{U}_b^{*(u,v,p)} = \sum_{f\in\mathcal{F}_{(u,v,p)}} \left(b_f^{(u,v,p)} + r_f T_{(u,v,p)}\right), \tag{2.39}$$

which highlights the contribution of each individual flow to the burst increase. As a result, the burst of a flow $f \in \mathcal{F}_{(u,v,p)}$ when entering a queue $(v, t, q) \in \mathcal{G}_E$ after having traversed queue $(u, v, p) \in \mathcal{G}_E$ is given by

$$b_f^{(v,t,q)} = b_f^{(u,v,p)} + r_f T_{(u,v,p)}, \tag{2.40}$$

which depends, through $T_{(u,v,p)}$, on other flows traversing the same physical link, and hence, through $b_f^{(u,v,p)}$ on other flows traversing other physical links. This dependency of the burst increase on other embedded flows is problematic. Indeed, this means that, when a flow is embedded in a queue, the burst increase of other flows traversing the same link might change, possibly violating already performed access control checks. As explained in Sec. 2.3.4, such a situation must be avoided and the burst increase of a flow must therefore be, as the worst-case delay of a queue, independent of the

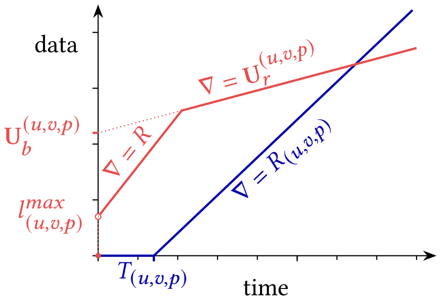


Figure 2.29: Shaped arrival curve of an aggregate flow traversing a queue $(u, v, p) \in \mathcal{G}_E$ coming from an input link with rate R. The knowledge of the physical properties of the input link of the flow allows to limit the burst and rate of the aggregate respectively to the maximum packet size $l^{max}_{(u,v,p)}$ of the flow and to the maximum rate R of the link. Graphically, we can see that such a shaping reduces the values of the backlog and delay bounds.

network state. From Eqn. 2.29, 2.30, and 2.34, it is straightforward that

$$T_{(u,v,p)} \leq \mathbf{M}_t^{(u,v,p)} \leq \mathbf{T}_t^{(u,v,p)} \quad \forall\ (u, v, p) \in \mathcal{G}_E. \tag{2.41}$$

Therefore, the burst increase of a flow f is such that

$$b_f^{(v,t,q)} \leq b_f^{(u,v,p)} + r_f \mathbf{T}_t^{(u,v,p)}. \tag{2.42}$$

The TBM can then compute $b_f^{(v,t,q)}$ using this bound which is independent of the network state.

We note that, if the cycle time (or inter-arrival time of packets) of a flow is greater than its latency requirement, then the burst increase can be neglected. Indeed, in such a case, a packet is ensured to reach its destination before the following packet is sent. As a result, packets of the same flow will not queue up at any queue and the burst of the flow will never increase.

2.4.7 Input Link Shaping

So far, we considered that the arrival curve of the aggregate flow entering a queue $(u, v, p) \in \mathcal{G}_E$ is $\gamma_{\mathbf{U}_r^{(u,v,p)},\mathbf{U}_b^{(u,v,p)}}$, that is, that the burst of the aggregate flow entering a queue is given by the sum of all the bursts of all the flows composing the aggregate (see Eqn. 2.26). Nevertheless, the individual flows come from physical links of finite capacity. Hence, the traffic entering a given queue is further limited by the capacity of these links it is coming from. Considering this new bound on the traffic entering a queue, we can lower the corresponding arrival curves, yielding lower bound values and thereby potentially accepting more flows in the network.

2.4.7.1 Concept

The idea, to which we refer to as input link shaping (ILS), is illustrated in Fig. 2.29 for a given queue (u, v, p) traversed by a set of flows coming from a common input link of capacity R. From the knowledge of the physical properties of the input link, besides its traditional arrival curve, the aggregate flow is additionally constrained by a token bucket arrival curve with rate R and burst $l^{max}_{(u,v,p)}$. If a flow is constrained by two different token bucket arrival curves, a better arrival curve for the flow is the minimum of these curves (see Sec. 2.2.4.1 and 2.2.6.4). The resulting arrival curve of

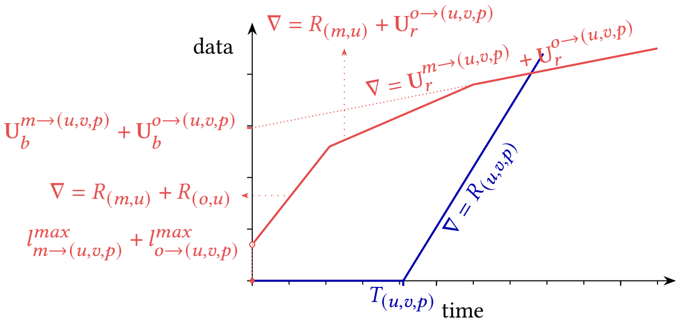


Figure 2.30: Example of shaped arrival curve. The aggregate flow traversing queue (u, v, p) comes from two input links (m, u) and (o, u). Each input link has shaped the traffic it carries as shown in Fig. 2.29 and the resulting aggregate, corresponding to the sum of the two shaped arrival curves, is composed of three segments with decreasing slopes. The backlog and delay bounds can then be reached at any angular point of both curves. The bounds will always be lower than if shaping was not taken into account.

the aggregate flow is of the form shown in Fig. 2.29. We can see that the backlog and delay bounds will always be smaller than if shaping was not taken into account.

In Sec. 2.2.6, we have presented DNC results for computing the output arrival curve of a flow after it has traversed a network node characterized by a given service curve. Our proposal to cut off a part of this arrival curve, though intuitive, might seem to contradict these DNC results. The justification is the following. The DNC results presented are solely based on the basic arrival and service curve concepts. While the service curve gives a lower bound on the service a network node will offer to a flow, it does not specify anything regarding the maximum service the node could offer, hence potentially allowing infinite service, i.e., infinite rate. Taking this into account, an infinite service could instantly output the current backlog as a single burst. As a matter of fact, we know more than what the basic service curve concept provides to DNC theory: the service provided by the network node can never be higher than the link rate. The shaping we introduce is hence augmenting the presented DNC results, rather than contradicting them. A similar idea is formalized by the concept of *maximum service curve* in DNC.

2.4.7.2 Adapting the Threshold-Based Model

In order to introduce input link shaping (ILS) for the TBM, we must now keep track of the incoming link from which each flow arrived to a queue. To do so, we add the incoming link as parameter to the state variables of the TBM. The variables $\mathbf{U}_r^{m\to(u,v,p)}$, $\mathbf{U}_b^{m\to(u,v,p)}$ and $l^{max}_{m\to(u,v,p)}$ now respectively keep track of the rate, burst and maximum packet size of the aggregate flow coming from the physical edge (m, u) and traversing the queue-link edge (u, v, p). The arrival curve considered at a queue (u, v, p) can then be computed as

$$\sum_{x:(x,u)\in\mathcal{P}_E} \left(\min\left\{\gamma_{R_{(x,u)},l^{max}_{x\to(u,v,p)}}, \gamma_{\mathbf{U}_r^{x\to(u,v,p)},\mathbf{U}_b^{x\to(u,v,p)}}\right\}\right), \tag{2.43}$$

i.e., as the sum of the shaped arrival curves.

An example for two input links is shown in Fig. 2.30. One can see that the summed up arrival curve can have up to n knee points, where n is the number of physical input links.

Obviously, the GETDELAY method in Fig. 2.27 does not change. The RESERVE and FREE methods have to be updated to keep track of the new state variables. For its part, the HASACCESS method has to be changed at lines 6–7. Since the arrival curves are not token buckets anymore, the formulas for computing the worst-case delay $\mathbf{M}_t^{(u,v,p)}$ and backlog $\mathbf{M}_b^{(u,v,p)}$ are not valid anymore. These values have now to be computed geometrically. From the convexity of the region between the curves, the delay (resp. backlog) bound can be computed by comparing the horizontal (resp. vertical) deviations for all the knee points of the two curves.

The burst increase can still be computed per-flow using Eqn. 2.42.

2.5 DetServ: Evaluation

In this section, we describe the evaluation of the TBM model and its ILS extension. The goal of this evaluation is to answer three fundamental questions.

1. Are latency guarantees provided by the TBM and its ILS extension indeed correct?
2. Is the request processing time induced by the TBM reasonable for online flow embedding?
3. What is the impact of ILS on the runtime and performance of the TBM in terms of number of accepted flows?

With this aim, the evaluation is separated in two parts. First, in Sec. 2.5.1, in order to answer the first question, we run a packet-level simulation of one physical link managed by the TBM with and without ILS and observe the delay experienced by the individual packets. We show that the TBM and ILS indeed respect the delay guarantees provided to the different flows. Although the simulation is performed only at a single link, this also confirms that the models are valid E2E. Indeed, if the worst-case delay of each queue is guaranteed, the E2E delay of each flow, corresponding to the sum of the individual worst-case delays of each queue visited by the flow, is also guaranteed. Second, in Sec. 2.5.2, in order to answer the second and third questions, we run a network-wide simulation by generating series of flow requests for different network settings and observe the request processing time of the TBM with and without ILS along with the amount of flows they can accept. We show that the runtime of the TBM is indeed reasonable. Indeed, in our simulations, the total request processing time of the TBM with ILS remains lower than 350 ms in 99% of the cases and never exceeds 620 ms. We further show that ILS allows the TBM to accommodate more traffic in the network.

The intent here is not to compare the performance of the DetServ architecture and its models with the SoA but simply to validate the correctness and usability of DetServ. A thorough comparison with the SoA, in terms of number of flows that can be accommodated and reachable network utilization, is done in chapter 4.

2.5.1 Packet-level Simulation: Confirming Correctness

We simulate the access control of a single 1 Gbps link with four priority queues and varying amount of input links (1, 2, 3, 5 and 10). For each model and amount of input links, we generate flow registration and termination requests during 100 seconds. A flow is added to a queue (RESERVE)

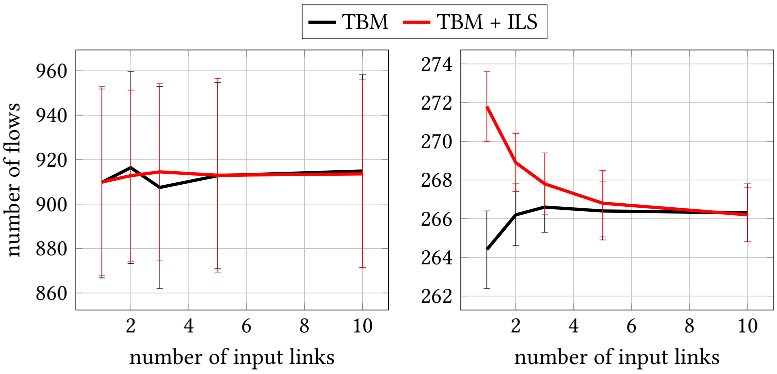


Figure 2.31: On the left diagram, results of the packet-level simulation when flows are evenly distributed among the combinations of input link and queue. ILS has no influence on the performance of the model. On the right diagram, one priority queue receives more traffic from a given input link and the traffic is more bursty. ILS now increases the performance of the TBM when the amount of input link is low. No packet loss nor deadline violation was observed in both scenarios.

only if the access control (hasAccess) accepts it. We generate requests at a rate that is high enough for saturating the link (250 requests per second) and hence for experiencing rejections of requests.

We consider a resource allocation algorithm that assigns the delays 0.487 ms (high priority), 1.437 ms, 3.035 ms, and 4.709 ms (low priority) to the different queues. These delay values are chosen so that they lead to a nice distribution of QoS levels among the queues. As we focus the simulation on one physical link, the *routing* and *cost function* components (see Sec. 2.3.4) are not involved. Those will be briefly covered in Sec. 2.5.2 and chapter 3 will deeply cover the routing component.

2.5.1.1 First Configuration: No Delay Violations

In a first configuration, each request is defined by a data rate (between 400 Kbps and 1200 Kbps), a burst size (between 70 bytes and 150 bytes), a maximum packet size (between 64 bytes and the burst of the flow) and a delay constraint (between 10 ms and 100 ms) which are uniformly randomly distributed in their respective ranges. These are values in line with traffic traces observed in an operational industrial wind park network in the context of the VirtuWind H2020 European Project [Mah+16]. We consider L^{max} as the maximum Ethernet frame size including preamble, VLAN tag and inter-frame gap, i.e., $L^{max} = 1542$ bytes. Because the delay constraint is greater than the delay of all the queues, the delay will not influence the rejection or acceptance of requests. This is intentional. The reason for this is that we are willing to fully saturate the considered link. As a result, having requests rejected because of their delay constraint will not affect the amount of flows that are accepted. The generated flow requests are evenly distributed among the different combinations of input link and queue of the considered link. Flow requests are characterized by a duration which is randomly generated from an exponential distribution with an average duration of 100 seconds, representing the long-duration characteristic of industrial flows.

For each run, the amount of flows embedded at the link was sampled every second. The left diagram of Fig. 2.31 shows, for each amount of input link, the average and the standard deviation of these sampled values. We observe that ILS does not provide any benefit. This could have been expected. Indeed, as shown in Fig. 2.30, the maximum burst computation will be reduced only if one knee point of the arrival curve is after the knee point of the service curve. In our particular setup of requests distributed evenly among the combinations of input link and queue, the knee points of the arrival curves are always before the knee point of the service curve, thereby explaining why ILS has no impact in this configuration.

During all the simulations, out of hundreds of millions of transmitted packets, no packet loss was observed and the highest packet delay to deadline ratio was 1.07%.

2.5.1.2 Second Configuration: ILS Increases Utilization

In a second configuration, we change the requests generation. The data rate and burst size are now varying between 56.688 Kbps and 64.688 Kbps and 879 bytes and 889 bytes, respectively. That is, the traffic is more bursty. Additionally, the requests are not anymore distributed evenly among the combinations of input link and queue but we generate 10 times more requests from the first input link for the highest priority queue than for all other combinations of input link and queue. In such a way, because more flows will be embedded in the high priority queue, the knee point of the corresponding shaped arrival curve will be shifted towards the right, thereby potentially reducing the maximum burst computation. Besides, since ILS shapes bursts, having more bursty traffic should increase the effect of ILS.

The right diagram of Fig. 2.31 shows the result of the simulation for the second configuration. We observe that ILS now improves the performance of the TBM when the amount of input links is low. This is due to the fact that, when the amount of input link increases, the ratio of requests from the first input link for the high priority queue to the total of requests decreases. Therefore, as increasing the amount of input links leads to a more even distribution of requests among the combinations of input link and queue (as in the first configuration), the performance gain of ILS decreases. This shows that ILS behaves better when the flows at one link are not distributed evenly among the input links. During all the simulations, out of tens of millions of transmitted packets, no packet loss was observed and the highest packet delay to deadline ratio was 0.47%.

2.5.2 Monte Carlo Simulation

The first part of our evaluation confirmed that the TBM and its ILS extension are correct. Further, it has shown that the benefit of ILS grows when the traffic entering a link is not distributed evenly among the incoming links. However, we only observed the impact of ILS on the allowed bursts. In order to observe the impact of ILS on both the allowed bursts and the delay computation, a global network simulation is required. As part of the complete DetServ architecture (Sec. 2.3.4, the performance of the network model depends on the associated components (resource allocation, cost function and routing algorithm) and on the scenario (topology and type of flow requests). As such, with the aim of observing the influence of the network model only, we run a *Monte Carlo simulation* varying the different components (defined in Sec. 2.5.2.1) and scenarios (defined in Sec. 2.5.2.2) used next to the TBM and its ILS extension. In other words, we randomly vary the context in which the model is used in order to isolate their impact on the overall performance of the DetServ architecture.

2.5.2.1 Other Components: Resource Allocation, Routing Algorithm and Cost Function

Resource allocation algorithm. We allocate resources among the queues identically for each link and following the resource allocation algorithm used in the first evaluation (Sec. 2.5.1). That is, we assign the delays 0.487 ms (high priority), 1.437 ms, 3.035 ms, and 4.709 ms (low priority) to the different queues.

Routing algorithms and cost functions. As we will show in chapter 3, we can use existing DCLC routing algorithms for finding a delay-constrained path in the queue-level topology while minimizing a given cost function. Among the plethora of such algorithms available in the literature, we consider *CBF* [Wid94] for its optimality, *LARAC* [Jüt+01] for its good average performance and Dijkstra computing the least-delay (LD) path for its simplicity (we will show the performance of these algorithms in chapter 3). We use different cost functions based on the priority of a queue link, the amount of average flows that can still be embedded in it or a combination of those.

2.5.2.2 Scenario

Topologies. We define two network topologies based on lines and rings, which are typical structures for industrial networks. The two topologies are shown in Fig. 3.1 (the two leftmost topologies). The first topology consists of a ring of size $m + 1$ to which one programmable logic controller (PLC) and m lines composed of n remote input/outputs (I/Os) are attached. The second topology extends the first one by connecting another ring of size $m + 1$ to the former loose ends of the remote I/Os lines. The $(m + 1)$th switch not connected to the lines is then connected to the PLC. Communication is only considered from the remote I/Os to the PLC. Both topologies can be scaled along the two n and m dimensions ($4 \leq n \leq 10, 4 \leq m \leq 10$).

Flow requests. In order to generate a request for a given topology, a random remote I/O is selected to communicate with the PLC. Requests are defined as in Sec. 2.5.1.1.

2.5.2.3 Evaluation Metrics

For a given iteration of the Monte Carlo simulation, i.e., for a given network model (and associated resource allocation algorithm), cost function, routing algorithm and topology, a binary search is started in order to find, for this scenario, the greatest *traffic intensity* for which every request can be embedded. Traffic intensity is defined as the arrival rate of flows multiplied by their average duration (100 seconds, see Sec. 2.5.1.1), which also corresponds to the amount of active flows in the network (when the system converges). The traffic intensity associated to an iteration then corresponds to the maximum traffic intensity that could be reached. The runtime associated to an iteration corresponds to the average runtime of a request routing plus the average runtime of a path registration plus the average runtime of a path deregistration, i.e., to the average runtime of a request processing life cycle that was observed during the complete binary search. The runtime was measured on a machine equipped with an Intel Xeon E5 2690v2 CPU @ 3.00 GHz.

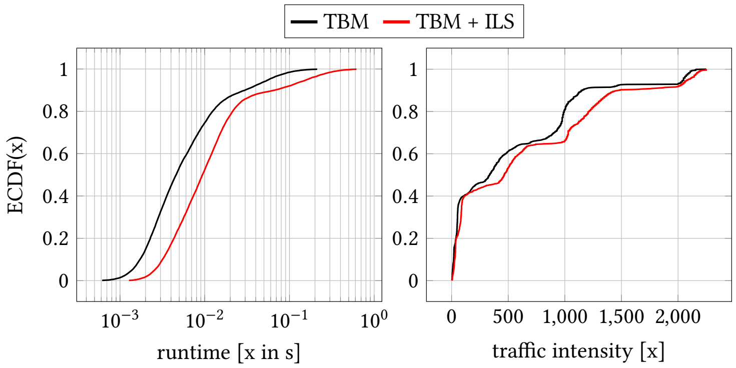


Figure 2.32: Results of the evaluation. The left plot shows the ECDF of the average runtime of one complete request life cycle (routing, embedding, deregistration) for the different models and their corresponding variations with ILS. The right plot shows the ECDF of the traffic intensity that the different models were able to reach. As expected, ILS has a greater impact on the TBM, both in terms of runtime and traffic intensity. We can observe that the TBM with ILS has the potential of reaching a high traffic intensity, but at the cost of a higher runtime.

2.5.2.4 Results

Fig. 2.32 shows the results of the Monte Carlo simulation. The left and right plot show the empirical cumulative distribution function (ECDF) of, respectively, the runtime and the traffic intensity for the MHM with and without ILS.

Runtime. The runtime of the TBM is highly affected by the introduction of ILS (slowed down by a factor of around 2). This was expected and is due to the increased complexity for computing horizontal and vertical deviations when introducing ILS to the TBM. However, the runtime stays lower than 350 ms in 99% of the cases and never exceeds 620 ms, which corresponds to a single-threaded worst-case performance of 1.6 requests per second, which is a reasonable performance for industrial applications.

Furthermore, because the runtime shift between the two versions of the TBM stays roughly equal, Fig. 2.32 clearly shows that the network model is the main driver for the runtime of the complete DetServ system.

Traffic intensity. We observe that the introduction of ILS brings a performance increase to the TBM. While we have seen that the runtime is mostly influenced by the network model, we observe that this is not true for the traffic intensity. Indeed, the slope of the traffic intensity ECDFs vary differently, which means that other components used in the Monte Carlo simulation have a significant impact on the performance of the model. For example, though ILS improves on the performance of the TBM, the minimum and maximum achieved traffic intensities are equal. Generally, we observe that there is a trade-off between low runtime and high traffic intensity. This was expected, as more precise models require more complex computations.

2.6 WDetServ: Support for Hybrid Wired/Wireless Networks

Based on the DetServ architecture described in the previous sections and in [GVK17]; [Guc18], we presented wireless deterministic services (WDetServ) in [Zop+18], a system for additionally supporting wireless hops. This constitutes the first framework for seamless provisioning of predictable latency in arbitrary hybrid wired/wireless networks. In this section, we briefly describe how this can be achieved.

Industrial control systems are foreseen to operate over hybrid wired/wireless networks. While controllers responsible for automation process control decisions will be deployed in the wired network, sensors and actuators will be deployed in a wireless sensor network (WSN). To support the QoS level needed for control systems, an E2E delay bound and a target reliability must be provided in both wireless and wired domains. The target reliability defines the percentage of sent packets that have to reach the destination within the delay bound. Indeed, unlike wired links which are assumed to be 100% reliable, wireless links are naturally subject to packet loss and an additional *reliability* metric must be introduced.

The model in the DetServ architecture described in Sec. 2.3.4 is responsible for ensuring that, if the routing procedure appropriately uses its interface, the requirements of all applications will be met at any time. Besides the five model functions described in Sec. 2.3.5, the network model must additionally provide the routing procedure a topology on which to perform path finding. In order to provide support hybrid wired/wireless networks, it is sufficient to provide a network model for the wireless domain that satisfies these requirements.

We accordingly propose a WDetServ model, a network model for TDMA-based wireless communications. We consider a setup in which WSN nodes are connected to the wired infrastructure via wireless gateways (WGs) in a star topology fashion, setup particularly common in dense industrial scenarios [Zop+18]. We assume that a WDetServ model operates independently for each WG in the network. This can be provided by ensuring that motes connected to different WGs do not interfere with each other, what can be provided through appropriate planning or coordinated scheduling [Zop+18]. As a result, while the wired part of the network, including the WGs, is modeled by a unique DetServ model, the different wireless sub-networks are modeled by independent WDetServ models.

Analogously to the queue-level topology exposed by a DetServ model (Sec. 2.3.3), a WDetServ model exposes a so-called *frame size link topology* to the routing procedure. Every TDMA wireless link is modeled as a set of directed edges defining the different TDMA sub-frame sizes that can be used. The size of every sub-frame defines the maximum delay and reliability that can be achieved using this sub-frame. Indeed, since a smaller sub-frame size reduces the number of transmission opportunities, it ensures a smaller delay at a price of a reduced reliability. The five model functions defined in Sec. 2.3.5 can then be implemented as follows.

- GETDELAY. The deterministic worst-case delay of a packet transmission using a given sub-frame size can be easily computed based on the sub-frame duration d and the maximum number B of packets to be sent at once as $(B + 1)d$ [Zop+18].

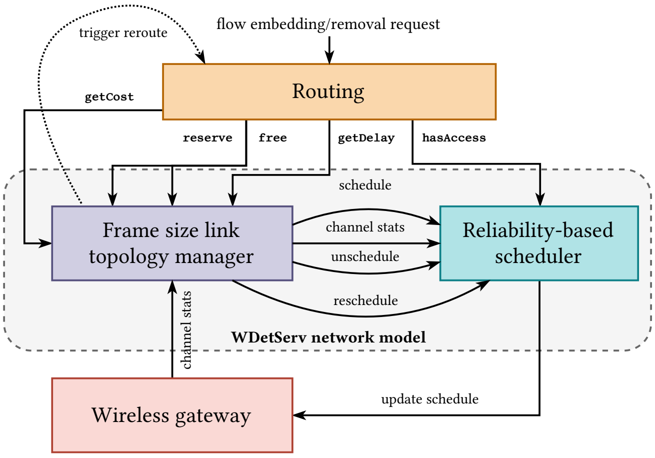


Figure 2.33: The WDetServ CP architecture. WDetServ fullfils the DetServ interface defined in Sec. 2.3.5.

- HASACCESS. A wireless scheduler checks that a schedule providing the target reliability with the given sub-frame size is available, that enough data rate is available (i.e., that the rate of the application is lower than its minimum packet size divided by the sub-frame duration d), and that the packet size of the application is smaller than the maximum packet size that can be sent in a single TDMA slot. As wired links are assumed to provide 100% reliability, the E2E reliability of a path corresponds to the reliability of its wireless hop and this check is sufficient for providing E2E reliability.

- RESERVE (resp. FREE). The wireless scheduler schedules (resp. unschedules) the given application according to its reliability level to the sub-frame corresponding the given edge (*schedule* (resp. *unschedule*) arrows in Fig. 2.33). The result is then forwarded to the WG in order to notify the involved wireless devices (*update schedule* arrows in Fig. 2.33). In order to evaluate the reliability of a given schedule, channel statistics must be gathered. This information is obtained by the WG and forwarded to the scheduler (*channel stats* arrows in Fig. 2.33). Upon any channel statistics change, applications reliability requirements can get violated. In this case, the scheduler tries to reschedule the applications such that their respective target reliability values are met again (*reschedule* arrow in Fig. 2.33). If the rescheduling succeeds, the WG receives and forwards the new schedules to the devices. If the rescheduling fails, E2E rerouting is necessary (*trigger reroute* arrow in Fig. 2.33). Indeed, changing the paths taken in the wired infrastructure will potentially allow to use other sub-frame sizes on the wireless link, thereby potentially making the application schedulable again. The design and evaluation of such a scheduler is part of the contributions of [Zop+18] but is omitted in this thesis.

- GETCOST. The implementation can depend on any parameters such as channel statistics and number of connected devices.

In order for the routing procedure to operate on a single topology, the topologies provided by both the DetServ and WDetServ models have to be merged. As each WG appears once in the DetServ queue-level topology and in the WDetServ frame size link topologies, the final topology is simply obtained by merging the individual topologies at the different WGs. While embedding flow requests, the routing procedure uses the DetServ or WDetServ model depending on whether the edge it visits is a queue-level topology edge or a frame size link topology edge.

2.7 Summary

In this chapter, we provided a detailed description of DetServ, an SDN-based architecture and an accompanying network model for the provisioning of real-time QoS in programmable networks.

The architecture is based on a centralized controller receiving flow embedding requests from applications. A *routing procedure* is then responsible for finding a path satisfying the delay requirement of the request. For doing so, the routing procedure relies on a *network model* responsible for providing latency values at each hop and for performing access control. The network model itself relies on a *resource allocation* module that allocates resources to individual queues in the network, an *access control* module that ensures no more than the available resources are used, a *resource reservation* module that keeps track of the per-queue resources usage, and a *cost function* that is responsible for driving the routing procedure towards paths where the application will consume less resources.

The presented TBM fixes a maximum delay for each queue and ensures that the worst-case delay bound for traversing a queue always remains lower than the assigned maximum delay. Through packet-level simulations, we confirmed that the guarantees provided by the model are indeed never violated. Our evaluations have additionally shown that the runtime cost of the DetServ architecture and its TBM model is reasonable for industrial scenarios: the total request processing time never exceeds 620 ms.

One major benefit of the proposed model is that it can be used with simple commodity switches supporting priority scheduling and any SDN protocol providing standard enqueuing and forwarding primitives, e.g., OF 1.0 [Ope09].

This chapter further layed the ground for the remainder of this thesis. In chapter 3, we investigate in detail the routing procedure of the DetServ architecture. In partcular, we investigate how it can operate in coordination with the interface provided by the DetServ network model and what influence the latter has on the optimality and completeness of existing SoA routing algorithms. In chapters 4 and 5, we conduct detailed SDN forwarding hardware measurement campaigns and observe that both the architecture and model cannot be used directly *as is* with existing forwarding hardware. In both chapters, we then investigate what are necessary and possible amendments to the architecture and model so that predictable latency can practically be provisioned in data center networks (chapter 4) and in small networks (chapter 5), i.e., networks with low-bandwidth low-capacity forwarding elements.

Chapter 3

Optimization of the Path Selection Strategy

The previous chapter described an architecture and a network model for the provisioning of predictable latency in programmable networks. As part of this architecture, a *routing procedure* is responsible for path finding. Using the interface provided by the network model (Sec. 2.3.5), the routing procedure is responsible for embedding online flow requests by finding paths in the network that satisfy the individual latency requirements of the applications.

The routing procedure has to provide a minimum set of features. First, the algorithm should only return correct paths, i.e., paths with a delay lower or equal to the latency requirement of the flow to be embedded. Second, the algorithm should be able to provide a solution of sufficient quality. Quality is here measured in terms of resources utilization: the flow should be embedded in a way that leaves the most amount of resources available for subsequent flows. In order to measure quality, the routing procedure relies on the cost function of the network model. This cost function is supposed to transform the network state (e.g., data rates usage, buffer consumption, already embedded flows) into a cost metric for each queue, the minimization of which maximizes the total number of flows that the network can accept. On the queue-level topology (see Sec. 2.3.3) of the network, the routing algorithm must hence find a delay-constrained least-cost (LC) path using the getCost and getDelay methods of the network model. Algorithms that solve this problem are called delay-constrained least-cost (DCLC), or more generally constrained shortest path (CSP), algorithms. At the same time, in order to serve requests as fast as possible, the algorithm should achieve a short runtime. Finally, the algorithm should ideally be complete. Indeed, connection requests should not be rejected if they can actually be accommodated.

The main contribution of this chapter is the in-depth study of this routing procedure, including the design of new algorithms and the performance evaluation of SoA and new routing procedures. First, we conduct a performance evaluation of available algorithms that satisfy the above-mentioned requirements. We evaluate the performance of 26 SoA algorithms and provide insights into their behavior in different scenarios. This analysis has been done as part of a bilateral collaboration, the results of which have already been published in a more extended version in a previous doctoral thesis [Guc18]. We here present only the main findings of the study. Our results show that the performance of algorithms greatly varies based on the network setup and specific routing request but we identify a small set of algorithms as best-fit for most scenarios: *LARAC* and *H_MCOP*. These algorithms are based on the Lagrangian relaxation mathematical method and rely on multiple runs of a Dijkstra [Dij59] shortest path (SP) subroutine. Second, based on this observation that Dijkstra forms the basis of many more complex routing algorithms, we investigate how such subsequent Dijkstra runs relate to each other. We observe that additional information can be provided to these SP subroutines to reduce their runtime while not impacting the result of the routing algorithm. We accordingly propose BD, a search space reduction technique for the SP subroutines. BD allows the Dijkstra runs to terminate earlier, when it is known that further exploration results will not be used by the main CSP routing algorithm. Through comprehensive performance evaluations, we show that BD allows, in favorable cases, to reduce the runtime of some algorithms by 96% on average, without impacting their output. Third, based on the observation that the best algorithms are based on the Lagrangian relaxation, we apply this method to design Lagrange relaxation based aggregated cost for specified nodes (LARAC-SN), an extension of *LARAC* that is able to solve routing requests that, in additional to their delay requirement, specify intermediate nodes that must be visited. That scenario is of major importance in SFC environments, where a flow has to traverse a set of service functions, or VNFs, that perform some intermediate processing (e.g., security functions). We further propose mole in the hole (MITH), a graph transformation algorithm that enables any routing algorithm to be forced to visit intermediate nodes. While LARAC-SN is sub-optimal, we show that MITH, in combination with an optimal CSP algorithm, can ensure optimality but at the cost of a higher runtime. Last but not least, we observe that the DetServ interface used by the routing procedure leads to link metrics that violate the optimal substructure property (OSP) on which SoA algorithms rely to ensure their completeness and/or optimality properties. We investigate how algorithms can be adapted to circumvent this issue and propose solutions, namely edge-based Dijkstra (EBD) and graph transformation algorithm (GTA), to restore the completeness and/or optimality of SoA routing algorithms when used as part of the DetServ architecture.

Content and outline of this chapter. Sec. 3.1 provides a short introduction on routing terminology. Sec. 3.2 presents the *Dijkstra*, *Bellman-Ford (BF)*, *Yen's*, and *Chong's* algorithms, SoA algorithms for computing SPs in a graph that are used extensively throughout this chapter. Sec. 3.3 investigates the performance of SoA CSP routing algorithms for providing predictable latency within the DetServ architecture and is based on content from [Guc+17]. Then, Sec. 3.4 proposes BD, an optimization of the Dijkstra algorithm when it is run as a subroutine of an overlay routing algorithm such as those evaluated in Sec. 3.3. This section relies on content from [Van+19c]. Sec. 3.5 proposes the LARAC-SN and MITH algorithms, adaptations of SoA algorithms to route requests with delay requirements through VNF chains. Finally, Sec. 3.6 investigates the completeness and/or optimality loss of SoA routing algorithms when used with metrics such as those defined by the TBM DetServ network model and proposes the EBD and GTA algorithms to recover these properties in some cases. These last two sections are respectively based on [Van+18b] and [Van+18a].

3.1 Background: Terminology and Definitions

In this section, we introduce the general routing terminology and definitions used throughout this chapter. First, in Sec. 3.1.1, we define routing metrics and in particular constraint and optimization metrics and additive, multiplicative and concave metrics. Based on these definitions, we define the completeness and optimality properties of routing algorithms. Second, in Sec. 3.1.2, we define different routing optimization problems based on the number of constraint and optimization metrics considered. Finally, Sec. 3.1.3 mathematically formalizes the previous problems.

3.1.1 Routing Metrics

Independently of the type of route (e.g., a single path, several disjoint paths, or a tree) that a routing algorithm has to find, there is usually more than one possible solution to a given problem. In order to prefer one solution over another, or to provide additional requirements regarding the solutions to be accepted, so-called *routing metrics* are introduced. Each metric defines a value, referred to as a *metric value*, for each edge of the subject graph. These values can be updated for each routing request. The metrics can then be used in three different ways for selecting one or several of the available solutions.

Local constraint metrics. First, the edges that can be used by the solutions can be restricted to those satisfying a given condition based on a metric value. We refer to such metrics as *local constraint metrics*. For example, in order to ensure enough bandwidth is available for a given video stream, one might want to use only links whose bandwidth is greater than the bit rate of the video stream.

Global constraint metrics. Secondly, in order to further restrict the set of solutions that can be returned, the values of a metric at each edge of a solution can be combined using a so-called *link combination operator* [Bau+07], e.g., the sum or the multiplication, the result of which must satisfy a given constraint. We refer to such metrics as *global constraint metrics*. For example, in order to ensure that the packets of a unicast flow with strict latency requirements arrive on time, one might want to find a path for which the sum of the delays of each of its constituting edges is lower than the requirement of the flow.

Global optimization metrics. Finally, in order to rank solutions and search for the preferred one(s), the values of a metric at each edge of a solution can be combined using a given link combination operator whose result is used for ordering the solutions. We refer to such metrics as *global optimization metrics*, or simply *optimization metrics*. For example, in order to ensure that data from a unicast request is transferred as fast as possible, one might want to find the path for which the sum of the delays of each of its constituting links is minimal.

Based on these definitions, an algorithm is said *complete* if it always finds a solution, if one exists, satisfying both the local and global constraints. In case of a single global optimization metric, an algorithm is said *optimal* if the solution it finds is always the optimal one. Completeness does not imply optimality.

Depending on the link combination operator used by a global metric, three different categories of metrics can be defined: *additive*, *multiplicative*, and *concave* metrics. The E2E values of these three metric categories are, respectively, the sum, the product, and the minimum (or the maximum) of the metric values for the individual links. Delay, packet loss probability, and bandwidth are examples of additive, multiplicative and concave metrics, respectively.

3.1.2 Optimization Problems

For presentation reasons, we now focus on the problem of finding a single *path* between a single source and a single destination. That corresponds to the problem of routing a unicast flow. The concepts and definitions can however be anagolously applied to any other type of route (e.g., a tree).

Problem type (acronym)	**Nb. of glob. opt. metrics**	**Nb. of glob. const. metrics**
shortest path (SP)	1	0
constrained shortest path (CSP)	1	1
multi-constrained shortest path (MCSP)	1	M
multi-constrained path (MCP)	0	M

Table 3.1: Conceptual comparison of routing problem types based on the number and type of routing metrics involved.

Depending on the number and type of global constraint metrics, different algorithmic problems can be defined[1]. The most commonly encountered problems are defined as follows and listed in Tab. 3.1.

- shortest path (SP): The path has to minimize a unique global optimization metric.
- constrained shortest path (CSP): The path has to minimize a global optimization metric while keeping a global constraint metric below a prescribed *bound*.
- multi-constrained shortest path (MCSP): The path has to minimize a global optimization metric while keeping multiple global constraint metrics below individual bounds.
- multi-constrained path (MCP): MCSP problem without optimization metric, i.e., the path only has to keep multiple global constraint metrics below prescribed bounds.

These problems can be extended to k path versions that find k distinct paths. These are the k shortest paths (kSP), k constrained shortest paths (kCSP), k multi-constrained shortest paths (kMCSP), and k multi-constrained paths (kMCP) problems. For the kSP, kCSP and kMCSP problems, the problem is to find the k best paths according to the optimization metric. For the kMCP problem, the problem is to find any k paths satisfying the global constraint metrics.

It is also possible to define multi-objective problems that consider several global optimization metrics [Deb01]; [GGT10]; [MA04]. These multi-objective problems are however out of the scope of this thesis. We focus on single-objective problems. The optimization metric is often referred to as the *cost* metric.

[1] The number of local constraint metrics can vary because it does not impact the algorithmic problem to solve. Indeed, a local constraint metric can simply be handled by, before routing, removing from the graph the edges not satisfying the constraint.

3.1.3 Mathematical Formulation

Consider routing to be performed on a network graph $G = \{V, E\}$, whereby V is the set of vertices (network nodes) and E is the set of directed edges (with $|E|$ denoting the number of edges in the network). The vector of costs of the edges is denoted by c, $\mathrm{c} \in \mathbb{R}_{+}^{|E|}$. Let d, $\mathrm{d} \in \mathbb{R}_{+}^{M}$, denote a vector with M elements that represent the bounds for the global constraint metrics. Let D, $\mathrm{D} \in \mathbb{R}_{+}^{M \times |E|}$, denote a matrix of the global constraint metrics values for the individual edges. Let P_{sd}, $P_{sd} \subseteq \{0,1\}^{|E|}$, denote the set of paths from source node s to destination node d (whereby a value of 1 for an edge means that the edge belongs to the path). For additive metrics, the SP, CSP, and MCSP problems can be mathematically formulated as:

$$z_{\mathrm{opt}} = \min_{\mathrm{x} \in P_{sd}} \ \mathrm{c}^{\mathsf{T}} \mathrm{x} \tag{3.1}$$

$$\text{s.t.} \quad \mathrm{D}\mathrm{x} \leq \mathrm{d}. \tag{3.2}$$

The SP, CSP, and MCSP problems correspond to the cases $M = 0$, $M = 1$, and $M > 1$, respectively.

An *optimal* algorithm always finds the optimal path with cost z_{opt}. A *heuristic* is an algorithm that finds a possibly sub-optimal path, i.e., a path with cost $z' \geq z_{\mathrm{opt}}$. The optimality gap (OG) of an algorithm, measured in %, is defined as

$$\mathrm{OG} = \frac{z' - z_{\mathrm{opt}}}{z_{\mathrm{opt}}} \times 100. \tag{3.3}$$

An optimal algorithm therefore always has an OG of 0 %.

3.2 Background: Shortest Path Algorithms

In this section, we provide background material on basic routing algorithms that are often used and referred to throughout this chapter.

The *Dijkstra algorithm* [Dij59] computes the SP from a single source node to all other nodes (i.e., a shortest path tree (SPT)) in a graph with non-negative edge costs by performing a breadth-first search (BFS) starting from the source node. The algorithm maintains a priority queue containing a set of partial paths, i.e., paths starting from the source node and reaching an intermediate destination node which is not the ultimate destination. At each iteration, it takes the LC path among the paths in the queue and generates n new paths by extending this partial path with the n outgoing edges of the node at which the given path terminates. Among those paths, only paths with lower cost than the current LC path in the queue towards the same destination are added back to the queue. That is, the Dijkstra algorithm *relaxes* based on the cost values. Doing so, the algorithm keeps track of the best path found to each node. Nodes with LC distance from the source node are expanded first, thereby ensuring that any node has to be visited only once. The SP to a node is found as soon as the algorithm pops this node from the priority queue.

The *BF algorithm* [Bel58]; [For56]; [Moo59]; [Shi54] also computes the SPT in a graph, including graphs with negative edge costs. The algorithm maintains the current best path found to each node and runs $|V| - 1$ (where $|V|$ is the number of nodes in the network) iterations updating, for each node, the current best path to all neighbor nodes based on the current best path to the presently considered node. Since the path to any node is at most $|V| - 1$ hops long, all SPs will eventually be found. If an iteration yields no update, the algorithm can be immediately terminated, as subsequent iterations will not lead to any change. Also, if the cost of the current best path to a node has not changed since the last iteration, then the outgoing edges of this node can be skipped since they will not lead to any new changes.

Both algorithms can be adapted to compute the SP from any node to a single destination. These versions are called the *reverse Dijkstra* and *reverse BF* algorithms, respectively, and are simply obtained by considering incoming edges rather than outgoing edges when going from one node to the next node(s).

The *A* algorithm* [HNR68] is an improvement to the Dijkstra algorithm for finding a single-destination SP. It introduces a so-called *guess function*. At each node, this guess function provides a guess for the cost of the SP from this node to the destination node. Paths out of the priority queue with least *projected cost* (i.e., sum of the current cost to the last node of the path and of the guess value at this node) are expanded first. To ensure the correctness and optimality of the A* algorithm, the guess values have to be lower than the real values. The closer the guess values are to the real values, the faster the A* algorithm will reach the destination. At one extreme, the A* algorithm with an exact guess function will directly traverse the SP to the destination. At the other extreme, the A* algorithm with a guess function of zero corresponds to the original Dijkstra algorithm. A straightforward guess function corresponds to the least-hop count multiplied by the cost of the LC edge in the graph. Such a guess function has to be recomputed upon any topology change but can be precomputed offline if the topology does not change.

We will consider the Dijkstra algorithm for finding an SPT and the A* algorithm for finding an SP as underlying algorithms for the different CSP algorithms. Indeed, they perform generally better [CGR96].

Yen's algorithm [Yen71] solves the kSP problem. First, the SP is found using a traditional SP algorithm. Then, subsequent SPs are found based on the knowledge of this initial path. The $(k+1)$th SP is found by starting at intermediate nodes of previously found paths, blocking the next edge in the path to force the algorithm to find another path, and running an SP algorithm from there. The LC path out of all these new paths is the $(k+1)$th SP.

Yen's algorithm does not need to know the value of k when starting, i.e., it is an iterative k shortest paths (ikSP) algorithm [Guc+17]. In contrast, *Chong's algorithm* [Cho95] requires k to be known in advance, i.e., it is a static k shortest paths (skSP) algorithm [Guc+17]. The algorithm is then identical to the Dijkstra (or A* in our case) algorithm, but keeps track, for each node, of the current k best paths found. Once the destination(s) has (have) been visited k times, the algorithm can stop.

3.3 Evaluation of the Available Routing Algorithms

Generally speaking, the routing procedure of the DetServ architecture has to find a path that

- minimizes the sum of the GETCOST values obtained from the network model,

- keeps the sum of the getDelay values obtained from the network model lower than the E2E latency requirement of the request, and
- only uses queues for which the hasAccess model function returned *true*.

That means that the getCost function is an additive global optimization metric, the getDelay method is an additive global constraint metric, and the hasAccess function is a local constraint metric (as defined in Sec. 3.1.1). This corresponds to a CSP problem. With such additive metrics, this problem is also commonly referred to as delay-constrained least-cost (DCLC) routing problem.

Generally, the CSP problem is *NP-complete* [AMO93]. Therefore, there is a fundamental trade-off between OG and low runtime and, given the importance of routing for communication networks, many routing algorithms and heuristics have been specifically designed for particular network settings or applications. However, these algorithms have been evaluated separately in individual studies, making it difficult to select the most appropriate algorithm for the routing procedure of the DetServ architecture. This is the gap we fill in this section. First, in Sec. 3.3.1, we briefly list the 26 SoA CSP algorithms surveyed in [Guc+17]. Second, in Sec. 3.3.2, we evaluate these algorithms in different settings. Third, in Sec. 3.3.3, we interpret the results from Sec. 3.3.2 to identify the best algorithm to use for the DetServ architecture. While the best performing algorithm depends on the desired optimality/runtime trade-off and on the specific network and routing request, we identify the *LARAC*, *H_MCOP*, *SSR+DCCR* algorithms and their variants as the best performing algorithms on average. Finally, Sec. 3.3.4 summarizes the results of this section.

3.3.1 List of Available Algorithms

We list the 26 SoA CSP algorithms[2] extensively surveyed in [Guc+17]. Note that we focus on algorithms that are able to accommodate real values for the metrics and that can run with any arbitrary cost function as returned by the DetServ network model. We classify the algorithms into the five categories described below based on their underlying routing strategy.

- *Elementary algorithms.* Very simple algorithms based on simple procedures that accordingly achieve either high runtime or high OG.
- *Algorithms based on a priority queue.* Like the *Dijkstra* algorithm (see Sec. 3.2), algorithms that are based on a priority queue that stores intermediate or partial paths during the search.
- *Algorithms based on the Bellman-Ford (BF) SP algorithm.*
- *Algorithms based on the Lagrange relaxation.* In order to solve a constrained optimization problem, the Lagrange relaxation technique removes some constraints of the original problem and introduces them in the optimization objective. Many algorithms are based on this technique to solve the CSP problem with a series of SP runs.
- *Algorithms that follow the LC and LD paths in the network.* Algorithms that build the LC and LD trees from all nodes to the destination and use a combination of the resulting paths to build a solution to the CSP problem.

Algorithm	Number of runs of underlying algorithms					Optimal	Complete	Param.
	*ik*SP	*sk*SPT	*sk*SP	SPT	SP			
Elementary algorithms								
LDP [Guc+17]					1		✓	
FB [LHH95]					(1 ⟶ 2)		✓	
*k*SPMC [Guc+17]	1					✓	✓	
Algorithms based on a priority queue								
CBF [Wid94]						✓	✓	
A*Prune [LR01]						✓	✓	
Algorithms based on Bellman-Ford (BF)								
DCBF [JV01]				1			✓	
*k*DCBF [JV01]		1					✓	k_d, k_c
DEB [CA03]							✓	
Algorithms based on the Lagrange relaxation								
LARAC [AN78]; [HZ80]; [BG96]; [Jüt+01]					≥ 1		✓	*MD*
LARACGC [HZ80]	(0, 1)				≥ 1	if $\delta = 0$	✓	δ
SCRC [SCC07]	(0, 1)				(1, 2)	✓	✓	
*k*LARAC [JV01]			≥ 1				✓	k
H_MCOP [KK01]				1	(0, 1)		✓	
*k*H_MCOP [KK01]			(0, 1)	1			✓	k
NR_DCLC [Fen+02a]				≥ 0	≥ 1		✓	
MH_MCOP [Fen+02b]				≥ 1	≥ 0		✓	
E_MCOP [Fen+02b]	(0 ⟶ 2)				(1 ⟶ 2)	✓	✓	
DCCR [GM03]			(0, 1)		(1, 2)		✓	k
SSR+DCCR [GM03]			(0, 1)		(1 ⟶ L + 2)		✓	L, k
Algorithms based on LC and LD paths								
DCUR [SRV97]; [RS00]				(1, 2)			✓	
DCR [SL98]				(0, 1)	1		✓	
IAK [IAK98]				1	(0, 1)		✓	
SMS-RDM [SMM98]				1			if $p \geq \Delta(G)$	p
SMS-CDP [SMM98]				2			if $p \geq \Delta(G)$	p
SMS-PBO [SMM98]							if $p \geq \Delta(G)$	p
SF-DCLC [LLF05]				(1, 2)			✓	

Table 3.2: Comprehensive list of CSP and MCSP algorithms that can be used as part of the DetServ architecture. The algorithms are categorized according to the underlying algorithmic strategy into algorithms based on priority queues, Bellman-Ford, Lagrange relaxation, as well as LC and LD paths. For each algorithm, we indicate key characteristics, including optimality property and the accepted parameters, as well as the number of underlying algorithm runs, e.g., SP runs. When the exact number of runs depends on the specific scenario, the possible numbers of runs are indicated through a comma-separated list or a range (with the arrow (⟶) symbol) within parentheses. Unbounded numbers of runs are indicated with the greater or equal (≥) sign. $\Delta(G)$ indicates the maximum node degree in the graph G.

The list of the algorithms is given in Tab. 3.2. While most algorithms are complete, only a few algorithms are optimal. We will see in Sec. 3.3.2 that optimal algorithms exhibit a very high runtime. We will also see that the main driving factor for the performance of the algorithms is the number and type of runs of underlying algorithms. This information is given in the table for each algorithm. For example, the LDP algorithm runs a single SP search, the LARAC algorithms runs a series of SP searches, and the *k*H_MCOP runs a single SPT search and none or one *sk*SP search. A detailed description of the algorithms can be found in [Guc+17] or in the original publications referenced in the table.

3.3.2 Performance Evaluation

Generally, the performance of an algorithm depends on the specific scenario in which it is executed. In order to evaluate the behaviors of the different algorithms across a wide set of scenarios, we introduce four critical dimensions. First, we define four *topologies* which we describe in Sec. 3.3.2.1. A topology describes both the underlying structure of the network and the nodes that communicate with each other in the network. Second and third, we scale these topologies in two directions. Fourth, we distinguish requests based on the level of strictness of the delay constraint, see Sec. 3.3.2.2. Sec. 3.3.2.3 presents the evaluation procedure and the metrics used. Sec. 3.3.2.4 presents the evaluation results for the fourth dimension, i.e., the behavior of the algorithms for different tightness levels of the delay constraint. Sec. 3.3.2.5 then focuses on the three first dimensions. Sec. 3.3.2.6 then investigates the average behavior of the algorithms across all the dimensions. We here present only very condensed results. Finally, Sec. 3.3.2.7 discusses the absolute runtime (in ms) of the algorithms on a particular hardware architecture. Further results and more detailed explanations are available in [Guc+17]. Besides, we have made the entire set of raw results and graphs for all the algorithms available online at [Van19c].

We implemented all the 26 algorithms from Tab. 3.2 in Java 8 and, for each of them, ran our evaluation procedure[3]. We will identify parameterized algorithms by the name of the original algorithm to which we append the dash-separated parameter values in the same order as in Tab. 3.2. For example, LARACGC with $\delta = 25\%$ will be referred to as *LARACGC-25*.

3.3.2.1 Topology and Scaling

As first dimension for our evaluation, we define four topologies (shown in Fig. 3.1) based on three different base topologies. We focus on industrial topologies, where flows with predictable latency requirements are common. Nevertheless, the topologies we define are also common in and representative of data center, metro, grid, and enterprise networks. All topologies can be scaled according to two scale parameters m and n that represent the size of the topology layout, as illustrated in Fig. 3.1 and defined in detail in the following for the four different topologies. The second and third dimensions of our evaluation correspond to varying the two scale parameters m and n from 4 to 13, thereby defining 100 different scalability levels. The four topologies are described below.

[2] This includes MCSP algorithms since these algorithms are a subset of the CSP algorithms.

[3] We acknowledge that the results may be subject to our specific implementations; however, we tried to be fair and optimize all implementations equally and as much as we could.

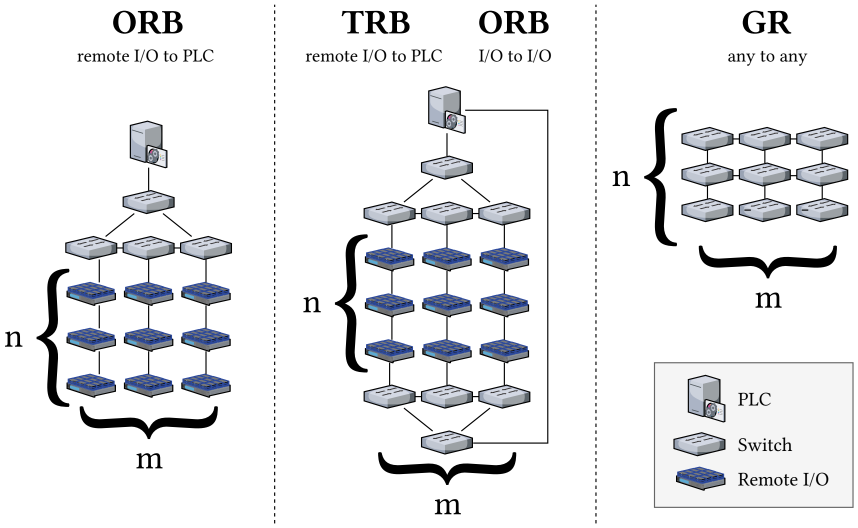


Figure 3.1: The four topologies considered in the evaluation are based on three different base topologies which can be scaled in two different directions. The topologies also differ in terms of allowed communications patterns: remote I/O to PLC, remote I/O to remote I/O, or any to any.

- One ring bottleneck (ORB): The ORB topology consists of a base ring of $m+1$ switches. A PLC is connected to one switch of this ring. A branch composed of a series of n remote I/O nodes, e.g., sensors, is connected to each of the other m switches of the ring. Thus, there are a total of mn I/Os. Remote I/Os have an internal switch allowing traffic to flow along the branches. Thus, remote I/Os act as traffic sources as well as traffic forwarders, which is common in sensor networks and industrial networks [KOK14]; [GH13]; [GJF13a]; [STV12]. Traffic is only considered from the remote I/Os to the PLC.

- Two rings bottleneck (TRB): The TRB topology extends the ORB topology with an additional ring consisting of $m+1$ switches. The $m+1$ switches connect the loose (bottom) ends of the m branches of remote I/Os (of the ORB topology) to the PLC. Traffic is still considered only from the remote I/Os to the PLC.

- Two rings random (TRR): The TRR topology is the same as the TRB topology, but traffic is now considered between any pair of remote I/Os. As the remote I/Os, the PLC is able to forward traffic not destined for it.

- Grid random (GR): The GR topology is a grid of width m and height n. In the GR topology, traffic is considered between any pair of nodes.

We do not consider random topologies generated based on models, such as the *Waxman model* [Wax88]. Instead, striving for a fair and reproducible evaluation, we only use deterministic topologies.

Each directed link is considered to have four output priority queues and routing is then performed on the corresponding queue-level topologies (see definition in Sec. 2.3.3). The costs and delays of the four queue-link edges are respectively set to 2 and 0.48 ms, 1.5 and 1.26 ms, 1.33 and 2.83 ms, and 1.25 and 7.55 ms.

3.3.2.2 Delay Constraint Tightness

The delay constraint of routing requests can range from loose values for which the LC path is feasible to tight values for which no feasible path exist. Within this range, we define seven subranges of equal size, which we refer to as *delay levels*.

3.3.2.3 Evaluation Procedure and Metrics

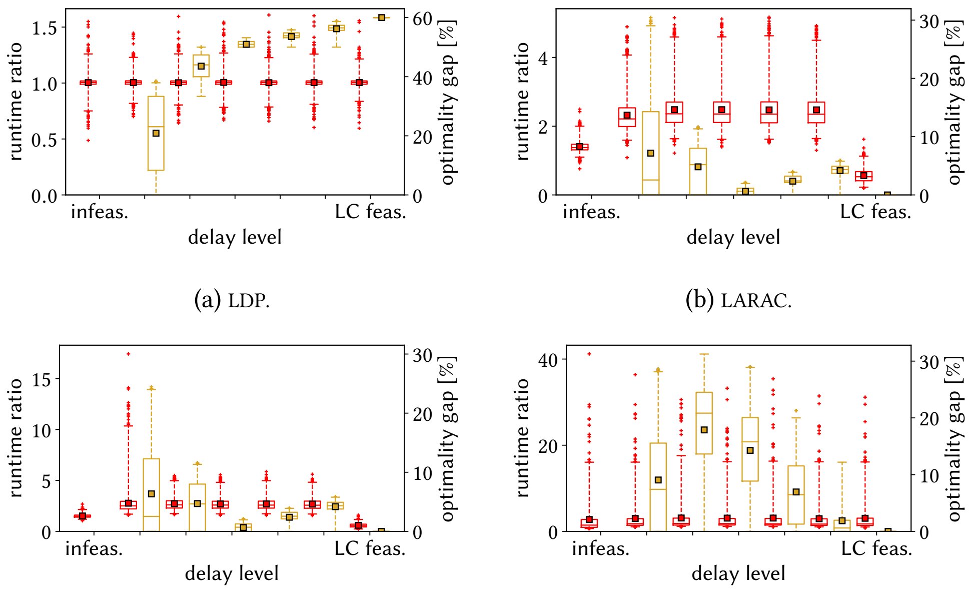
We evaluate each algorithm along the four dimensions defined above. For each particular topology and combination of the scale parameters m and n, we sequentially simulate 20,000 routing requests. The first 1000 requests are used as warm-up for the Java HotSpot optimizer and their results are not considered. For each request, the source and destination are generated uniformly randomly from the possible set of combinations defined by the topology and scale parameters. The delay constraint is distributed uniformly randomly among the seven delay levels and then uniformly randomly within the selected delay level.

For a given algorithm under test (AUT) and request, we run three algorithms. First, we run CBF in order to obtain the cost z_{opt} of the optimal solution. Second, we run the AUT. The OG of the AUT is then evaluated according to Eqn. 3.3. Third, we run a LD search using A* (which is then equivalent to a run of the LDP algorithm, as the latter always returns the LD path). We define the runtime of the AUT divided by the runtime of the LD search as the runtime ratio of the AUT. This normalization allows to filter out runtime variations due to the varying runtime behaviors of the testing machines (caused by OS tasks or the Java garbage collector execution). Indeed, both algorithms are run one after the other, i.e., within a short time window during which the runtime behavior of the testing machine can be assumed to be constant.

We found that kSPMC, A*Prune, LARACGC, SCRC, E_MCOP, and the three SMS variants were not able to complete the evaluation in a reasonable amount of time compared to CBF. This leads to our first observation that algorithms using an ikSP algorithm to reach optimality have a very long runtime. Indeed, the considered queue-level topologies are dense with high numbers of possible paths. Thus, the number of paths to discover until reaching optimality is also high, yielding intractable runtimes. A*Prune, and the three SMS variants are not based on an ikSP algorithm but their structure is such that, if their initial search direction is not the correct one, they have to explore a high number of paths to reach the destination.

3.3.2.4 Fingerprints: Influence of the Delay Constraint Tightness

We analyze the fourth dimension using so-called *fingerprint* graphs (Fig. 3.2). The fingerprint graph for a given combination of topological and scale parameters m and n, shows the distribution of the runtime ratio (left, in red) and OG (right, in yellow) of an algorithm for the seven different delay levels (loose levels on the right and tight levels on the left). Since we have four different topologies with 100 different scalability levels (combinations of m and n values), each algorithm has 400 fingerprint graphs. Nevertheless, we observed that the shapes of all fingerprint graphs for a given algorithm are similar; Fig. 3.2 shows fingerprints for the GR topology with scale parameters $m = n = 10$. That is why we refer to these graphs as *fingerprints*: they nearly uniquely identify an algorithm based on its behavior and are (nearly) always the same for a given algorithm. Only the absolute values vary



(a) LDP.

(b) LARAC.

(c) LARACGC-25.

(d) H_MCOP.

Figure 3.2: Fingerprints for selected CSP algorithms. These graphs show, for the GR topology with $m = n = 10$, the runtime ratio (plotted in red on the left) and OG (in yellow on the right) of selected algorithms for the seven different delay levels (where loose delay constraints are on the right and tight delay constraints are on the left). Since the leftmost delay level corresponds to an infeasible problem, no OG value is shown and the runtime then corresponds to the time required to detect that the problem is infeasible. The lower and upper whiskers of the boxplots, respectively, correspond to the 0.5% and 99.5% percentiles. Crosses identify *all* the outliers.

depending on the topology and its scaling. These variations will be discussed in Sec. 3.3.2.5. We discuss only a selection of the most representative algorithms.

The elementary LDP algorithm simply returns the LD path. As expected, the OG of LDP gets better for tighter constraints since the LD path becomes closer to the optimal solution. In terms of runtime, as LDP is compared with itself, the LDP fingerprint shows that the accuracy of our runtime metric is good (the 0.5 and 99.5 percentiles are very close to 1 and the median is approximately 1).

LARAC (Fig. 3.2b) can find the optimal solution with one LC search when the LC path is feasible. Fig. 3.2b (for the rightmost, i.e., loosest delay constraint) shows that this run is roughly two times faster than an LD search. This is due to the fact that the delay and cost values have ranges of different absolute sizes and the guess function of A* is better for the cost metric. When the problem is infeasible, LARAC notices the infeasibility with an additional LD search, hence one additional runtime ratio unit compared to the case for which the LC path is feasible. For intermediate delay levels, LARAC requires a few additional SP runs, hence leading to slightly higher runtimes. Nevertheless, these additional runs are worth it as we can observe that the OG of LARAC stays then much lower than 10 % in most cases.

While LARACGC did not complete the evaluation within a reasonable amount of time, LARACGC with $\delta = 25$ % (Fig. 3.2c) did. With a given δ, the LARACGC algorithm improves on the solution of LARAC with an *ik*SP algorithm if the OG is greater than δ. As LARAC has a OG higher than 25% only for the tightest feasible delay level (see Fig. 3.2b), LARACGC-25 only behaves differently than

LARAC for this delay level. As expected, LARACGC-25 then brings the OG to less than 25% but at a high runtime cost even for such a small gap closing: LARACGC closes a gap of at most 5% (as the OG of LARAC is at most 30%) but the algorithm can be more than 3 times slower (from a runtime ratio of less than 5 to more than 15). That confirms that algorithms based on an *ik*SP algorithm are too slow compared to their benefit for our dense queue-link network topologies.

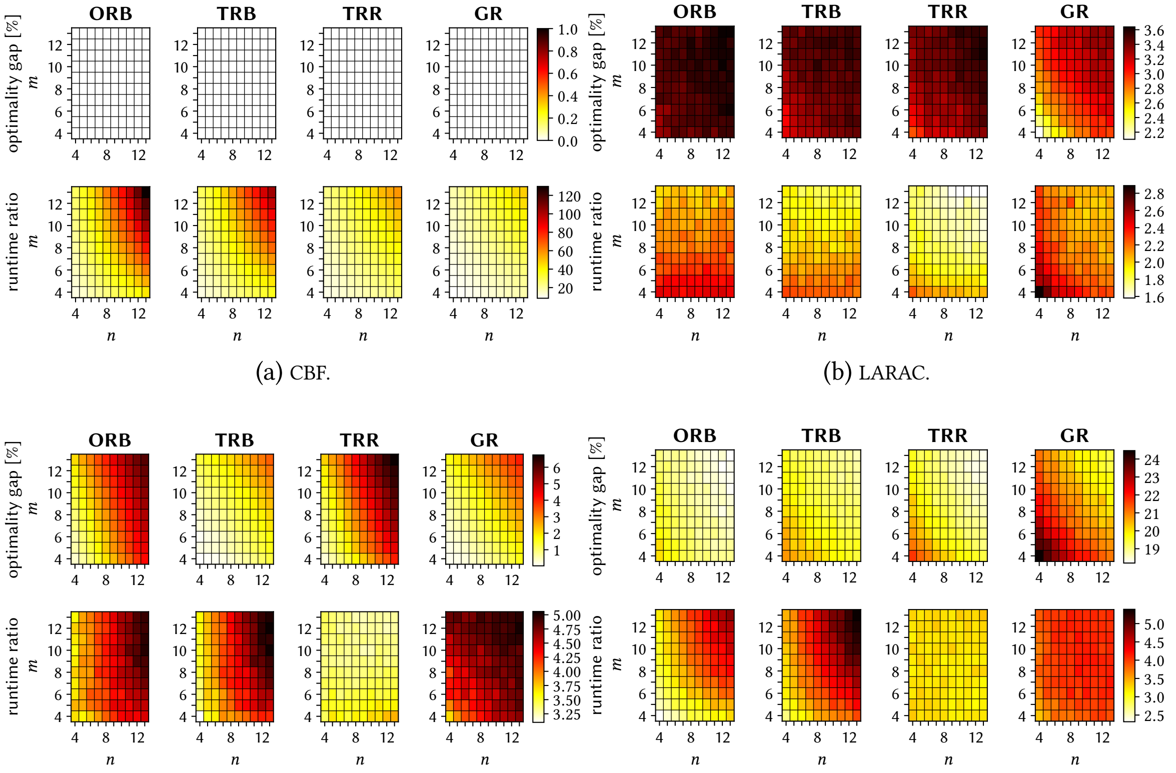
The fingerprint of H_MCOP (Fig. 3.2d) shows the runtime difference between SP and SPT searches. Indeed, for detecting an infeasible problem, H_MCOP first computes a reverse SPT. As can be seen (leftmost delay level in Fig. 3.2d), this has a much longer runtime than the single LD search of LDP. More precisely, the H_MCOP median runtime is only slightly greater, but the 99.5% percentile of the runtime is much higher than for LDP: around one order of magnitude greater. For all other cases, H_MCOP requires an additional forward SP search. The H_MCOP runtime for these delay levels is hence always slightly higher (by 0.5 since it is an LC search) than for the infeasible delay level. In terms of OG, H_MCOP interestingly presents a fingerprint of different shape than LARAC and LARAGC-25. In terms of absolute values, the OGs of H_MCOP are usually slightly worse than for the different LARAC versions, except when the delay constraint is either very loose or very tight, where H_MCOP and LARAC perform similarly.

Summary. In general, we observe that the algorithms exhibit a very different performance for each delay level and that these behaviors are also dependent on the algorithms. There is accordingly no clear best algorithm for all cases. While the LDP algorithm is the best choice if the problem is infeasible or if low runtime is critical, the LARAC algorithm is a better choice if the OG has a higher importance. If the OG is very critical and the runtime not critical, LARACGC is even a better choice (among the algorithms discussed in this section). In terms of underlying algorithms, we observed the high runtime cost of an underlying SPT run (up to 10 runtime ratio units) compared to a normal SP search (around 1 runtime ratio unit). We also observed the high runtime cost of an *ik*SP algorithm compared to the OG improvement it can bring. Finally, we confirmed the accuracy of our runtime ratio metric.

3.3.2.5 Heatmaps: Impact of Network Topology and Scale

In order to observe the behaviors of the algorithms for the different topologies and scalability levels, we collapse the fourth dimension (delay constraint tightness) of our evaluation by retaining only the average runtime ratio and OG over all delay constraint levels. This yields, for a selection of algorithms, the heatmaps shown in Fig. 3.3. While observing the scalability of the different algorithms with these heatmaps, the reader should pay attention that the scalability of the algorithm is compared to an LD search. That is, if an algorithm presents the same runtime ratio for all the scalability levels of a topology, that does not mean that its runtime is always the same but rather that the considered algorithm has a *similar scaling behavior* as an LD search. Due to the vastly different OGs and runtime ratios of the different algorithms, Fig. 3.3 has different scales for the different algorithms. An absolute comparison of the different algorithms is provided in Sec. 3.3.2.6.

The heatmaps of CBF (Fig. 3.3a) illustrate the limitation of CBF: the CBF runtime grows exponentially with the size of the network. That is consistent with the observations in [Wid94].



(a) CBF.

(b) LARAC.

(c) kH_MCOP-10.

(d) DCR.

Figure 3.3: Heatmaps showing the behaviors of selected CSP algorithms for different topologies and scalability levels. For a given algorithm, the four upper heatmaps show the OG for the four different topologies, and the four lower heatmaps show the runtime ratio. A given heatmap shows the OG or runtime ratio as a function of the scale parameters $n = 4, 5, \ldots, 13$, and $m = 4, 5, \ldots, 13$, i.e., for a total of 100 different scalability levels. Each cell corresponds to the average results of 20,000 requests (with randomly drawn delay constraints from across the seven considered delay constraint levels and corresponding subranges) simulated for this specific n and m combination. Unfortunately, because of the high variability between the algorithms, the scales are different for each algorithm.

Fig. 3.3b shows an interesting property of the LARAC algorithm: in terms of runtime, it scales better than an LD search, but only in the m scale direction; the n scale dimension affects it roughly as it affects an LD search. We observe that other selected algorithms scale worse than an LD search. In terms of OG, we see that LARAC is relatively stable over the different scalability levels. Not shown in the plots, SSR+DCCR scales similarly to the LARAC algorithm, as it is based on it.

The kH_MCOP-10 runtime (Fig. 3.3c) scales worse than an LD search in both directions (m and n) for the ORB, TRB, and GR topologies but exhibits much better scaling behavior for the TRR topology. While kH_MCOP-10 reaches low OG for small topologies, we observe that its OG grows quickly for larger topologies, especially for ORB and TRR topologies, for which its scalability is worse.

DCR (Fig. 3.3d) presents a similar scaling behavior, in terms of runtime, as H_MCOP. Compared to LARAC, this indicates the poor runtime scaling behavior of an SPT search compared to a SP search. DCR exhibits the interesting behavior that its OG improves as the topology scales up. Although not shown, this is also the case for DCUR and IAK.

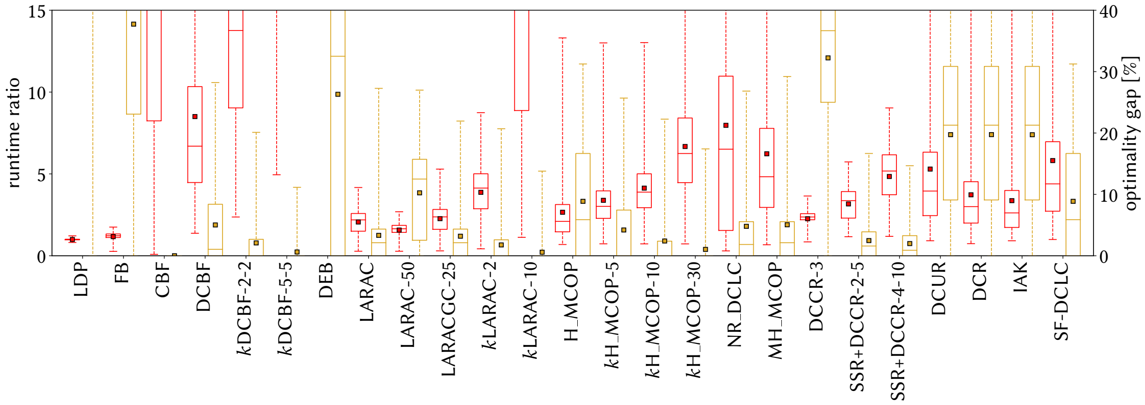


Figure 3.4: Boxplots of the runtime ratios and OGs (for all the runs of our evaluation) of the different investigated algorithms. The whiskers show the 0.5% and 99.5% percentiles, outliers are not shown.

Summary. Generally, we observe that the different algorithms have very different scaling behaviors. For example, the OG of some algorithms improves with the topology size, degrades for some and is relatively stable for others. Further, the scaling behavior of a particular algorithm (both in terms of runtime and OG) appears to heavily depend both on the topology and specific scaling direction. In terms of runtime, we observed that CBF scales very poorly and algorithms based on SPT searches scale worse than algorithms based on SP runs solely.

3.3.2.6 Average Performance Comparison

Our analysis of our four selected dimensions showed that the algorithms behave very differently depending on the specific request type and topology setup. In this section, we investigate the average performance of the different algorithms (for the algorithms that accept parameters, particular parameters values have been chosen). In order to facilitate comparisons among algorithms, we show the performance of the different algorithms on a common scale.

Fig. 3.4 shows the distribution of runtime ratio and OG values observed for the different algorithms during the complete evaluation, i.e., including all topologies, scaling sizes, and delay levels. We can again confirm the accuracy of our runtime metric: the runtime ratio of LDP is around 1 across all the evaluated dimensions. Besides the poor runtime behavior of algorithms based on i*k*SP searches already evidenced in the previous sections, we observe that algorithms based on s*k*SP/static *k* shortest paths tree (s*k*SPT) runs have a higher runtime than algorithms based on SP/SPT runs and that algorithms based on SP searches are much faster than algorithms based on SPT searches. In particular, besides CBF and DEB, the worst runtime performance belongs to *k*DCBF, *k*LARAC, and *k*MH_MCOP, which are based on s*k*SP/s*k*SPT runs. Then come, among others, NR_DCLC, MH_MCOP, DCUR, DCR, IAK, and SF-DCLC which are all based on SPT runs. Finally, the fastest algorithms are LDP, FB, and LARAC which are solely based on SP runs. The OG distributions show that high runtime does not always imply low OG. For example, DCUR, DCR, IAK, and SF-DCLC exhibit both a high runtime and a high OG.

In order to quantify the relationship between runtime ratio and OG, Fig. 3.5 shows the average runtime ratio and OG values of Fig. 3.4 in the *runtime ratio–OG* plane. It is interesting to identify the

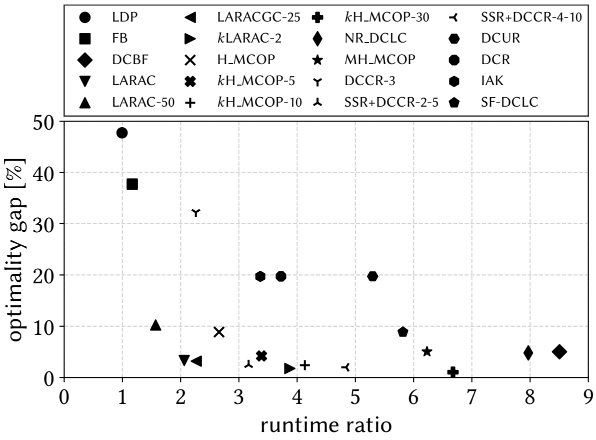


Figure 3.5: Average runtime ratio and OG of the different CSP algorithms over our entire evaluation space. Compared to Fig. 3.4, CBF, *k*DCBF-2-2, *k*DCBF-5-5, DEB, and *k*LARAC-10 are not shown because of their high runtime.

algorithms on the pareto frontier, i.e., the algorithms that, on average, are not outperformed by any other algorithm both in terms of runtime and OG. We observe that, besides CBF (not shown), LDP and FB, the algorithms making the pareto frontier are variants of the LARAC, SSR+DCCR, and H_MCOP algorithms. Considering only average behavior, these are hence the best performing algorithms, the choice among them depending on the relative importance of runtime and OG. If we look at the algorithms with the best balance between runtime and OG, i.e., the algorithm on the lower left part of the pareto frontier, we observe that they all are LARAC variants. LARAC accordingly appears to be, on average, the best choice.

Summary. On average, the best performing algorithms are LDP, FB, CBF, and the variants of LARAC, H_MCOP, and SSR+DCCR. The relative importance of runtime and optimality then dictates the best performing algorithm, on average. However, we observed that the LARAC algorithm is the one with the most potential.

3.3.2.7 Absolute Runtime Performance

While the runtime ratio facilitates the relative comparison of the algorithms in different setups, the absolute runtime is also a relevant metric for the practical deployment of the DetServ architecture. Runtime measurements performed on an Intel Core i7-3770 CPU @ 3.40 GHz showed that the runtimes of a LD search is mainly in the order of milliseconds and scales up to around 10 ms [Guc+17]. Hence, most algorithms of Fig. 3.4 have average absolute runtimes on the order of tens of milliseconds. On the other hand, CBF can reach average runtimes of up to 500 ms, which justifies the need for faster algorithms, i.e., heuristics, for online embedding of flow requests [Guc+17].

3.3.3 Which Algorithm is Best?

After analyzing the behaviors of all the algorithms, we are in a position to address the question: *which algorithm is the best?* From our observations, the answer is: *it depends.* Indeed, none of the algorithms is better than all others in terms of both runtime and OG for all topologies, scalability levels, and delay levels.

Accordingly, the selection of the best routing algorithm depends on the specific region in four dimension evaluation space where the algorithm is supposed to operate and on the relative importance of runtime and optiamlity. For example, we have seen that for small topologies and/or tight delay constraints, CBF remains a very good candidate. As the topology grows, algorithms with better scalability are needed. In terms of cost, DCR, DCUR, and IAK are the only algorithms with decreasing OG values for large topologies and these algorithms are therefore good choices for very large topologies. In terms of runtime, only the different LARAC and SSR+DCCR variations scale better than an LD search and are therefore also good candidates for large topologies. Further, While kLARAC, kH_MCOP, and all optimal algorithms are good solutions if the cost is the most important criterion, algorithms such as LDP or FB should be preferred if a very short runtime is critical.

Nevertheless, we can identify the LARAC, SSR+DCCR, and the H_MCOP variants as being among the best routing algorithms, on average. Indeed, for example, on average, for the simulated topologies and network scales, both LARAC and SSR+DCCR keep their runtime ratio lower than 4 and their OG lower than 4 %. Moreover, their behavior on the fourth dimension, i.e., for the different delay levels, is quite stable. Last but not least, the LARAC, SSR+DCCR, and H_MCOP algorithms accept several parameters that allow to tailor them to specific usage scenarios.

3.3.4 Summary

In order to select an appropriate algorithm for the routing procedure DetServ architecture, this section provided a detailed performance evaluation of CSP algorithms. We observed that the performance of the different algorithms is highly dependent on the specific scenario and that there is no one universal best CSP algorithm for all scenarios. However, we observed some general trends. First, algorithms using an *ik*SP algorithm to reach optimality or a given optimality level, have a very long runtime. Algorithms making SPT computations have much shorter runtimes, but are outperformed by algorithms that use only the results of single-source single-destination SP runs. Second, we identify the variants of LARAC, SSR+DCCR, and H_MCOP as achieving relatively good performance in most of our evaluation space.

Our analysis focused on algorithm finding single-source single-destination paths. In order to enable the routing procedure to embed multicast flows, flows with a 1+1 resilience scheme requirement, or flows that traverse intermediate VNFs, future work should investigate multicast [Ram00]; [Hos+07]; [WH00], multipath [SDJ15], and SFC routing algorithms for the CSP problem. The latter, by far the least explored, is the problem we tackle in Sec. 3.5. Before that, in Sec. 3.4, we discuss a runtime optimization strategy that can be applied to many of the algorithms presented in this section.

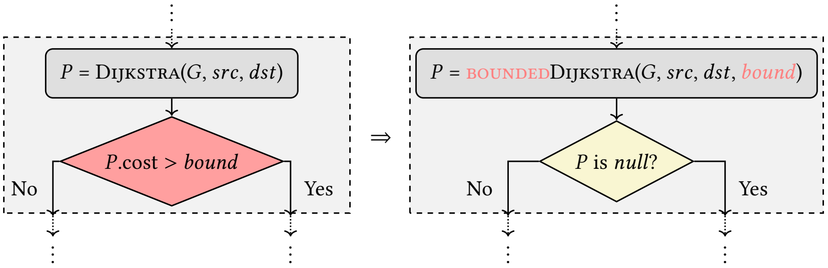

P = Dijkstra(G, src, dst)
P.cost > bound
No
Yes
⇒
P = boundedDijkstra(G, src, dst, bound)
P is null?
No
Yes

Figure 3.6: When using a SP subroutine, algorithms often do not use the paths returned by Dijkstra if they are more costly than a given *bound*. The usage of BD consists in incorporating this decision in the Dijkstra subroutine in order to avoid exploring these useless paths, thereby reducing the search space of Dijkstra. The input and output of the overlay algorithm are left unchanged, only its runtime is affected.

3.4 Search Space Reduction for Expediting Shortest Path Subroutines

We have seen in Sec. 3.3 that SP and SPT algorithms are used by many CSP and MCSP routing algorithms for solving subroutines (see Tab. 3.2). Besides algorithms for solving the CSP and MCSP, other types of more complex overlay algorithms, e.g., MCP [KK01] and constrained minimum Steiner tree (CMST) [ZPG95] algorithms, make use of SP and SPT algorithms as subroutines. For SP/SPT problems, the Dijkstra algorithm [Dij59] is commonly acknowledged as the fastest optimal algorithm for centralized implementations [CGR96]. Often, such overlay algorithms do not use the result of an SP (or SPT) subroutine if its total cost is greater than a given bound. For example, for delay-constrained problems, paths resulting from a LD SP run and the delay of which is greater than the delay constraint of the original problem are not used by the overlay algorithm to construct its solution [Guc+17]. As a result of the existence of these bounds, and because the Dijkstra SP algorithm discovers paths in increasing order of cost (see Sec. 3.2), we can terminate the SP search earlier, i.e., once it is known that paths with a greater total cost will not be considered by the overlay algorithm. This early termination allows to reduce the runtime of the SP subroutine, thereby reducing the runtime of the overlay algorithm without impacting its final result. We refer to this adaptation of Dijkstra for centralized implementations as bounded Dijkstra (BD). This constitutes the main contribution of this section. BD can be used by any routing algorithm making use of one or several SP/SPT search(es) and able to provide a bound to these subroutines (Fig. 3.6) above which results are unnecessary. On the example of CSP algorithms, we confirm the usefulness of BD by showing that it can reduce the runtime of some algorithms by 75% on average.

First, in Sec. 3.4.1, we present related work aiming at optimizing the runtime of SP/SPT searches. Second, after presenting the simple functioning and the benefits of BD in Sec. 3.4.2, we show how BD can be used, i.e., how a bound can be provided to the SP/SPT subroutines, in the particular case of centralized CSP algorithms (Sec. 3.4.3), the focus of this thesis. We show that BD can be used by a wide range of algorithms. Indeed, 20 out of the 26 CSP algorithms listed in Sec. 3.3.1 can make use of BD to improve their runtime. For each of them, we detail how bounds can be provided to the SP/SPT subroutines. Finally, in Sec. 3.4.4, we evaluate the impact of BD on the performance of these CSP algorithms. We observe that BD can reduce the runtime of some CSP algorithms by 75%

on average. For favorable cases, BD reduces the runtime of several algorithms by 96% on average. We further confirm that using BD does not change the final solution found by the algorithms. These algorithms hence have only benefits in using BD: reduced runtime at no cost. Due to the high number of algorithms, we only present the most interesting and insightful results and conclusions. The entire set of raw results and graphs is available online at [Van19c].

3.4.1 Related Work

The SP/SPT problem has been thoroughly investigated in the literature. In this section, we classify the attempts at making SP/SPT routines faster in six categories for which we list representative examples and with respect to which we highlight our contribution.

Heuristics. Some approaches improve the runtime of SP/SPT searches by accepting to find sub-optimal solutions [FSR06]. In contrast, for positive metrics, BD guarantees to find the optimal result.

Hierarchical routing. A way of reducing the complexity of SP/SPT routing is to apply hierarchical routing, thereby running SP/SPT algorithms on smaller graphs [KK77]; [JSQ02]. As it does not modify the subject graph, BD can be used as part of any such hierarchical routing scheme.

Data structure optimizations. Several studies propose optimized data structures for the implementation of Dijkstra [FT87]; [CGS99]; [Tho03]. BD is independent of the data structure used and can hence be used with any of these data structures.

Improvements of existing algorithms. Several proposals introduce extensions to the well-known Dijkstra [HNR68]; [GH05] and BF [Yen70]; [BE12] algorithms. BD falls into this category but can be used in parallel with these improvements.

Bi-directional searches. Dijkstra explores the graph from the source towards the destination(s). For SP problems, bi-directional searches, starting from both the source and the destination simultaneously, have been proposed [Kwa89]; [Ike+94]; [RT12], potentially pruning parts of the individual searches when possible [Kwa89]. While bi-directional searches can only be used for SP problems, BD can be used for both SP and SPT problems.

Preprocessing. In the context of very large graphs such as road networks, a plethora of work [Bas+16] proposes to perform a preprocessing step computing intermediate information by, e.g., clustering nodes [HS09], defining overlay graphs [HSW09], defining important transit nodes [AIY13], or computing virtual links [Abr+12]. This preprocessed information is later used to solve routing requests faster. The precomputation step being costly, these approaches are only suitable for solving a batch of requests on the same static topology. Besides, these algorithms work well for large topologies but hardly outperform Dijkstra for general topologies [Bas+16]. On the other hand, BD can provide significant benefit on general and dynamic topologies. Some preprocessing algorithms involve SPT subroutines [AIY13] and expedite these subroutines by pruning parts of the network because the corresponding information was already obtained from previous SPT runs. In contrast, BD does not exploit previous searches in order to improve runtime but rather information provided by an overlay algorithm. Further, these proposals simply prune parts of the network when a given condition is met, while BD completely terminates.

We note that bounded Dijkstra runs were already used in the literature [Bos+08]; [LL09]; [BM12]; [RT13]; [Var14]; [VJ14]; [SPD14], but only as part of the design of new specific algorithms. Our contribution consists in the formalization and generalization of such an approach for any generic algorithm using SP/SPT subroutines, and in the quantification of its benefits for these algorithms.

3.4.2 Proposed Solution: Bounded Dijkstra (BD)

In this section, we present the bounded Dijkstra (BD) algorithm. After presenting the context in which BD can be used (Sec. 3.4.2.1), we describe the simple idea of the algorithm (Sec. 3.4.2.2) and detail the impact it can have on SP (Sec. 3.4.2.3) and SPT (Sec. 3.4.2.4) searches. Then, we show that the same idea can also be applied to the BF and *Chong* algorithms (Sec. 3.4.2.5 and 3.4.2.6) presented in Sec. 3.2.

3.4.2.1 Context: Centralized Bounded Shortest Path (Tree) Subroutines

The SP and SPT problems are core networking problems. Besides in their direct applications, these problems are often encountered as subproblems of other more complex centralized problem settings. For example, as observed in Sec. 3.3, many CSP and MCSP algorithms use results of SP/SPT searches to determine a solution to their problem. Similarly, MCP algorithms such as H_MCP [KK01] or CMST algorithms such as BSMA [ZPG95] make use of an underlying SPT algorithm to construct a solution to their problem. When an SP/SPT algorithm is used as such a subroutine of a centralized algorithm, it often happens that paths with a total cost greater than a given bound are not used. For an SP search, this means that the result itself is not used. For an SPT search, this means that the paths to some destinations (too costly) are not considered, while others are. For example, for delay-constrained problems, paths resulting from a LD SP/SPT run which have a delay higher than the delay constraint are not considered. Similarly, for DCLC, or CSP, problems, paths resulting from a LC SPT run which have a cost higher than the cost of the LD path will not be used, as these paths have a higher delay *and* cost than the LD path.

3.4.2.2 Idea: Early Termination for Search Space Reduction

As a result of the existence of these bounds, and because the Dijkstra algorithm discovers paths in increasing order of cost (see Sec. 3.2), the SP/SPT searches can be stopped earlier, i.e., once it is known that paths with a greater total cost will not be considered by the overlay algorithm, thereby reducing the search space of Dijsktra and hence the runtime of the overlay algorithm. We refer to such a modified version of Dijkstra as BD. The pseudo-code of BD is shown in Fig. 3.7 and the required modification in the overlay algorithm is shown in Fig. 3.6. For positive metrics, BD does not influence the result of the overlay algorithm, as the latter considers having no path and having a too costly path identically. In order to use BD, an algorithm must of course be able to provide a bound value above which results are unnecessary. We detail in Sec. 3.4.3, on the example of CSP algorithms, how overlay algorithms can provide such bounds.

```
function boundedDijkstra(G, src, dst, bound)
    Create empty priority queue Q
    for each node ∈ G do
        node.cost ← +∞
    src.cost ← 0
    Q.add(src)
    while not Q.empty do
        node ← Q.popLeastCostNode()
        if node is dst then return getPredecessors(dst)
        if node.visited then continue
        node.visited ← true
        for each outgoing edge of node as edge do
            newCost ← node.cost + edge.cost
            if newCost < bound then
                if newCost < edge.dst.cost then
                    edge.dst.cost ← newCost
                    edge.dst.predecessor ← node
                    Q.add(edge.dst)
    return null
```

Figure 3.7: Pseudo-code of the Dijkstra algorithm and the BD adaptation (shown in light red). Note that, depending on the data structure in use for the priority queue (Q), the pseudo-code may vary slightly. We show here the most common pseudo-code using a heap, which we used for our implementation and which performs best among the available data structures [CGR96].

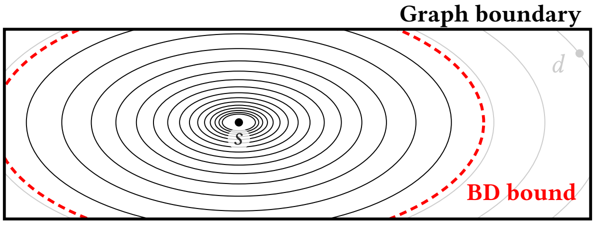


Figure 3.8: Dijkstra discovers paths in increasing order of cost (as illustrated by the ellipses representing nodes which are equidistant from the source node) and stops once the destination (d) is reached. BD (dashed ellipse) terminates the search at a given cost from the source node, once it is known that longer paths will not be considered by the overlay algorithm.

3.4.2.3 Impact on an SP Search

For an SP search, the Dijkstra algorithm discovers paths in increasing order of cost from the source node and stops once the destination is reached. The pseudo-code is shown in Fig. 3.7. When a bound is provided to Dijkstra (or BD), two different cases can happen.

BD: the destination is further than the bound. First, the provided bound can be lower than the cost of the SP to the destination. In this case, BD avoids exploring unnecessary parts of the network (Fig. 3.8) by preventing Dijkstra from considering paths with a cost greater than the provided bound and hence terminating before reaching the destination, i.e., before Dijkstra would have terminated. The path returned is then “NULL”, which is considered by the overlay algorithm in the same way as a path which is too costly: it does not use it.

BD: the destination is closer than the bound. Second, the provided bound can be greater than the cost of the SP to the destination. In this case, BD might appear useless, as it will, like Dijkstra, terminate when reaching the destination. However, BD can also have a benefit in this case. Indeed, BD can avoid putting an element in the queue whose associated cost is greater than the provided bound (line 14 in Fig. 3.7). Because such elements can be reached before the destination, this allows BD to avoid unnecessary operations (lines 15–18) and to reduce the size of its priority queue, thereby increasing the speed of the upcoming popping operations (line 8). Let us consider an example where the cost of the SP to the destination is 16 and the BD bound is 18. BD pops an element with an associated cost value of 15 out of its queue (line 8) and expands it. This expansion leads to elements with associated costs 20, 22 and 24. While the traditional Dijkstra would execute lines 15–18, BD knows that these elements will never be used. Hence, BD can directly discard these elements (line 14), thereby preventing executing lines 15–18. As a result, even when the bound provided to BD is greater than the cost of the SP to the destination, BD can be beneficial. We will confirm this in our evaluations (Sec. 3.4.4.2). However, the benefit is expected to decrease as the bound gets greater. Indeed, the number of elements that can be discarded will decrease. In the example above, a higher BD bound of 23 would for example allow to discard only one of the elements, rather than the three of them with a BD bound 18. Note that this phenomenon also happens when the destination is further than the provided bound.

3.4.2.4 Impact on an SPT Search

For an SPT search, Dijkstra behaves as in the SP case (Fig. 3.7) but instead of stopping when reaching a given node (line 9 of Fig. 3.7), the algorithm stops when its priority queue is empty, i.e., when it has reached all the nodes.

When provided with a bound, BD can potentially stop the SPT expansion before exploring the whole graph. That is, BD can avoid waiting for reaching some nodes which are too far away. As for the single-destination case, this allows to reduce the runtime by preventing the exploration of unnecessary parts of the network and by avoiding unnecessary operations and the addition of unnecessary elements to the priority queue (see Sec. 3.4.2.3).

3.4.2.5 BD Idea for Bellman-Ford

The BF algorithm (presented in Sec. 3.2) is another algorithm for solving SP/SPT problems. Because of its structure, the algorithm is more often used in distributed implementations. However, it is also used as a subroutine of other complex algorithms where Dijkstra cannot be used (e.g., DEB, see Sec. 3.4.3.4). While the structure of the BF algorithm is very different from the structure of Dijkstra, it also discovers paths in increasing order of cost. Hence, the BD idea can also be applied to BF by simply discarding paths more costly than the given bound. As a result, algorithms making use of the BF algorithm as a subroutine can also apply the BD principle.

3.4.2.6 BD Idea for Chong's Algorithm

The problem of finding the kSP between two nodes (or the k shortest paths tree (kSPT) from one node to several destinations) also arises often as a subroutine of more complex algorithms (e.g., kDCBF and *k*H_MCOP, see Sec. 3.4.3.3). *Chong's algorithm* (presented in Sec. 3.2) solves this problem by assuming that the *k* value is known a priori. The algorithm is identical to the Dijkstra algorithm but keeps track, at each node, instead of one single path, of the current *k* best paths found. Hence, the BD idea can be applied to Chong's algorithm in the same way as it is applied to Dijkstra. As a result, algorithms making use of Chong's algorithm as a subroutine can also apply the BD principle.

3.4.3 Application: BD for CSP Routing

In this section, we show that BD can be used by a wide range of algorithms by showing *(i)* how the CSP algorithms listed in Sec. 3.3.1 can replace their SP and SPT subroutines with BD, and *(ii)* how bounds can be provided to these BD runs.

Tab. 3.3 shows, for each algorithm of Tab. 3.2, how many times it can replace a SP/SPT run with a BD run. The cases for which BD can be used are separated based on the metric (cost, delay or a combination) and on the algorithm on which the BD principle is applied (SP/SPT refers to Dijkstra, *k*SPT to Chong and BF to BF for SP). Out of the 26 algorithms, only 6 cannot make use of BD. A detailed description of the algorithms can be found in [Guc+17] or in the original publications referenced in the table.

3.4.3.1 Algorithms that Cannot Use BD

First, CBF, A*Prune, and SMS-PBO have a specific structure making use of no underlying SP/*k*SP/SPT/*k*SPT algorithm and can hence not make use of BD. Second, *k*SPMC, E_MCOP, and *k*LARAC exclusively make use of *k*SP and SP algorithms to which no bound can be provided. They can hence also not make use of BD.

3.4.3.2 Algorithms that Can Use BD for SP Only

The LDP, FB, LARAC, LARACGC, SCRC, DCCR, and SSR+DCCR algorithms run a LD SP procedure (i.e., optimizing the delay metric) which can make use of BD by using the bound of the original problem. After this LD SP run, the LARAC, LARACGC, SCRC, DCCR, and SSR+DCCR algorithms run one or several LC SP runs (i.e., optimizing the cost metric). These runs could be provided with the cost of the LD path as bound. However, if provided with this bound, this BD run will always be in the case where the provided bound is greater than the cost of the SP to the destination (see Sec. 3.4.2.3). As we will see in Sec. 3.4.4.2, on average, the usage of BD in such a case increases the runtime of the SP run. As a result, we do not consider the LC run as a BD run. The LARACGC, SCRC, DCCR, and SSR+DCCR further execute a *k*SP run to which no bound can be provided.

3.4.3.3 Algorithms that Can Use BD for SPT

The DCUR, SF-DCLC, SMS-CDP, SMS-RDM, IAK, DCR, H_MCOP, *k*H_MCOP, DCBF, and *k*DCBF algorithms run a LD search to which the delay bound of the CSP problem can be provided as a bound. While DCR runs a LD SP search and *k*DCBF a LD *k*SPT (Chong) search, all the others run a LD SPT search. DCUR, SF-DCLC, SMS-CDP and DCR then possibly execute a LC SPT run to which the cost of the LD path from the source to the destination can be provided as a bound. Indeed, any

Number of BD usages

Algorithm	Delay				Cost	Comb.
	BF	SP	SPT	*k*SPT	SPT	SPT
Algorithms that cannot use BD (Sec. 3.4.3.1)						
CBF [Wid94]						
A*Prune [LR01]						
kSPMC [Guc+17]						
E_MCOP [Fen+02b]						
SMS-PBO [SMM98]						
kLARAC [JV01]						
Algorithms that can use BD for SP only (Sec. 3.4.3.2)						
LDP [Guc+17]		1				
FB [LHH95]		(0, 1)				
LARAC [AN78]; [HZ80]; [BG96]; [Jüt+01]		1				
LARACGC [HZ80]		1				
SCRC [SCC07]		1				
DCCR [GM03]		1				
SSR+DCCR [GM03]		1				
Algorithms that can use BD for SPT (Sec. 3.4.3.3)						
DCUR [SRV97]; [RS00]			1		(0, 1)	
SF-DCLC [LLF05]			1		(0, 1)	
SMS-CDP [SMM98]			1		(0, 1)	
SMS-RDM [SMM98]			1			
IAK [IAK98]			1			
DCR [SL98]		1			(0, 1)	
H_MCOP [KK01]			1			
kH_MCOP [KK01]			1			
NR_DCLC [Fen+02a]		(0, 1)				≥ 0
MH_MCOP [Fen+02b]			1			≥ 0
DCBF [JV01]			1			
*k*DCBF [JV01]				1		
Algorithm that can use BD for BF (Sec. 3.4.3.4)						
DEB [CA03]	1					

Table 3.3: Number of times the CSP algorithms of Tab. 3.2 can make use of BD based on the metric (cost, delay or a combination) and algorithm on which BD can be applied (SP/SPT refers to Dijkstra, *k*SPT to Chong and BF to BF for SP). When the number of times BD can be used depends on the routing request, the set of possible values is given between parentheses and unbounded values are given using the ≥ symbol. Underlined algorithms are optimal.

path with a cost higher than the LD path will never be used by the algorithms, as they would then rather choose to follow the LD path, which has both a lower cost and delay. The IAK, H_MCOP, *k*H_MCOP, DCBF and *k*DCBF algorithms further execute a LC *k*SP/SP search. As for the algorithms in Sec. 3.4.3.2, a bound could be provided to this LC run but, for the same reason, we do not consider it.

The NR_DCLC algorithm starts like FB and can hence make use of BD in the same way. Then, if the problem is feasible, it runs several times H_MCP [KK01]; [Fen+02b] (an MCP algorithm), a modified version of H_MCOP, to improve on the LD path result. H_MCP uses a metric combining the cost and delay metrics for its SPT search. Since bounds on both the delay (the bound of the CSP problem) and on the cost (the cost of the best path found so far) are known, the first step of H_MCP can also make use of BD. Hence, NR_DCLC can further make use of BD by using H_MCP with BD.

The MH_MCOP algorithm is similar to NR_DCLC but, instead of using H_MCP to improve on the LD path result, H_MCP is used to improve on the path found by H_MCOP. Hence, MH_MCOP can make use of BD by using both H_MCOP and H_MCP with BD.

3.4.3.4 Algorithm that Can Use BD for BF

The DEB algorithm runs a LC and a LD SP search using BF. As for LARAC, a bound can be provided to both the LD and LC searches but we only consider the LD search as a BD run.

3.4.4 Evaluation

The goal of our evaluation is twofold. First, in order to confirm our expectations of Sec. 3.4.2.3 and 3.4.2.4, we quantify the impact of BD on an SP and an SPT run. To do so, we observe the behavior of the LDP (Sec. 3.4.4.2) and IAK (Sec. 3.4.4.3) algorithms, which are using BD respectively for a single SP and a single SPT run based on the delay metric. Second, in order to confirm the applicability of BD, in Sec. 3.4.4.4, we observe its impact on the performance of the CSP algorithms presented in Sec. 3.4.3. Because of the big amount of resulting data, we only present here the most insightful and representative results. The complete data and set of graphs is available online at [Van19c].

Among all the runs performed during the evaluation, the paths returned by the algorithms with and without BD were always identical, thereby confirming that BD does not impact the output of the algorithms. Hence, in the following, we only discuss the runtime of the algorithms.

The algorithms have been implemented using Java 8 and evaluated on an Ubuntu 16.04 PC equipped with an Intel Core i7-4790 CPU @ 3.60GHz.

3.4.4.1 Setup

In this section, we define the three dimensions along which we run our evaluation and describe how our plots and routing requests are generated.

First dimension: distance between nodes. From Fig. 3.8, we can expect that, if the source and destination nodes are far apart from each other, the impact of BD will be lower. Indeed, in most directions, the graph boundary itself will be expected to stop the expansion of Dijkstra before BD does it. If the source and destination nodes are closer to each other compared to the graph size (alternatively, if the graph boundary rectangle in Fig. 3.8 gets bigger), we can expect that the BD bound will be reached more often before the boundary of the network and hence BD will provide more benefit. That is, the impact of BD potentially depends on the relative distance (in terms of cost) between the source and destination nodes compared to the size of the topology. To ease the definition of this dimension, we only consider a grid topology of size $N \times N$. For a given grid size, we define 10 different so-called *distance buckets*. These buckets correspond to source and destination nodes pairs whose least-hop distance in the grid is between 0 and 10%, 10% and 20%, ..., 90% and 100% of the longest path in the grid (i.e., of $2 \times (N - 1)$).

Second dimension: tightness of the constraint. We also observed in Sec. 3.4.2 that the impact of BD potentially differs based on the tightness of the constraint of the overlay problem (CSP problem in our case). The delay constraint can range from a loose value for which the LC path is feasible to tight values for which the problem is infeasible. Within this range, we define 7 ranges of equal sizes. We refer to these ranges as *delay levels*.

Third dimension: grid size. We further consider the grid size N as an evaluation dimension. We vary N from 6 to 20. We observed that the grid size does not influence the impact of BD. Hence, we here ignore this dimension (we always aggregate all the results for all the sizes) but the corresponding graphs are available online [Van19c].

Plots generation. For each algorithm, we generate plots showing the distribution[4] of the *runtime ratios* observed for the different values of a given dimension (i.e., node distance, delay level or grid size). For a given request, the runtime ratio is defined as the runtime of the algorithm without BD divided by the runtime of the algorithm with BD. Hence, values greater than 1 corresponds to a situation in which BD reduces the runtime of the algorithm. The hidden dimensions are either aggregated (i.e., the runtime ratios for all their values are incorporated in the distributions) or only a specific value of these dimensions is incorporated in the distributions.

Requests generation. For each algorithm and for each combination of distance bucket, delay level and grid size, we generate random cost and delay values between 1 and 2 for each link and we randomly generate 5000 requests (within the corresponding distance bucket and delay level). The 5000 requests are then solved by the considered algorithm and its corresponding version with BD. The first 500 runs are used as warm-up for the Java HotSpot optimizer and their results are not considered. The order in which the algorithm and its BD version are run is alternating. This prevents the Java HotSpot optimizer from optimizing one of the run over the other. The distance bucket dimension cannot be aggregated simply by considering all the runtime ratios for all the different buckets. Indeed, considering random source-destination pairs, small distances are more probable than long distances. Hence, for plots aggregating the distance bucket dimension, for each algorithm

[4] After removing the values below the 1% percentile and above the 99% percentile, the distributions are shown as boxplots showing the 10%, 25%, 50%, 75% and 90% percentiles. A red square identifies the average. Versions of the plots also showing the outliers are available on the accompanying web interface [Van19c].

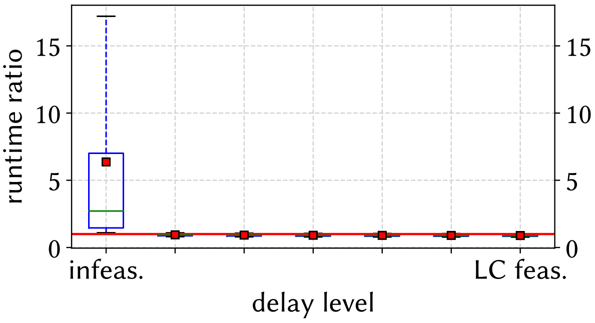


Figure 3.9: Runtime ratios of LDP for different delay levels. On average, for single-destination SP searches, BD is only useful when the provided bound is lower than the cost of the SP to the destination.

and combination of delay level and grid size, we generate 50,000 (out of which 5000 are used as warm-up) requests by randomly selecting a source and a destination node[5].

3.4.4.2 LDP: Influence of BD on an SP Search

We first observe the impact of BD on the runtime of LDP. This allows us to gain insight into the behavior of BD for SP runs (see Sec. 3.4.2.3).

Fig. 3.9 shows the impact of BD on the runtime of LDP for the different delay levels, all the other dimensions being aggregated. As expected, we observe that BD allows to dramatically reduce (more than 6 times faster on average) the runtime of an SP search when the bound is lower than the cost of the SP to the destination (*infeasible* delay level). For all the other cases (the delay bound is greater than the cost of the SP to the destination – see Sec. 3.4.2.3), we however observe that, *on average*, the additional runtime induced by BD for checking if the bound is violated (line 14 in Fig. 3.7) is not compensated by its benefit. Indeed, we observe that the runtime ratios are, on average, slightly lower than 1.

However, interestingly, even if the provided bound is greater than the cost of the SP to the destination, there are cases for which BD reduces runtime (this is due to the fact that BD then avoids to place unnecessary elements in the priority queue – see. Sec. 3.4.2.3). Fig. 3.10 shows the impact of BD on the runtime of LDP for the different distance buckets, the grid size dimension being aggregated and for the first feasible delay level. The figure confirms that BD can also improve the runtime of an SP search when the provided bound is greater than the SP to the destination (see Sec. 3.4.2.3) but that the benefit of BD only balances its additional overhead for low distances and tight bounds. Fig. 3.9 however shows that, *on average*, BD is only beneficial when the provided bound is lower than the cost of the SP to the destination.

Fig. 3.10 also shows that, when the provided bound is greater than the SP to the destination, the impact of BD decreases as the distance between the nodes compared to the topology size increases. When the provided bound is lower than the SP to the destination, this effect is compensated by the fact that, when the distance is low, Dijkstra will anyway terminate before BD can stop it, thereby preventing BD from significantly reducing the search space. Hence, in this case, the impact of BD

[5] 10 times more than for each distance bucket since there are 10 different distance buckets.

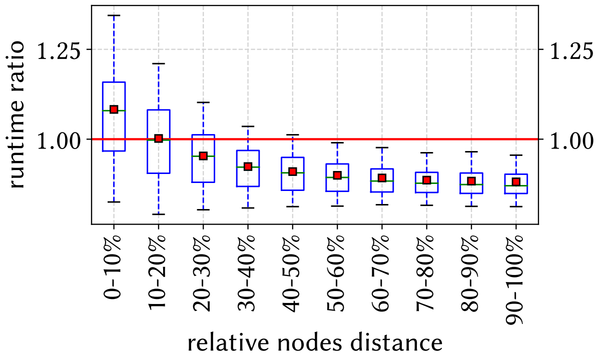


Figure 3.10: Runtime ratios of LDP for the different distance buckets and for the first delay level that is feasible. For SP searches, in some favorable cases (short distances), BD can still be beneficial even if the provided bound is greater than the cost of the SP to the destination.

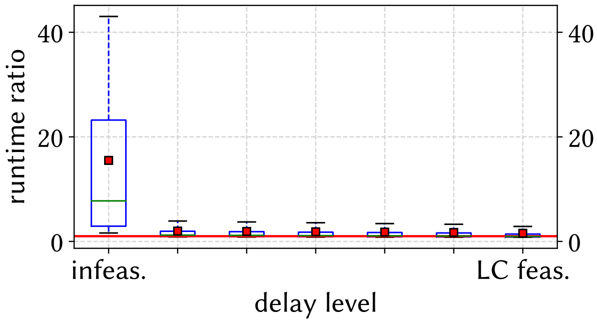


Figure 3.11: Runtime ratios of IAK for different delay levels. For SPT searches, BD is beneficial in any case but better when the provided bound is lower.

is relatively stable along the different distance buckets. This can be seen on the additional graphs available online [Van19c].

3.4.4.3 IAK: Influence of BD on an SPT Search

We observe the impact of BD on the runtime of IAK. As IAK simply runs a LC SP search (without BD) and a LD SPT search with BD, this allows us to gain insight into the behavior of BD for SPT runs.

Fig. 3.11 shows the impact of BD on the runtime of IAK for the different delay levels, the other dimensions being aggregated. In comparison to Fig. 3.9, this shows that BD has a much higher impact for SPT runs (up to 15 times faster on average for infeasible cases, against 6 times faster for SP searches). Further, we observe that, even when the problem is feasible, BD still provides benefit. This is because, even if the destination is closer than the provided bound, other nodes further away may be neglected by BD. Hence, this shows that, on average, BD is useful in any case for SPT searches. As expected, we observe that the impact of BD decreases as the delay bound gets looser.

Fig. 3.12 shows the impact of BD on the runtime of IAK for the different distance buckets, the other dimensions being aggregated. We observe that BD has less impact when the source and destination nodes of the CSP problem are further away from each other. This was expected. Indeed, as nodes get further away from each other, the SPT search is more often blocked by the graph boundary

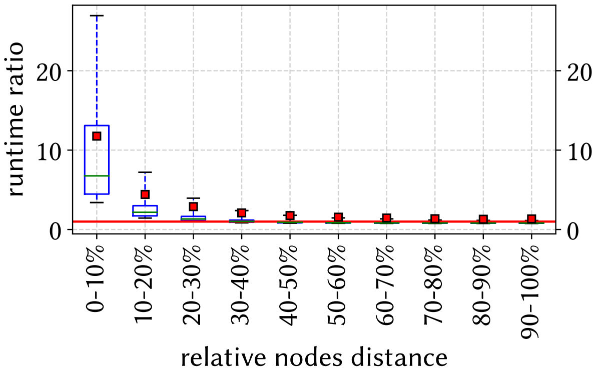


Figure 3.12: Runtime ratios of IAK for the different distance buckets. This shows that, for SPT searches, BD is beneficial in any case but better when the source and destination of the CSP problem are close to each other.

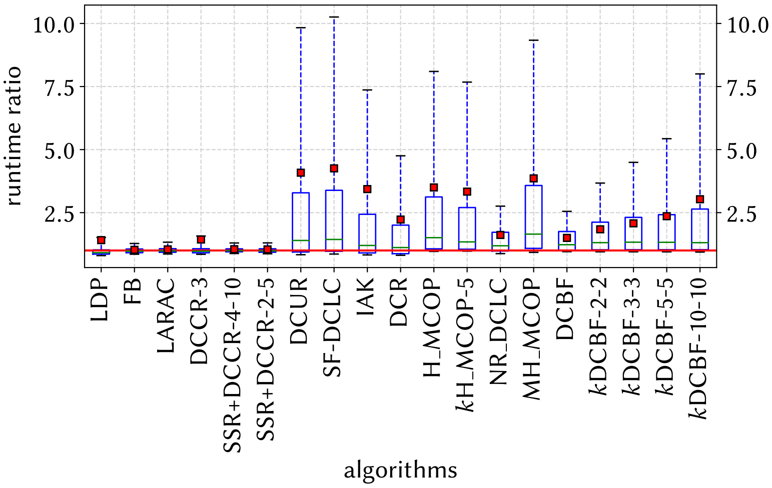


Figure 3.13: Distributions of the runtime ratios of the different algorithms, all the dimensions being aggregated. We observe that BD can greatly reduce the runtime of some algorithms (up to 4 times faster, i.e., runtime reduced by up to 75% on average for some algorithms).

itself rather than by the BD bound. As Fig. 3.11, Fig. 3.12 also confirms that BD is more efficient for SPT runs than for single-destination SP runs and that it is, on average, beneficial.

3.4.4.4 BD Impact on All CSP Algorithms

In this section, we observe the impact of BD on all the CSP algorithms presented in Sec. 3.4.3. Algorithms requiring parameters have been configured as in [Guc+17]. In the plots, the parameters values are appended to the algorithm names.

Because too slow, SCRC, LARACGC, SMS-CDP, SMS-RDM, SMS-PBO and DEB were not able to run the evaluation in a reasonable amount of time. However, because of their similarity with LARAC, the impact of BD on LARACGC and SCRC is supposed to be similar to the impact on LARAC.

Fig. 3.13 shows the runtime ratios of all the algorithms, all the dimensions being aggregated. As can be seen, BD is, on average, beneficial for all the algorithms. However, we can see that algorithms which can only use BD for SP runs (LDP, FB, LARAC, DCCR and SSR+DCCR – Sec. 3.4.3.2) are only slightly improved. This was expected based on our observations of Sec. 3.4.4.2. Indeed, the average

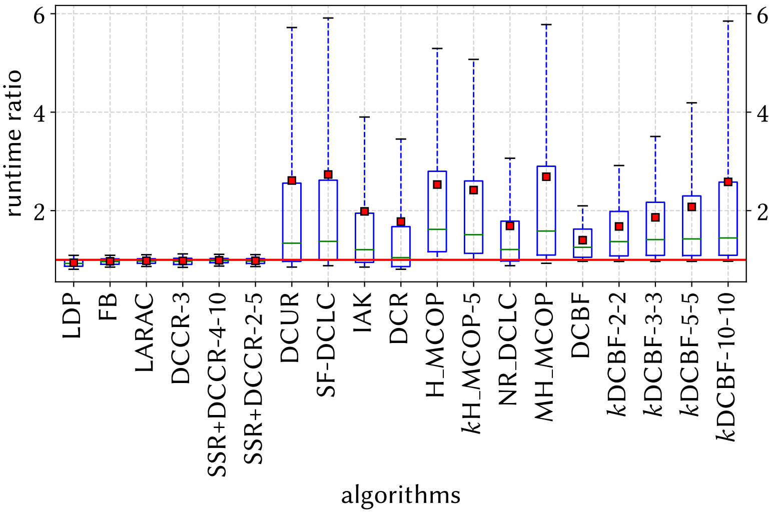


Figure 3.14: Distributions of the runtime ratios of the different algorithms for the first feasible delay level, the other dimensions being aggregated. We observe that, even outside of the infeasible case, BD can provide significant benefit to some algorithms (around 3 times faster, i.e., runtime reduced by up to 66% on average for some algorithms).

impact of BD on SP runs is marginal. On the other hand, we can see that the runtime of algorithms which can use BD for SPT runs (DCUR, SF-DCLC, IAK, DCR, H_MCOP, kH_MCOP, NR_DCLC, MH_MCOP, DCBF and *k*DCBF – Sec. 3.4.3.3) can be dramatically improved by BD. For example, on average, DCUR, SF-DCLC and MH_MCOP are 4 times faster with BD, i.e., their runtime is reduced by 75% with BD. Also, most algorithms see their runtime improved by at least 20% in 50% of the cases. DCUR, SF-DCLC, IAK and DCR present an interesting behavior. The two algorithms benefiting the most from BD are DCUR and SF-DCLC, because both their LC and LD SPT runs can use BD. Then, IAK benefits less because its LC SP run cannot benefit from BD. Finally, DCR benefits less than IAK even though its LD SP run and its LC SPT run can both benefit from BD. This shows that, while SPT runs benefit more from BD than SP runs, LD SPT runs benefit more from BD than LC SPT runs. MH_MCOP further shows that using BD for SPT runs based on a combination of the cost and delay metrics can provide as much benefit as for SPT runs based on the delay metric solely.

In order to highlight that BD can also provide significant benefit for feasible delay bounds, Fig. 3.14 shows the distributions of the runtime ratios of the different algorithms for the first feasible delay level, the other dimensions being aggregated. We can see that BD can still drastically reduce the runtime of some algorithms when the delay bound is feasible. For example, DCUR, SF-DCLC, H_MCOP, MH_MCOP and *k*DCBF-10-10 are, in this case, on average around 3 times faster with BD, i.e., their runtime is reduced by 66%. In further evaluations available online [Van19c], by aggregating all dimensions except the distance between the source and destination nodes, we have observed that, when the problem is not infeasible, BD is beneficial to the algorithms as long as the relative distance stays below 40-50% of the topology size. When the nodes are further apart from each other, the graph boundary stops the expansion of Dijkstra before BD can have any significant impact.

We have seen that BD has potentially more impact when the delay constraint is tighter and the distance between the source and destination nodes is smaller. Fig. 3.15 shows the runtime ratios

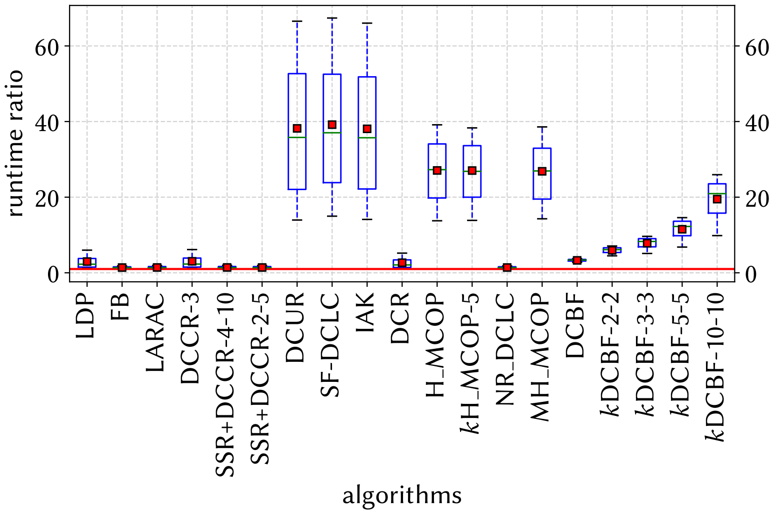


Figure 3.15: Distributions of the runtime ratios of the different algorithms for favorable cases (infeasible delay constraint and the 0-10% distance bucket). We observe that, in these favorable cases, BD can drastically reduce the runtime of some algorithms (more than 25 times faster, i.e., runtime reduced by at least 96% on average for some algorithms).

of all the algorithms for the infeasible delay level and for the 0-10% distance bucket, the grid size dimension being aggregated. We observe that, in this favorable case, BD allows to drastically reduce the runtime of all the algorithms, including those only using BD for SP runs. For example, SF-DCLC, IAK, DCUR, H_MCOP, kH_MCOP and MH_MCOP are more than 25 times faster with BD, i.e., they see their runtime reduced by more than 96% on average. Interestingly, DCR and NR_DCLC, which have a good average runtime improvement (see Fig. 3.13), do not benefit much in this favorable case. This is because, in the infeasible case, both algorithms only run SP searches, thereby having a benefit similar to the algorithms only using BD for SP searches (e.g., LDP and LARAC).

3.4.5 Summary

SP and SPT algorithms are often used as subroutine of overlay algorithms solving more complex problems, e.g., the CSP problem encountered by the routing procedure of the DetServ architecture. In such a situation, it often happens that the result of an SP subroutine is not used if its total cost is greater than a given bound. Because Dijkstra discovers path in increasing order of cost, we can terminate the execution of Dijkstra as soon as it reaches paths which have a cost greater than the known bound. We refer to this adaptation of Dijkstra as BD. By terminating Dijkstra earlier, its search space is reduced, thereby reducing its runtime and hence the runtime of the overlay algorithm using it. BD can be used by any routing algorithm making use of an underlying SP/SPT algorithm and that can provide a bound to this algorithm. We evaluated the impact of BD on the specific example of CSP algorithms. We have shown that BD does not impact the output of the algorithms but can dramatically decrease their runtime. While BD can be beneficial for both SP and SPT searches, we showed that its benefit is greater for SPT runs. The runtime of some algorithms is reduced by 75% on average. We further showed that BD is more efficient for tight delay constraints and when the source and destination nodes of the CSP problem are close to each other compared to the size of the

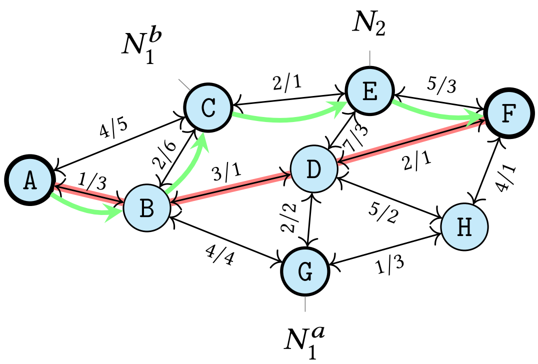


Figure 3.16: Illustration of the ordered specified nodes extension (OSNE) of the unicast CSP problem. The solution of the CSP problem is shown in red. Edges are labeled with their *cost/constraint* values. The solution to the OSNE of the CSP problem that has to visit either N_1^a or N_1^b and then N_2 is shown in green.

topology. For these favorable cases, several algorithms see their runtime reduced by 96% on average (i.e., BD allows them to be more than 25 times faster).

3.5 Enabling Routing through Service Function Chains

So far, we focused on the problem of routing flows from a single source node to a single destination node. However, with the emergence of network function virtualization (NFV), the problem of routing through intermediate specified nodes has become an important issue. The NFV paradigm consists in virtualizing network functions in order to, e.g., deploy and migrate them at runtime where necessary [Mij+15]. In such a scenario, applications can define a set of VNFs that have to be traversed (e.g., security functions). This is referred to as service function chain (SFC). Besides, as part of NFV, the function placement problem (FPP) [Bas+14] consists in finding the optimal locations for hosting the VNFs. The routing of a request through a SFC corresponds to the problem of routing through specified nodes. The FPP corresponds to having several candidate nodes as potential locations for each VNF. At the same time, the latency requirements of the applications still have to be met. Accordingly, routing problems can be extended by requiring the solution to traverse a given set of n so-called *specified nodes* (denoted by $\{N_1, \ldots, N_n\}$). We define such an extension as a specified nodes extension (SNE) or as an ordered specified nodes extension (OSNE) if the set of specified nodes has to be visited in a specific order. In this section, we focus on OSNE. For a given specified node N_i, a set of up to c so-called *candidate nodes* (denoted by $\{N_i^a, N_i^b, \ldots\}$) can be given, one of them at least having to be visited. If only one candidate is defined for a given specified node N_i, we simply refer to it as N_i. An example of this problem is depicted in Fig. 3.16. Let us consider a CSP problem from A to F that has to visit two ordered specified nodes N_1 and N_2 with a maximum constraint value of 15. N_1 has two candidates G and C and N_2 only has one candidate E. For example, in Fig. 3.16, if N_1^a and N_1^b correspond to two nodes at which a firewall VNF can be deployed, and N_2 to a node hosting a deep packet inspection (DPI) VNF, the illustrated request corresponds to routing a flow through a SFC consisting of a firewall and a DPI and placing the firewall VNF at one out of two possible locations (i.e., solving the FPP). In Fig. 3.16, each edge is labeled with its cost and constraint metric values separated by a slash. Without considering the specified nodes, the optimal CSP (highlighted in red) is A-B-D-F with a cost of 6. This path can be found by traditional optimal CSP algorithms, e.g., CBF (see Tab. 3.2). Considering the specified nodes, the optimal CSP (shown in green) is A-B-C-E-F with a cost of 10. Traditional CSP algorithms as those listed in Tab. 3.2 are not able to solve this problem and alternative solutions are hence needed. Indeed, the routing procedure of the DetServ architecture has to be able to efficiently solve this problem in order to achieve the online provisioning of routing requests that have to traverse a series of VNFs.

In this section, we propose two algorithms for solving this problem. First, we propose Lagrange relaxation based aggregated cost for specified nodes (LARAC-SN), a fast and close to optimal algorithm (in our evaluations, its OG remains lower than 1.62% on average) for solving the OSNE of the unicast CSP problem based on the SoA LARAC algorithm [AN78]; [HZ80]; [BG96]; [Jüt+01]. However, LARAC-SN can only handle one candidate per specified node ($c = 1$) and is relevant only for unicast CSP problems. Second, for problems different than the unicast CSP problem and/or for problems considering several candidates per specified node ($c > 1$), we propose mole in the hole (MITH), a graph transformation algorithm that allows any routing algorithm to be able to solve the OSNE of the routing problem it is originally solving. The power of MITH resides in the fact that it is algorithm-agnostic and can hence be used by any routing algorithm. Through graph transformation, the algorithm forces any SoA algorithm to visit an ordered set of specified nodes. While LARAC-SN is bounded to the specific CSP problem and can only handle one candidate per specified node, MITH can be used for any routing problem and can deal with several candidates per specified node by automatically choosing the most appropriate one. For the specific case of the CSP problem with a single candidate per specified node, we show that LARAC-SN presents a much better runtime than algorithms extended with MITH (at least around 10 times faster in our simulations). However, while LARAC-SN is slightly sub-optimal, MITH has the potential of reaching optimality for any problem, but at the cost of a higher runtime. We further observe that the runtime of both LARAC-SN and MITH increases with the number of specified nodes.

After reviewing related work in Sec. 3.5.1, Sec. 3.5.2 and 3.5.3 respectively present the LARAC-SN and MITH algorithms. Sec. 3.5.4 reports on our performance evaluation and Sec. 3.5.5 summarizes the results.

3.5.1 Related Work

Since the emergence of the SDN and NFV technologies, a wide range of work has been addressing the algorithmic problem of routing flows through specified nodes. We classify the existing approaches in six overlapping categories for which we list the most representative examples and with respect to which we highlight the contributions of the work in the present section.

Unordered specified nodes. The original literature on routing through specified nodes considered the problem of routing through an unordered set of nodes [SK66]; [Dre69]; [Iba73]. That is, the set of specified nodes can be visited in any order. Recent work also studied this problem [Gom+15]; [And16]; [MGT16]; [Gom+17]. In this section, we consider that the order in which the nodes have to be visited is fixed. Further, these proposals do not consider several candidate nodes for the specified nodes, which our MITH proposal does.

Offline solutions. Several offline algorithms have been proposed [Add+15]; [RS16]; [Gha+16]; [Viz+17]. These algorithms tackle the problem globally, i.e., optimize the solution for a set of routing requests. In Sec. 2.3.2, we have seen that the routing procedure of the DetServ architecture must solve requests online. Hence, in the context of this thesis, we consider the online problem, i.e., we aim at solving the problem independently for each routing request.

Solutions based on integer linear programming. Some proposals model the problem as an integer linear program (ILP) and obtain a solution using an ILP solver [Bas+14]; [Add+15]; [RS16]; [MGT16]. In this section, we avoid formulating the problem as an ILP. Indeed, this often leads to high runtime and requires a new ILP formulation for every different type of routing problem.

Special purpose algorithms. Several papers propose a special-purpose algorithm for dealing with the SNE of specific problems [SK66]; [Dre69]; [Iba73]; [Gom+15]; [And16]; [MGT16]; [Gom+17]; [MMP15]; [Kuo+16]; [Gha+16]; [Viz+17]. Our proposed LARAC-SN algorithm falls into this category for the CSP problem. On the contrary, our MITH approach is independent of the routing problem and can be used along with any SoA routing algorithm.

Solutions based on layering. A couple of solutions duplicate the subject graph into different layers, the interconnections of which represent the different functions [Bar+15]; [RS16]; [Gha+16]. Our MITH algorithm belongs to this category. Existing proposals based on layering however then run a special-purpose algorithm for the specific problem the authors are dealing with. In contrast, our proposed MITH algorithm runs any SoA algorithm on the layered graph. As such, any routing algorithm can use MITH to deal with the OSNE of an existing problem, making our proposal more general.

Loop-free solutions. Some proposals require the solution to be loop-free [Gom+15]; [And16]; [Gom+17]; [MGT16]; [Add+15]. A reason for doing this is the resulting complex configuration of forwarding devices, as they need a mechanism to forward the same packet differently based on whether it traversed the device already or not. A way of solving this issue is to rely on source routing. This is the approach we take in chapter 4. As a result, like other works [SK66]; [Kuo+16], we do not define such a constraint and we consider that paths can contain loops.

3.5.2 LARAC-SN: OSNE of the CSP Problem

In this section, we consider the specific unicast CSP problem and its OSNE with a single candidate per specified node ($c = 1$). We propose Lagrange relaxation based aggregated cost for specified nodes (LARAC-SN), a heuristic for this problem. After mathematically defining the OSNE of the SP and CSP problems (Sec. 3.5.2.1), we first consider the OSNE of the SP problem (Sec. 3.5.2.2). Indeed, our proposed LARAC-SN heuristic is based on this problem. Then, we present the LARAC-SN heuristic in Sec. 3.5.2.3.

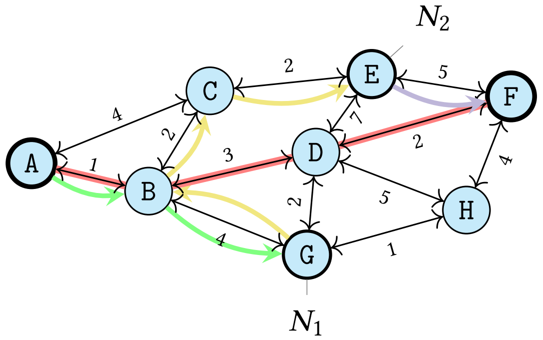


Figure 3.17: Illustration of the OSNE of the SP problem with one candidate per specified node. The solution of the SP problem is shown in red (A-B-D-F). The solution of the OSNE of the SP problem corresponds to the concatenation of the SPs (green (A-B-G), yellow (G-B-C-E), purple (E-F)) between the successive pairs of specified nodes.

3.5.2.1 Mathematical Formulation

From the mathematical formulation of the CSP problem in Sec. 3.1.3, the formulation of its OSNE with one candidate per specified node is straightforward. The solution x further has to belong to the set of paths that traverse nodes $N_1, \dots, N_n$. Let $P_{sd}^{N_1,\dots,N_n}$ denote the intersection of this set of paths and P_{sd}. The OSNE of the CSP problem with one candidate per specified node can then be formulated as

$$z_{\text{opt}} = \min_{\mathrm{x} \in P_{sd}^{N_1,\dots,N_n}} \quad \mathrm{c}^{\mathsf{T}}\mathrm{x} \tag{3.4}$$

$$\text{s.t.} \qquad \mathrm{d}^{\mathsf{T}}\mathrm{x} \leq d. \tag{3.5}$$

The OSNE extension of the SP problem with one candidate per specified node then corresponds to Eqn. 3.4 only.

3.5.2.2 Solving the OSNE of the SP Problem

The adaptation of any SP algorithm to the OSNE of the SP problem with one candidate per specified node is straightforward. The solution corresponds to the concatenation of the SPs between the successive pairs of specified nodes (Fig. 3.17). We refer to such an algorithm as a shortest path with specified nodes (SP-SN) algorithm, which can be implemented using any SP algorithm (e.g., the Dijkstra algorithm [Dij59]).

3.5.2.3 Description of LARAC-SN

As we have seen in Sec. 3.3, the CSP problem has been thoroughly investigated in the literature and a wide range of algorithms have been proposed [Guc+17]; [PR02]. Among others, the CBF algorithm [Wid94] is often considered as the reference optimal algorithm. However, unlike for the SP problem, the adaptation of CSP algorithms for visiting a set of ordered specified nodes is not straightforward. Indeed, because of the constraint metric, the search cannot be split as for the SP problem, as this would require to determine the distribution of the constraint among the different segments, which is not straightforward. This is illustrated in Fig. 3.16. With a constraint bound of 15, CBF finds the path A-B-D-F (highlighted in red) as the optimal CSP from A to F. If nodes G and E have to be visited, the found path is not valid anymore. Because of the constraint, a valid path cannot be found by splitting the search among different segments, as it can be done for the SP problem. Indeed, what constraint bound should be chosen for the A-G segment? For example, the strategy of equally distributing the constraint would fail in the example of Fig. 3.16. Indeed, the optimal path visiting the specified nodes being A-B-G-D-E-F (shown in green), equally dividing the constraint among the different segments would allocate a bound of 5 to all the segments, thereby preventing the algorithm from finding the A-B-G segment (which has a constraint metric of 7) belonging to the optimal solution.

The state-of-the-art LARAC algorithm. As observed in Sec. 3.3.3, the LARAC algorithm generally achieves one of the best performance among the existing CSP algorithm. The algorithm is based on the *Lagrange relaxation technique*, a mathematical optimization technique that allows to solve a constrained problem by removing some of the constraints and by introducing them in the optimization objective [BV04]. For example, the Lagrange relaxation of the CSP problem (Eqn. 3.1–3.2) is

$$L(\lambda) = \min_{\mathrm{x} \in P_{sd}} \quad \mathrm{c}^{\mathsf{T}}\mathrm{x} + \lambda(\mathrm{d}^{\mathsf{T}}\mathrm{x} - d). \tag{3.6}$$

It can be shown that, if the original problem is feasible, then there is an optimal solution to

$$z_L = \max_{\lambda \in \mathbb{R}_+} L(\lambda), \tag{3.7}$$

that is a feasible solution of the original problem. The idea of the Lagrange relaxation technique is then to obtain a solution to the primal problem (3.1)–(3.2) by solving the dual problem (3.7), which is potentially easier. Solving problem (3.7) requires to solve the relaxed problem (3.6) several times in order to find the λ maximizing $L(\lambda)$. For the CSP problem, the relaxed problem corresponds to a SP problem with a modified cost function $c'_i(\lambda) = c_i + \lambda d_i$. This means that, for solving the CSP problem, the LARAC algorithm subsequently runs several SP searches optimizing the combined metric $c'_i(\lambda)$ with different λ values. The computation of the λ values and the terminating condition can be found in the original references of the LARAC algorithm [AN78]; [HZ80]; [BG96]; [Jüt+01] and in [Guc+17]. The algorithm always finds a solution if one exists (i.e., it is *complete*) but is, by the properties of the Lagrange relaxation technique [BV04], not optimal.

Adaptation of the LARAC algorithm. We propose to also use the Lagrange relaxation technique for the OSNE of the CSP problem with one candidate per specified node. The Lagrange relaxation of problem (3.4)–(3.5) is

$$L(\lambda) = \min_{\mathrm{x} \in P_{sd}^{N_1,\ldots,N_n}} \quad \mathrm{c}^{\mathsf{T}}\mathrm{x} + \lambda(\mathrm{d}^{\mathsf{T}}\mathrm{x} - d), \tag{3.8}$$

which corresponds to the OSNE of the SP problem with the combined metric $c'(\lambda)$. As such, the LARAC algorithm can simply be adapted by running an SP-SN (see 3.5.2.2) algorithm at each iteration instead of a simple SP algorithm. The computation of the λ values and the terminating condition are identical to the original LARAC algorithm. We refer to this algorithm as Lagrange relaxation based aggregated cost for specified nodes (LARAC-SN). By the properties of the original LARAC algorithm, LARAC-SN is complete but not optimal. By including the constraint metric in the combined optimization metric $c'(\lambda)$ of the SP-SN runs, the LARAC-SN algorithm automatically distributes the usage of the constraint metric budget along the different segments between the different specified nodes.

3.5.3 Mole in the Hole (MITH)

While LARAC-SN is an interesting solution for the OSNE of the CSP problem with one candidate per specified node, it presents several drawbacks: *(i)* it is not optimal (though close to optimal – see Sec. 3.5.4.2), *(ii)* it cannot deal with several candidates per specified node and *(iii)* it is tailored to the unicast CSP problem. In this section, we present mole in the hole (MITH), our solution to overcome the limitations of LARAC-SN. First, MITH allows to obtain an optimal solution for the OSNE of any routing problem, overcoming limitations *(i)* and *(iii)* of LARAC-SN. Second, MITH allows to deal with any number of candidate nodes per specified node, thereby overcoming limitation *(ii)* of LARAC-SN.

MITH is a graph transformation algorithm based on layering. The introduced layers correspond to copies of the original graph for routing before, between and after the specified nodes. The transformed graph enables SoA algorithms to solve the OSNE of their original problem. We detail the graph transformation procedure in Sec. 3.5.3.1. Because routing is then performed on the transformed graph, the original routing request has to be mapped to the transformed graph and the obtained solution has to be transformed back to the original graph. These two procedures are described in Sec. 3.5.3.2 and 3.5.3.3, respectively. The section is concluded by a discussion of the limitations of MITH (Sec. 3.5.3.4).

3.5.3.1 Graph Transformation

MITH defines a new graph consisting of several layers. Each layer corresponds to an exact copy of the original graph, including the metrics associated to the different edges. The different layers are ordered and correspond to the routing before, between and after the different specified nodes. That is, if there are n specified nodes, $n + 1$ layers are defined. Subsequent layers are interconnected by connecting pairs of nodes corresponding to the same original node. An interconnecting edge corresponds to the visit of the original node corresponding to the nodes used for the interconnection. That is, the interconnections between the layers correspond to the visit of the different specified nodes. For a specified node with several candidates, the corresponding layers are simply interconnected several times through the corresponding nodes. The edge(s) created for interconnecting the layers can be assigned metric values quantifying the visit of this node.

The procedure is illustrated in Fig. 3.18. Let us consider a request with two specified nodes ($n = 2$), the first one having two candidates ($N_1^a = n_1$ and $N_1^b = n_3$) and the second one having only a single candidate ($N_2 = n_4$). As there are two specified nodes, three layers are defined. The first two layers are interconnected via nodes n_1 and n_3, corresponding to the two candidates of the first specified node (N_1^a and N_1^b), and the last two layers are interconnected via node n_4, corresponding to the single candidate of the second specified node (N_2).

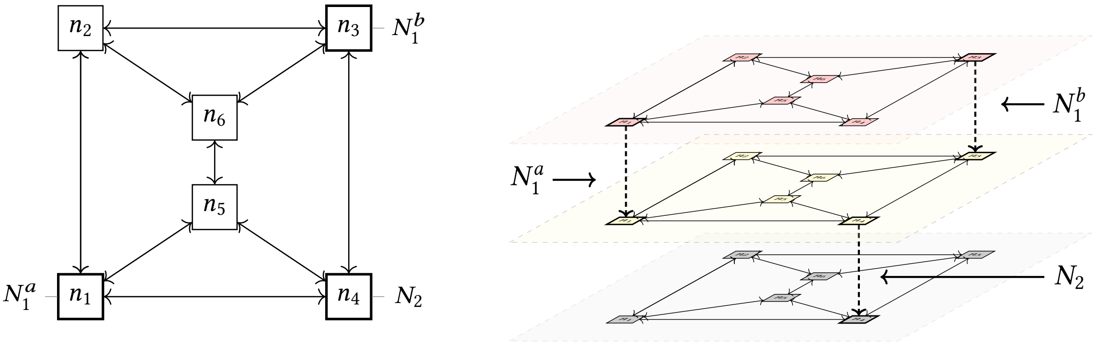


Figure 3.18: On the left, example graph for the illustration of the MITH graph transformation algorithm. On the right, obtained transformed graph for a request with two specified nodes, the first one having two candidates (n_1 and n_3) and the second one having only one candidate (n_4).

3.5.3.2 Request Transformation

The original request has to be mapped to the transformed graph. From a graph point of view, any routing request without specified intermediate nodes can be represented by a set of source nodes and a set of destination nodes. The transformation is then straightforward. As the source nodes have to be visited before the first specified node, the source nodes in the transformed graph correspond to the copy of the source nodes of the original request in the first layer. Similarly, as the destination nodes have to be visited after the last specified node, the destination nodes in the transformed graph correspond to the copy of the destination nodes of the original request in the last layer.

3.5.3.3 Solution Transformation

Once the SoA algorithm returns a solution on the transformed graph, the latter has to be mapped back to the original graph. The solution in the original graph corresponds to the concatenation of all the original edges corresponding to the edges used in the different layers, except the interconnecting edges (which do not have any corresponding edge in the original graph).

3.5.3.4 Limitations of MITH

While MITH presents itself as a general algorithm for enabling routing through specified nodes for any routing algorithm, it has some limitations. First, if the original graph includes local constraint metrics (as defined in Sec. 3.1, e.g., finite bandwidth usage capacity), the routing algorithm extended with MITH could lose its completeness or optimality. Indeed, using the Mn taxonomy defined in Sec. 3.6, such a constraint on the layered graph is an M∞ local constraint metric, which, as we show in Sec. 3.6, results in the loss of the completeness and optimality of routing algorithms. Note that this observation is also valid for LARAC-SN. Second, node- or edge-disjointness on the transformed graph does not ensure node- or edge-disjointness on the original graph. That is, using MITH, a multipath routing algorithm can potentially lose its main property of finding disjoint paths. Hence, MITH does not support multipath routing. Third, as mentioned, MITH can only deal with an *ordered* set of specified nodes.

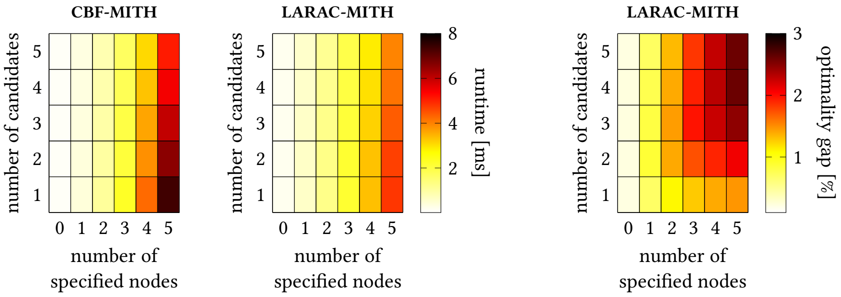


Figure 3.19: Runtime and OG of CBF with MITH (CBF-MITH) and LARAC with MITH (LARAC-MITH) for different numbers of specified nodes and candidates per specified node. Since CBF-MITH is optimal, its OG is omitted. Each point corresponds to the average of 10,000 runs.

3.5.4 Evaluation

The goal of our evaluation is twofold. First, in Sec. 3.5.4.1, we give an insight on the impact of MITH on the performance of SoA routing algorithms. Second, in Sec. 3.5.4.2, as LARAC-SN and MITH can both solve the OSNE of the unicast CSP problem with one candidate per specified node, we compare the performance of both algorithms for this specific problem.

We perform both evaluations on the example of the unicast CSP problem. We use the topologies of the Topology Zoo [Kni+11] which are connected, have between 10 and 100 vertices and less than 200 edges. The delay d_i of an edge is defined as the propagation delay and its cost c_i is defined as $1 + 1/d_i$. This ensures that LC and LD paths are not identical. For each combination of topology, number of specified nodes and number of candidate nodes per specified node, we generate 100 random sets of candidate nodes. For each of these, we generate 100 random unicast requests (random source and destination) with a delay bound randomly uniformly distributed between the minimum (delay of the LD path) and maximum possible values (delay of the LC path). This setup leads, for each scenario, to 10,000 requests per combination of topology and number of specified and candidate nodes. The evaluations were ran on an Ubuntu 16.04 PC equipped with an Intel Core i7-4790 CPU @ 3.60GHz.

3.5.4.1 Influence of MITH on the Performance of Algorithms

We observe the runtime and OG of CBF and LARAC [AN78]; [Jüt+01] when used with MITH for different numbers of specified nodes (from 0, i.e., without MITH, to 5) and for different numbers of candidate nodes per specified node (from 1 to 5). The results are shown in Fig. 3.19. Each cell represents the average runtime or OG observed for 10,000 runs.

Runtime. We observe that the impact of MITH on the runtime of the algorithms increases with the number of specified nodes. However, surprisingly, the runtime of the algorithms decreases with the number of candidate nodes. This is due to the fact that, because of the randomness of our evaluation, adding candidates potentially allows the algorithms to reach the last layer with shorter paths (i.e., faster). A thorough evaluation of the impact of MITH depending on the algorithm in use, on the location of the candidates in the graph and on the tightness of the delay constraint is out of the scope of this section and is left for future work.

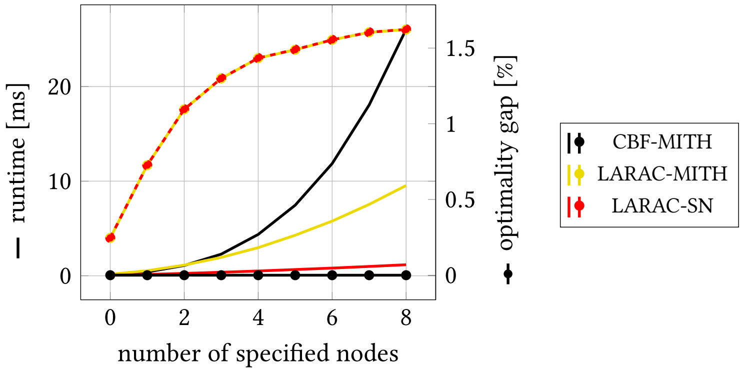


Figure 3.20: Runtime and OG of LARAC-SN, CBF with MITH (CBF-MITH) and LARAC with MITH (LARAC-MITH) for different numbers of specified nodes and one candidate per specified node. Each point in the graph corresponds to the average of the values observed for the 10,000 runs corresponding to this point.

Optimality gap. In terms of OG, with CBF as optimal benchmark, we observe that both increasing the number of specified nodes and increasing the number of candidate nodes per specified node increases the OG of LARAC. However, the OG remains low: on average, it never exceeds 2.6% in our evaluations.

3.5.4.2 LARAC-SN vs. LARAC with MITH

Fig. 3.20 shows the runtime and OG of LARAC-SN and LARAC and CBF extended with MITH for different numbers of specified nodes (from 0 to 8). Each point in the graph corresponds to the average of the values observed for the 10,000 runs corresponding to this point.

Runtime. As expected, since increasing the number of specified nodes increases the size of the graph on which path finding is performed, the runtime of CBF and LARAC with MITH increases exponentially with the number of specified nodes. As already observed in Sec. 3.3.2, LARAC scales better than CBF. We observe that LARAC-SN presents a much better runtime behavior. Indeed, the runtime of LARAC-SN is around 10% of the runtime of LARAC with MITH.

Optimality gap. At the cost of its high runtime, CBF with MITH is optimal. On the contrary, LARAC-SN and LARAC with MITH are both sub-optimal but their OG stays lower than 1.62% on average. They both exhibit the exact same OG behavior, as they are actually always returning the same paths. Interestingly, while the OG increases with the number of specified nodes, the increase seems to saturate.

3.5.5 Summary

The problem of routing through intermediate specified nodes has become more and more an important issue. Indeed, the problem finds applications in new emerging fields such as NFV and SFC, in which applications can also have predictable latency requirements. In this section, we proposed two algorithms for this problem. First, we proposed LARAC-SN, an extension of the SoA LARAC CSP algorithm which allows to find a CSP visiting specified intermediate nodes. While sub-optimal, we showed that LARAC-SN exhibits a small OG at a low runtime cost. Second, we proposed MITH, a graph transformation algorithm forcing any SoA algorithm to visit an ordered set of specified nodes. MITH further allows to define a set of candidate nodes per specified node such that the SoA routing algorithm used automatically selects the best candidate in the sense of the routing problem considered (e.g., least-hop, LC delay-constrained, etc.). We showed that MITH has the potential of finding the optimal solution to any problem, however at the cost of an increased runtime. We further showed that the performance of both LARAC-SN and MITH degrades as the number of specified nodes increases.

3.6 Handling the Optimality/Completeness Loss due to DetServ

The routing algorithms discussed in this chapter aim at coping with the demanded latency requirements of applications while optimizing the use of network resources. These algorithms rely on the optimal substructure property (OSP), which states that an optimal path contains other optimal paths within it. However, in this section, we show that QoS metrics such as those defined by the DetServ latency model, e.g., queuing delay and buffer consumption, do not satisfy this property, which implies that the used algorithms lose their optimality and/or completeness properties. As a result, to prevent the violation of the latency guarantees of applications and to ensure optimal use of network resources, we need new solutions for the DetServ architecture. In this section, we tackle this problem by first proposing a new so-called *Mn taxonomy* defining new metric classes. The taxonomy defines an M*n* metric as a metric which requires the knowledge of the *n* previously traversed edges to compute its value at a given edge. We will see that this taxonomy allows to determine when SoA algorithms based on the OSP lose their optimality and/or completeness properties (see Tab. 3.4). For these new classes of metrics introduced by the taxonomy, new optimal and complete algorithms are needed. We then present three solutions: *A*Prune*, edge-based Dijkstra (EBD) and a graph transformation algorithm (GTA). On the one hand, A*Prune [LR01], a SoA algorithm, and EBD, a newly proposed extension of the Dijkstra algorithm [Dij59], are algorithms for the specific, respectively, MCSP and SP problems, which keep their optimality and completeness properties for the new defined classes of metrics. On the other hand, GTA is an extension that can be applied to any SoA algorithm for recovering its optimality and completeness properties. Through evaluations, we show that SoA algorithms based on the OSP indeed lose their optimality and/or completeness properties depending on the M*n* classification of the considered metric(s). Further, we show that our proposed solutions are indeed complete and optimal for their respective problems but that this unfortunately comes at the price of an increased runtime.

After empirically evidencing in Sec. 3.6.1 that SoA algorithms can lose their optimality properties for metrics resulting from the DetServ architecture, Sec. 3.6.2 introduces the newly proposed *Mn taxonomy* for classifying routing metrics. In Sec. 3.6.3, we present *A*Prune*, EBD and GTA for keeping and recovering the completeness and optimality properties. Sec. 3.6.4 reports on the evaluation of our solutions, Sec. 3.6.5 analyzes when our solutions can be applied and Sec. 3.6.6 summarizes the contributions of this section.

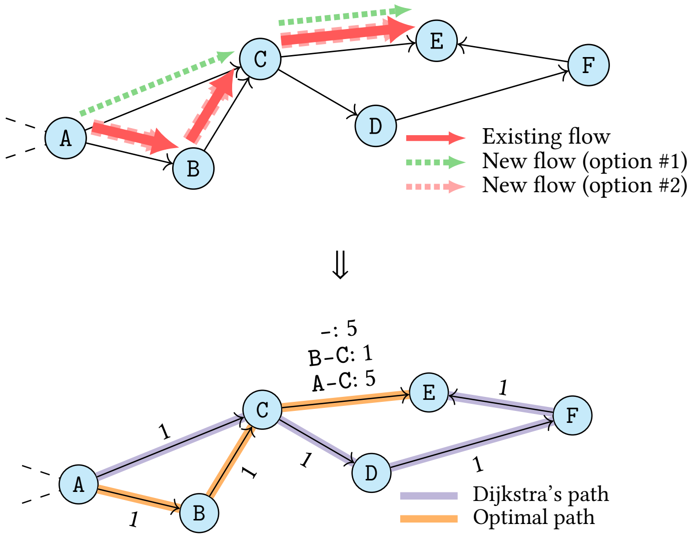


Figure 3.21: Example scenario involving an M1 metric. As elaborated in Sec. 3.6.1, we observe that, in such a situation, Dijkstra finds a sub-optimal path.

3.6.1 Motivation: Violation of the Optimal Substructure Property

SoA routing algorithms rely on the fact that the metric values associated to the edges of the graph are *static* (i.e., constant) for a given routing request [AMO93]; [GY05]. However, this is not always the case.

Let us examine the example depicted in Fig. 3.21, which considers as metric the queuing delay experienced at the ingress of each link (referred to as delay in this section). Let us assume that the network carries a flow on path A-B-C-E (continuous red arrows in Fig. 3.21). Because there is only one flow in the network, no queuing occurs at any link and all the links have the same delay metric value (e.g., 1 in this example). A new flow has to be routed from A to E. In order to reach C, the flow can either be routed through A-C or A-B-C. If it is routed through A-C, the flow may generate queuing at the egress of C, i.e., at link C-E, because of potential collisions of packets of the two flows coming from the two different ingress links of C. This implies that the delay metric value of C-E should be greater than without collisions (e.g., 5 in this example). On the other hand, if the new flow is routed through A-B-C, it will follow the same path as the original flow, thereby generating no queuing delay at any link[6]. Hence, all links keep the same original delay metric value (1 in our case). We observe that the delay metric value associated to C-E depends on the previous edge traversed by the new flow. This is shown in the lower diagram of Fig. 3.21. This scenario corresponds to the ILS concept of the DetServ TBM described in Sec. 2.4.7. Indeed, using ILS results in buffer consumption and delay values which depend on the previously visited physical link (Fig. 2.30).

In such a setting, the LD path from A to E is A-B-C-E, with a delay of 3. However, the Dijkstra algorithm (see Sec. 3.2), supposedly optimal, finds the path A-C-D-F-E, with a delay of 4. Indeed, Dijkstra performs a BFS and keeps only track of the best path to reach each node. While finding a path from A to E, Dijkstra will store A-C as the best path to reach C because it has a total delay of 1,

[6] We assume that the flows arrived in A via the same link and with the same total delay, thereby generating no queuing delay at the ingress of A-B.

Constraint type (Sec. 3.1)	M0	M1	··· Mn ···	M∞
Local constraint	C O	C O	C O	C O
Global constraint	C O	C O	C O	C O
Global optimization	C O	C O	C O	C O

Table 3.4: Impact of Mn metrics on the completeness (**C**) and optimality (**O**) of SoA algorithms based on the OSP.

which is lower than 2, the delay of `A-B-C`, the only other path to reach `C`. Then, Dijkstra will have two possibilities to reach the destination. Either using `C-E` with a delay of 5, or following `C-D-F-E` with a delay of 3. Since `C-D-F-E` has a lower delay, Dijkstra will choose it and return `A-C-D-F-E`, with a total delay of 4, as final solution. We observe that, in this situation, Dijkstra is not able to find the optimal path. We will show in Sec. 3.6.2.2 that this is due to the fact that the OSP, stating that an optimal path contains other optimal paths within it, is not satisfied.

Sec. 3.6.2.3 presents other scenarios for which SoA algorithms based on the OSP (i.e., nearly all SoA routing algorithms) lose either their optimality or their completeness property, or both. Currently, operators solve this problem by allowing sub-optimality or by having a looser modeling of QoS parameters, thereby wasting network resources, or, in the worst case, by having a too optimistic modeling of QoS parameters, thereby potentially leading to violations of SLAs. As a result, new optimal and complete algorithms for dealing with this type of metrics are needed.

3.6.2 The Mn Taxonomy

In this section, we present our novel routing metric taxonomy, the *Mn taxonomy*, classifying metrics into classes based on the amount of previous edges needed for computing their value at a given edge. We refer to a metric requiring the knowledge of the n previously traversed edges as an Mn metric.

3.6.2.1 M0 Metrics: No Additional Information Required

M0 metrics correspond to the traditional metrics considered in the SoA. The metric value associated to an edge depends only on the edge itself and requires no information on the other edges previously traversed. Examples of M0 metrics are the propagation delay and the total capacity of a link.

3.6.2.2 M1 Metrics: Values Depending on the Previous Edge

Definition. M1 metrics correspond to metrics whose value at a given edge depends on the previous edge used to reach the given edge.

Motivation: queuing delay. The example developed in Sec. 3.6.1, based on the ILS concept of DetServ, corresponds to an M1 metric. Indeed, depending on which ingress link to `C` is used, the queuing delay at the egress of `C` is different. As a result, the metric value of `C-E` depends on the previously traversed edge and the metric is an M1 metric.

Impact as global optimization metric. The example developed in Sec. 3.6.1 (and illustrated in Fig. 3.21) corresponds to an M1 metric used as global optimization metric for a SP problem. We have seen that the Dijkstra algorithm loses its optimality. The reason for this is that Dijkstra relies on the OSP, which states that sub-paths of optimal paths are also optimal [Cor+09]. While the OSP is satisfied for M0 metrics, it is not necessarily satisfied anymore for M1 metrics. Other routing algorithms typically also rely on the OSP, either because they are based on Dijkstra itself, or because they are based on dynamic programming or greedy approaches which are themselves based on the OSP [Cor+09]. Consequently, other optimal SP algorithms such as BF and A* (see Sec. 3.2 are also affected. Hence, when an M1 metric is used as optimization metric, SoA algorithms based on the OSP lose their optimality property. Note that, as the metric is only used for optimization, completeness is not impacted. This is summarized in Tab. 3.4 and will be confirmed in our evaluations (Sec. 3.6.4.3).

Impact as global constraint metric. Let us consider the M1 metric in Fig. 3.21 as global constraint metric (with a bound of 3.5) and further define the hop count (an M0 metric) as global optimization metric. This corresponds to a CSP problem. CBF [Wid94], an optimal CSP algorithm, is similar to Dijkstra. It performs a BFS and keeps only track of the best path at each node. However, it discovers paths in order of the constraint metric and stops once the bound is reached. In our example, CBF would hence also find `A-C` as the best path to reach `C`. From this path, the destination `E` cannot be reached within the deadline. Hence, CBF will conclude that no path is available. However, `A-B-C-E`, with a delay of 3, is a valid solution. As a result, CBF is incomplete. It can easily be shown that, with a bound of 4.5, CBF would find `A-C-D-F-E`, with a delay of 4, which is sub-optimal (`A-B-C-E`, with a delay of 3, still being the optimal path). As for Dijkstra, this is due to the fact that the OSP is not satisfied anymore. Hence, other optimal CSP algorithms, which are all based on Dijkstra and hence on the OSP, are also affected. Consequently, when an M1 metric is used as a global constraint metric, SoA algorithms based on the OSP lose both their completeness and optimality properties. This is shown in Tab. 3.4 and will be confirmed in our evaluations (Sec. 3.6.4.4).

Impact as local constraint metric. It can also be easily shown that M1 local constraint metrics also lead to the sub-optimality and incompleteness of SoA algorithms based on the OSP (see Tab. 3.4). Indeed, the OSP is also not necessarily satisfied. In the DetServ architecture, this corresponds to the hasAccess method of the TBM when it uses ILS. Indeed, in this situation, the access control is based on the buffer consumption of flows, which, because of ILS, depends on the previously traversed physical link.

3.6.2.3 M∞: Values Depending on the Complete Path

Definition. M∞ metrics correspond to metrics whose value at a given edge depends on the complete path traversed to reach the current edge.

Motivation: buffer management. Let us consider a metric representing the buffer consumption of a flow. As we have seen in Sec. 2.2.6, per DNC, while traversing a given path, the burstiness of a flow is increased at each hop by an amount depending on the flow and on the hop characteristics. The buffer consumption of a flow at a given node depends, among other things, on the burstiness of this flow. Hence, the buffer consumption of a flow at a given hop depends on all the previously traversed links. As a result, in the DetServ architecture, any metric that requires the computation of

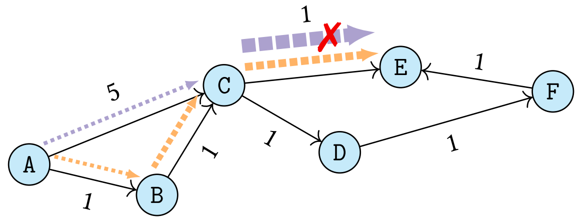


Figure 3.22: Example scenario involving an M∞ metric. We observe that the new flow will be rejected or accepted at C-E depending on where it is coming from.

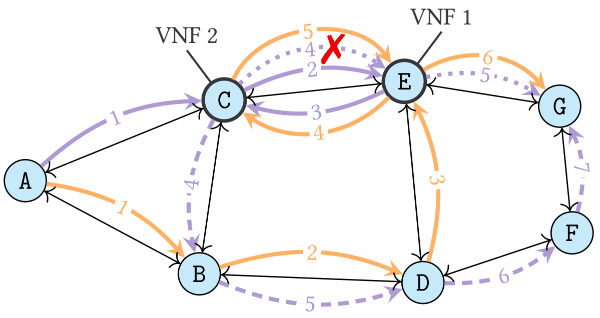


Figure 3.23: Example scenario involving an M∞ metric. The bandwidth consumption of the new flow at a link depends on whether or not it visited this link before. Hence, acceptance of the new flow at a link depends on all the previously visited links.

the buffer consumption of a flow is an M∞ metric. Indeed, we have seen in Sec. 2.4.6 that the burst of a flow depends on the delay of all the previously traversed queues. As the hasAccess method of the TBM is based on the burst of a flow, it is a M∞ local constraint metric. To avoid that, the buffer consumption of a flow at a hop can be computed based on the latency requirement of the flow, as this value is for sure greater than the delay of all the previously traversed queues (otherwise the path does not satisfy the requirements of the flow). In this case, the metric is not anymore an M∞ metric. However, as elaborated above, if ILS is used, the metric remains an M1 metric.

Motivation: routing through service function chains. Let us consider a metric representing the total bandwidth consumption of a flow at each link. While routing through SFCs, loops can be introduced. Hence, the total bandwidth consumption of a flow at a given link depends on how many times the flow already visited the given link. As a result, all the previously visited links have to be known and this corresponds to an M∞ metric. This corresponds to the situation described in Sec. 3.5.3.4 for MITH and LARAC-SN.

Impact as local constraint metric. Let us consider the buffer management scenario elaborated above and illustrated in Fig. 3.22. We consider the problem of routing a flow from A to E (Fig. 3.22) through the least-hop path (M0 metric). Analogously to the DetServ hasAccess method, We consider a local constraint metric rejecting flows consuming too much buffer space. In Fig. 3.22, each link is labeled with its queuing delay (generated by already embedded flows). Note that this queuing delay is not used as a metric. As per DNC (see Sec. 2.2.6, the burstiness of the flow, i.e., its buffer consumption, will increase approximately proportionally to the delay it experiences along its path. Fig. 3.22 shows the two options for routing the new flow. The width of the arrows represent the buffer consumption of the flow. While routing, Dijkstra will save the path A-C, with a hop count of 1, as the best path towards C. However, because the queuing delay experienced at A-C is 5, the burstiness of the flow greatly increased and it cannot be accepted at link C-E. As a result, Dijkstra would either find no solution (thereby losing completeness) or find A-C-D-F-E, with a hop count of 4, if the latter has enough buffer space available. However, A-B-C-E, with a hop count of 3, has a low delay, thereby only slightly increasing the burstiness of the flow and hence allowing it to use C-E. In this case, Dijkstra is hence sub-optimal. This is again due to the fact that the OSP is not satisfied.

Let us further consider the SFC routing example developed above (illustrated in Fig. 3.23) and the problem of routing a flow on the least-hop path (M0 metric) from A to G (Fig. 3.23) visiting the two VNFs E and F in the specified order. The flow consumes 700 Mbps and each link has 1 Gbps available bandwidth. Hence, the flow can only visit each link once. A least-hop algorithm would first reach the first VNF, i.e., E, through the least-hop path, i.e., A-C-E. Then, it would visit the second VNF, i.e., C, through the least-hop path from E, i.e., E-C. Finally, it would try to reach the destination with the least-hop path from C, i.e., C-E-G. However, this would imply visiting a second time C-E, which is not allowed. If the algorithm does not notice that the access to C-E is refused (e.g., if it cached the result of the access control when it first traversed the link), it will return an invalid path and hence be incomplete. If the algorithm notices that the access to C-E is refused, it would follow the second least-hop path, i.e., C-B-D-F-E, thereby finding A-C-E-C-B-D-F-E, with a hop count of 7, as final solution. However, A-B-D-E-C-E-G is a valid path with a hop count of 6. Hence, the algorithm loses its optimality. This is why LARAC-SN and MITH can lose their optimality if a local constraint metric is involved (see Sec. 3.5.3.4).

As a result, when an M∞ metric is used as a local constraint metric, SoA algorithms based on the OSP lose both their completeness and optimality properties (Tab. 3.4).

Impact as global optimization or constraint metric. As an M1 metric is also an M∞ metric, it can easily be shown that the impact as global optimization and constraint metric is the same for both M1 and M∞ metrics (Tab. 3.4).

3.6.2.4 M*n*: Values Depending on the *n* Previous Edges

As a generalization, we further introduce the class of M*n* metrics. M*n* metrics correspond to metrics whose value at a given edge depend on the *n* previous edges used to reach the current edge. Obviously, for the same reasons as for M∞ metrics, the impact of SoA algorithms based on the OSP is the same as for M1 metrics. This is shown in Tab. 3.4.

3.6.3 Solutions for the M*n* Taxonomy

In this section, we present three opportunities (Sec. 3.6.3.1, 3.6.3.2 and 3.6.3.3) to optimally and completely solve problems with M*n* metrics, $n > 0$.

3.6.3.1 Existing Solution: A*Prune

*A*Prune* [LR01] is a complete and optimal SoA algorithm able to solve the SP and MCSP problems. Although similar to Dijkstra, A*Prune does not rely on the OSP but is only faster when it is satisfied. Hence, it keeps its optimality and completeness properties for both M*n* and M∞ metrics. The reason for this is that it does not keep track of only one best path to reach each node. Instead, all feasible paths are kept in memory and the best ones are extended first. Path extension is stopped only once the next path to extend has an optimization metric value higher than the current best path for the destination.

In the example of Fig. 3.21, as Dijkstra, A*Prune will first find the path `A-B-C-E` with a total metric value of 7. However, the next path to extend, namely `A-C`, has a total metric value of 4, which is lower than the current best path to the destination. Hence, A*Prune will further extend `A-C` and thereby find `A-C-E`, which is optimal.

Unfortunately, as already observed in Sec. 3.3.2, this optimality for any type of M*n* and M∞ metric comes at the price of a poor scalability behavior. This will be further shown in our evaluations in Sec. 3.6.4.

3.6.3.2 New Solution: Edge-based Dijkstra (EBD)

In the particular case of the SP problem with an M1 optimization metric, Dijkstra can be slightly adapted. Instead of keeping track of the best path towards each node, our proposed adaptation keeps track of the best path towards each *edge*. We refer to this algorithm as edge-based Dijkstra (EBD).

In the example of Fig. 3.21, instead of keeping track of the best path towards node `C`, EBD will keep track of the best path towards `A-C` (which is `A-C` itself) and towards `B-C` (which is `A-B-C`). Then, when extending these paths to obtain the best path towards `C-E`, both paths will be considered and the path `A-C-E`, missed by the normal Dijkstra, will be found. Once EBD stops, the final solution then corresponds to the best path among those stored at all the ingress links to the destination node.

Because the amount of edges in a graph is usually higher than the amount of nodes, EBD keeps track and extends more paths than the traditional Dijkstra and the usage of EBD for M1 metrics hence results in a runtime increase compared to a normal Dijkstra run. This will be confirmed in Sec. 3.6.4.3.

3.6.3.3 New Solution: Graph Transformation Algorithm (GTA)

In this section, we propose a graph transformation algorithm (GTA) which transforms a graph with M*n* metrics to an equivalent graph with M0 metrics such that any routing problem with M*n* metrics can be solved with a SoA algorithm for the given routing problem with traditional M0 metrics.

Reasoning. For M1 metrics, edges can have as many different metric values as ingress links (plus one for the "*null*" ingress link when the flow starts at the given edge). This is shown in Fig. 3.21. The idea of GTA is to duplicate links as many times as they have different metric values such that each new edge has a static M0 metric value. Let us refer to the original graph with M*n* metrics as the *Mn graph* and to the transformed graph with only M0 metrics as the *M*0 *graph*. Each edge in the

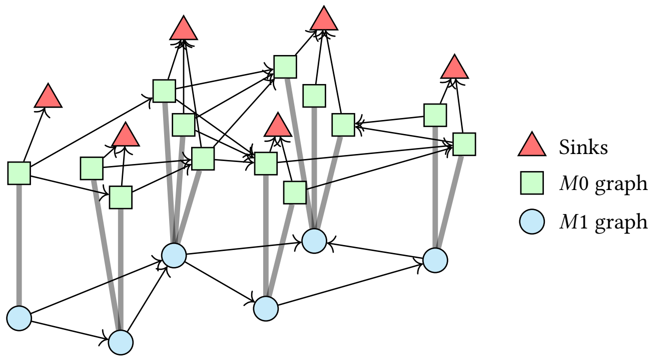


Figure 3.24: Illustration of the GTA procedure for M1 metrics. From the original graph with M1 metrics (the M1 graph), the algorithm creates a new graph (the M0 graph) with M0 metrics on which any SoA algorithm can run to solve the original M1 problem.

M0 graph then corresponds to *(i)* an edge of the original M*n* graph, and *(ii)* a set of *n* previous edges. In this way, edges in the M0 graph have only one metric value (i.e., the metric is now an M0 metric) and SoA algorithms based on the OSP can operate properly.

Algorithm description for M1 metrics. The GTA algorithm for M1 metrics is illustrated in Fig. 3.24. The blue circle nodes correspond to the original M1 graph. From this graph, each node is copied and then duplicated as many times as it has ingress links. Each M0 node (green square nodes in Fig. 3.24) then corresponds to an original M1 node and to one ingress link of this node (including the "*null*" ingress link). Then, M0 edges are obtained by creating edges towards the created M0 nodes from all the M0 nodes corresponding to the source of the original edge to which the destination M0 node corresponds. Each M0 edge then corresponds to *(i)* an edge of the M1 graph, and *(ii)* an ingress link to this edge. Hence, each M0 edge can be assigned a static M0 metric value corresponding to the metric value of the original edge when the given ingress edge is used to reach it. We refer to this procedure as GTA().

Request and result transformation. In order for the original algorithm to run on the transformed graph, the original request has to be mapped. First, the source nodes now correspond to their M0 equivalent which have no ingress link. Secondly, the destination nodes have now several M0 equivalents. To overcome this problem, so-called *sink nodes* have to be created. All the M0 nodes corresponding to the same original M1 node have to be connected to the same sink node with edges whose metric value does not change the metric value of the overall solution (that is, e.g., 0 for an additive metric). The destination(s) of the original request then become(s) the corresponding sink node(s). We refer to the procedure of creating the sink nodes as AddSinks(). Once the algorithm found a solution on the M0 graph, the solution on the M1 graph can be recovered by taking all the M1 equivalents of the elements of the M0 solution returned by the algorithm.

Algorithm description for M*n* metrics. In order to transform a graph with M*n* metrics, the GTA() procedure simply has to be applied *n* times and followed by the addition of sink nodes (Fig. 3.25). Indeed, as GTA() duplicates edges for each ingress link, applying it *n* times will duplicate edges for each possible set of *n* ingress links and hence lead to an M0 graph.

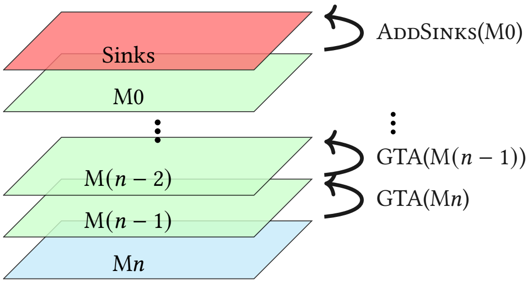


Figure 3.25: Illustration of the GTA procedure for Mn metrics. The procedure described in Fig. 3.24 simply has to be applied n times, provided that the sinks are only added at the end.

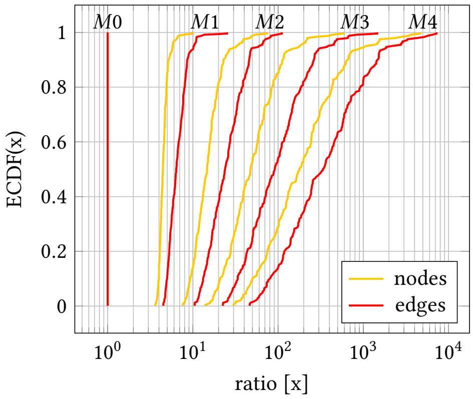


Figure 3.26: Evolution of the number of nodes and edges for the Internet Topology Zoo [Kni+11] topologies for different amount of executions of GTA (including the sinks creation).

Cost of the transformation. The size of the M0 graph increases with n. Fig. 3.26 shows the evolution of the amount of nodes and edges for all the topologies from the Internet Topology Zoo [Kni+11]. We can observe that the amount of nodes and edges increases by up to one order of magnitude for each application of the GTA() procedure. That is, while GTA allows to optimally solve any problem with Mn metrics using algorithms for M0 metrics, this comes at the price of a huge increase in the graph size. An insight on the runtime impact will be given in Sec. 3.6.4.

3.6.4 Evaluation

The goal of the evaluation is to observe the impact of M1 and M∞ metrics on the optimality and completeness of SoA algorithms based on the OSP and to show that our proposed solutions are correct. In particular, we show the influence of M1 and M∞ metrics on one SP (A*) and one CSP (LARAC) algorithm both with and without GTA. Algorithms are compared to A*Prune, which provides a benchmark for both completeness and optimality.

3.6.4.1 Setup

Topologies. We use the topologies from the Internet Topology Zoo [Kni+11] which are connected and have more than 10 nodes. Further, because A*Prune poorly scales both in terms of memory consumption and runtime (see Sec. 3.3.2), we filter out topologies with more than 100 nodes or 200 edges.

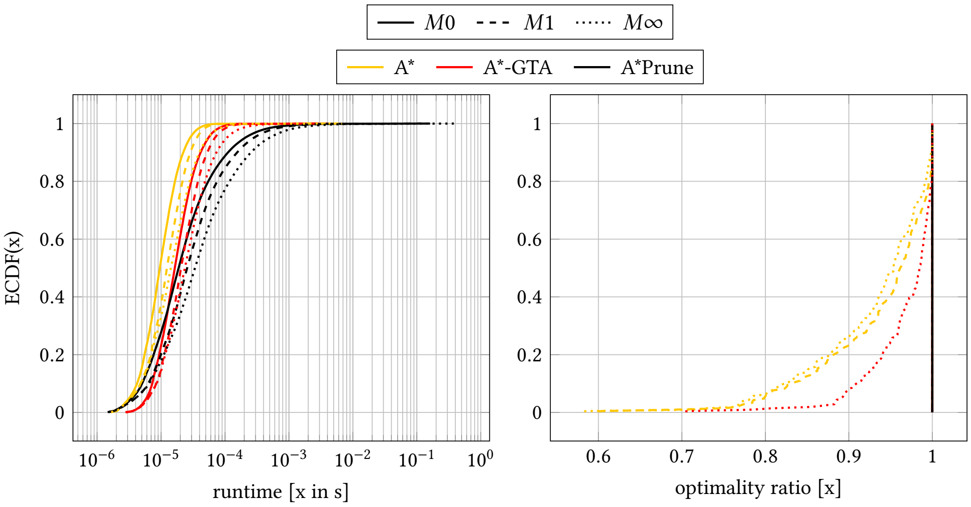


Figure 3.27: Runtime and optimality ratio of A*, A* with one GTA transformation (A*-GTA) and A*Prune for the SP problem with M0, M1 and M∞ optimization metrics.

Requests. For a given topology, source and destination nodes for requests are randomly selected from the whole set of nodes of the topology.

Metrics. We define one M0, one M1 and one M∞ metric. The metric values are random values between 1 and 2. For the M0 metric, the values are defined for each edge. For the M1 metric, the values are defined for each combination of edge and previous edge. For the M∞ metric, the values are defined for each path.

3.6.4.2 Measurement Environment

In both SP and CSP scenarios, each algorithm is ran 20,000 times for each topology and metric type. Prior to these 20,000 runs, 1.000 warm-up runs are used to prevent the Java Hotspot optimizer from influencing the runtime measurements. The evaluation ran on an Intel Core i7-4790 CPU @ 3.60GHz.

3.6.4.3 Shortest Path: Optimality Influence

Setup. We observe the runtime and optimality of A* (described in Sec. 3.2 and A* with GTA applied once (referred to as A*-GTA) for M0, M1 and M∞ metrics using A*Prune as a benchmark. The guess values for both A*Prune and A* correspond to the hop count.

Runtime results. The left plot of Fig. 3.27 shows the ECDF of the observed running times during the simulation[7]. We can observe that the GTA transformation for A* leads to an increase in runtime of around half an order of magnitude. Even though big topologies were not used, the figure also illustrates the poor scalability of A*Prune. Indeed, while A*Prune is sometimes significantly faster than A*-GTA (around 40% of the cases for the M0 metric, around 30% of the cases for the M1 metric and around 20% of the cases for the M∞ metric) its runtime becomes very high for bigger topologies.

[7] Note that, for GTA, the graph creation is not taken into account. Indeed, it can be done once for all the requests.

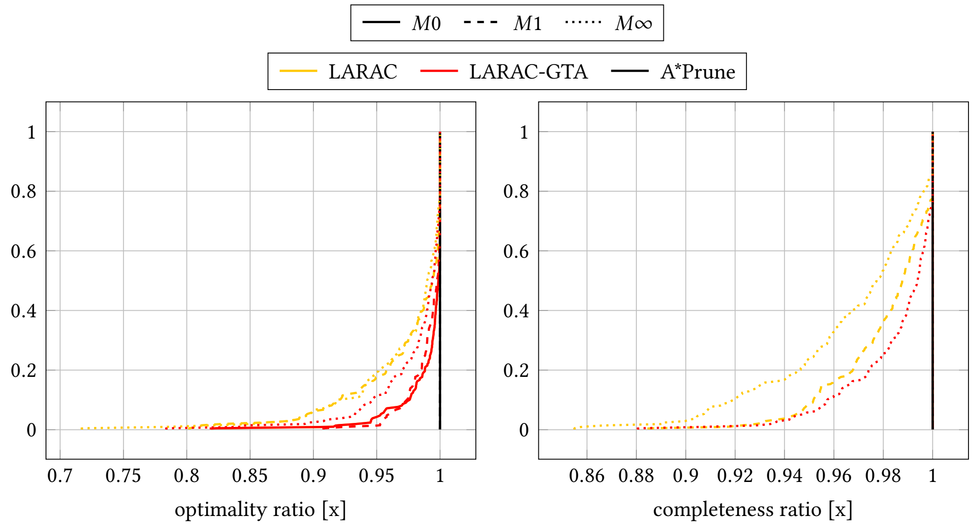


Figure 3.28: Optimality and completeness ratios of LARAC, LARAC with one GTA transformation (LARAC-GTA) and A*Prune for the CSP problem with M0, M1 and M∞ global constraint metrics and an M0 optimization metric.

Note that the runtime increase for each algorithm for the M1 and M∞ metrics is mostly due to the increased complexity of the metric values computation.

Optimality ratio results. The right plot of Fig. 3.27 shows the ECDF of the *optimality ratio* observed for each algorithm and metric types. The optimality ratio is defined per topology as the percentage of requests that the algorithm was able to solve optimally. As expected, both A* and A*-GTA are always optimal for M0 metrics and show a sub-optimal behavior for M∞ metrics. However, while A* presents a sub-optimal behavior for M1 metrics, A*-GTA does not. The sub-optimality of A* for M1 metrics corresponds to the behavior described in Sec. 3.6.1 and Fig. 3.21. This confirms that GTA allows SoA algorithms based on the OSP to keep their original properties for M1 metrics. For M∞ metrics, though GTA does not guarantee optimality for A*, we can observe that it improves its optimality ratio. All algorithms were complete, confirming that an M1 or M∞ global optimization metric does not impact the completeness of algorithms (see Tab. 3.4).

Conclusions. M1 and M∞ global optimization metrics indeed lead to the sub-optimality of A*. While A*Prune provides optimality for any type of metric, it presents a problematic scalability behavior. For its part, GTA allows A* to optimally solve problems for M1 metrics and improves its optimality ratio for M∞ metric, at the price of a reasonable increased runtime. Note that EBD led to an identical optimality behavior and a similar runtime behavior as A*-GTA.

3.6.4.4 Constrained Shortest Path: Completeness Influence

Setup. Because CBF, the optimal CSP algorithm, presents an exponential runtime behavior [Wid94], we use LARAC as sub-optimal but fast and complete CSP algorithm (see Tab. 3.2). We then observe the optimality and completeness of LARAC and LARAC with GTA applied once (referred to as LARAC-GTA) for M0, M1 and M∞ metrics, using A*Prune as benchmark. Note that, because of the poor scalability of A*Prune and the higher runtime required by a CSP search compared to an SP search, we further reduced the topologies to those with less than 50 nodes and 100 edges.

Metrics and constraint bounds. For the optimization metric, we use an M0 metric. The constraint metric upper bound is randomly distributed among all the possible values (between the minimum and the maximum values).

Runtime results. Because of space constraints and because the runtime impact of GTA appeared to be the same as for the SP problem, we omit the runtime values for the CSP problem.

Optimality ratio results. The left plot of Fig. 3.28 shows the ECDF of the optimality observed for each algorithm. LARAC and LARAC-GTA present exactly the same behavior for the M0 metric (the yellow curve being hidden in Fig. 3.28). As expected, LARAC is not optimal for the M0 metric. For LARAC, the M1 metric reduces its optimality ratio. However, the M∞ metric does not appear to further reduce this optimality ratio. For LARAC-GTA, the M1 metric does not have a big influence on its optimality. However, the M∞ metric appears to further reduce this optimality ratio. It is interesting to notice that this behavior is different for LARAC and LARAC-GTA.

Completeness ratio results. The right plot of Fig. 3.28 shows the ECDF of the *completeness ratio* observed for each algorithm. The completeness ratio is defined per topology as the percentage of requests for which the algorithm was able to find a solution. The same conclusions as for the optimality ratio of the SP simulations can be drawn. Indeed, GTA allows LARAC to be complete for the M1 metric and improves on its completeness ratio for the M∞ metric.

Conclusions. The M1 and M∞ global constraint metrics indeed lead to the incompleteness of LARAC. GTA allows LARAC to completely solve problems for the M1 metric and improves its completeness ratio for the M∞ metric.

3.6.5 Applicability of the Solutions

A*Prune and EBD can only be used for the SP/MCSP and SP problems, respectively. On the other hand, GTA can be applied to any routing algorithm. That is, GTA can be used for any routing problem (unicast, multicast, multipath, etc.) and amount of optimization and constraint metrics as long as a SoA algorithm for the corresponding problem with M0 metrics exists. However, we identify that multipath routing requires small adaptations. Indeed, after a GTA transformation, disjointness on the transformed graph does not guarantee disjointness of the corresponding elements on the original graph and multipath routing algorithms might hence return disjoint paths on the transformed graph which are not disjoint on the original graph. To circumvent this problem, the GTA transformation can be adapted by adding an intermediate *sink edge* to which all the transformed edges corresponding to an identical original edge connect. In this way, if the algorithm finds two paths that use different edges corresponding to the same original edge, it will have to use the same sink edge and will hence conclude that these paths are not disjoint. This additional procedure however again comes at the price of an increased amount of nodes and edges in the transformed graph.

3.6.6 Summary

SoA routing algorithms are optimal and complete when using metrics that satisfy the OSP. However, we have shown that relevant QoS metrics such as delay or buffer consumption do not satisfy this property. Hence, the algorithms lose their optimality and/or completeness. This causes the DetServ architecture to violate latency guarantees as well as to inefficiently use network resources.

In order to still guarantee optimal and complete results, we first proposed a new Mn metric taxonomy for classifying routing metrics based on the amount n of previously traversed edges needed to compute their value at a given edge. Based on this taxonomy, we presented solutions guaranteeing optimality and completeness. First, we presented *A*Prune* [LR01], a SoA algorithm that can deal with any type of Mn and M∞ metric for solving the SP and MCSP problems. Second, we proposed EBD, a newly proposed modification of Dijkstra for solving SP problems with M1 metrics. Finally, because A*Prune and EBD can only be used for particular problems and metric types, we proposed a GTA that allows any SoA algorithm for any routing problem (e.g., unicast, multicast, multi-constrained, etc.) to solve problems with Mn metrics. While A*Prune is the only opportunity for optimally solving a problem with M∞ metric, we have shown that it presents a poor scalability behavior. Besides, on the example of the A* and LARAC SoA algorithms, we have shown that GTA indeed recovers their properties for Mn metrics, at the cost of an increased running time.

3.7 Summary

This chapter provided an in-depth analysis of the routing procedure of a centralized architecture for the provisioning of real-time latency guarantees. Four main contributions were made. First, we comprehensively evaluated available algorithms and identified a small subset of algorithms as the best performing ones in most cases. The identified subset of algorithms shares a common particularity: they all rely on subsequent runs of a SP subroutine. Based on this observation, our second contribution consists in the design of an optimization of these subsequent runs by sharing information from one run to the next one. We showed that algorithms can benefit from this change without impacting the result of the routing procedure. Our third contribution extends the investigated algorithms to visit intermediate nodes while still providing latency guarantees. This routing problem is of particular interest in SFC environments. Our fourth and final contribution of this chapter unveiled a major impact the DetServ architecture and model has on the routing procedure: because link metrics provided by the TBM do not satisfy the OSP, SoA algorithms lose their completeness and/or optimality property. We investigated in which exact cases this happens and proposed solutions for recovering these two critical features of routing algorithms.

We have now a clear logic for all the components of the architecture defined in chapter 2. In the next two chapters, we fill the gap from theory to practice by deploying the DetServ architecture and logic on real hardware. We will see that, both for data center (chapter 4) and small networks (chapter 5), existing hardware requires the logic to be adapted. In particular, small networks require a redesign of the network models to accommodate the lower processing speed and throughput of forwarding devices and data center networks require a rethinking of the architecture for the configuration of the forwarding behavior of switches.

Chapter 4

Measurements and Testbed Implementation for Data Center Networks

Datacenter networks have become a critical infrastructure of our digital society. With the popularity of data-centric applications (e.g., related to business, health, entertainment and social networking) and machine learning, the importance of realizing communication networks that meet stringent dependability requirements will likely increase further in the next years. Already today, the usefulness of many distributed cloud applications, such as web search and online retail [Jal+13]; [DeC+07], critically depends on the performance of the underlying network [MP12], i.e., these applications are sensitive to both packet delay and available network bandwidth [Jan+15].

However, providing predictable network latency and throughput to cloud applications is challenging, especially in multi-tenant datacenters and under dynamic demands that come with uncertainty. In many scenarios, the predictability objective even seems to conflict with efficiency requirements, as the latter forbids conservative resource provisioning.

Previous chapters have described an architecture and optimization strategies for the provisioning of predictable latency in programmable networks. However, while these solutions are promising, the predictability of a network, whether software-defined or not, is at most as good as the predictability of its data plane. In fact, even seemingly simple tasks, such as forwarding, involve many complex components, such as link buffers, hardware memory units, switch CPUs, queuing disciplines, etc. A deep understanding of the behavior of all these components is necessary for guaranteeing predictable network operations. Existing mathematical models for per-packet latency guarantees [Jan+15]; [Gro+15]; [KCL14] and the developments of previous chapters rely on an expected standard behavior, i.e., model, of forwarding hardware, the correctness of which cannot be verified by packet-level simulations or E2E measurements.

The first contribution of this chapter is motivated by the following fact: the predictability of communication networks, both in terms of correctness and performance, critically depends on the underlying hardware, and especially the network devices used to process and forward packets. In order to shed light on the predictability of these network devices, we present an extensive measurement study of the behavior of SDN switches. In particular, we systematically benchmark seven SoA

SDN switches from four vendors in order to analyze the predictability of their behavior with respect to important metrics such as processing time and throughput, as well as with respect to the mechanisms used to ensure QoS (such as PQ). We also examine management aspects, e.g., related to flow tables and queues (packet buffers). Our findings are rather negative: none of the examined switches can directly be used with the aforementioned models in order to provide predictable latency. We find that not all switches support line rate forwarding as promised, we identify *aging effects* (some switches suffer from reduced table size over time), we observe that some switches blindly drop forwarding rules, that PQ comes with a processing time overhead neglected so far, or that packet buffers are not isolated per port or per queue, as assumed by traditional mathematical models (e.g., DNC) for buffer dimensioning. We also contribute a new measurement methodology for determining the throughput of programmable devices.

Based on our insights, the second part of this chapter investigates solutions to overcome the observed predictability limitations of existing hardware. The second contribution of this chapter is the design, implementation, and evaluation of *Chameleon*, a cloud network providing both predictable latency and high network utilization, typically two conflicting goals, especially in multi-tenant datacenters. The approach is based on end-host networking and source routing to circumvent the unpredictability issues revealed in the first part of the chapter. *Chameleon* can be deployed at the edge only; it does not require any modifications of network devices. High utilization is reached by further leveraging source routing to easily reconfigure flow priorities and path at runtime. We implement and extensively evaluate *Chameleon* in simulations and a real testbed. Compared to the SoA, we find that *Chameleon* can admit and embed significantly, i.e., up to 7 times more flows, improving network utilization while meeting strict latency guarantees.

Content and outline of this chapter. Sec. 4.1, based on content from [Van+19a], presents an empirical study of the predictability of SDN switches and discusses the impact on existing predictable latency solutions. Sec. 4.2, based on content from [Van+20], then describes the design, implementation, and evaluation of a complete proof-of-concept implementation of a data center network providing predictable latency to applications and overcoming the revealed unpredictability issues. Finally, Sec. 4.3 concludes and summarizes the contributions of the chapter.

4.1 Predictability Study of SDN Switches

This section presents an empirical study of the predictability of SDN switches. Our extensive benchmarking of seven hardware OF switches from four different manufacturers raises several concerns regarding the dependability of these switches. We uncover several incorrect and unpredictable behaviors and performance issues. In particular, we identify unpredictable behaviors related to the management of flows and buffers, and observe that existing QoS mechanisms, such as PQ, introduce unexpected overheads. The latter, in turn, can lead to violations of latency guarantees.

As a contribution to the research community and to ensure reproducibility, all the data sets, source code, and configuration files associated to the results presented in this section are publicly available online [Van19b].

Switch	PT *4.1.2.1*	PQ *4.1.2.2*	TP *4.1.2.3*	FM *4.1.3.1*	BM *4.1.3.2*
HP E3800	+	-	-	-	
HP 2920	+	-	-	-	
Dell S3048-ON	+	∼	+	-	-
Dell S4048-ON	+	∼	+	-	-
Pica8 P3290	+	∼	+	-	-
Pica8 P3297	+	∼	+	-	-
NEC PF5240	+	∼	+	-	∼

Table 4.1: Five predictability dimensions (here: regarding latency) of forwarding devices and whether they are verified for the seven switches. Green (+) means a switch behaves as expected/predicted, orange (∼) means a switch partially behaves as expected, red (-) means a switch does not behave as expected and gray means that it is not applicable because of another unmet requirement. None of the switches are predictable along all the five selected metrics.

This section is organized as follows. We present the related work in Sec. 4.1.1. The performance and management predictability measurements and results are presented in Sec. 4.1.2 and Sec. 4.1.3, respectively. We give some insights and discussions over these results in Sec. 4.1.4.

4.1.1 Related Work

Our work builds on a rich literature on switch performance measurements.

From the management and control plane point of view, studies in the recent years [KPK14]; [KPK15]; [Kuź+18]; [He+15a]; [BR13]; [Laz+14]; [HYS13]; [Rot+12]; [He+15b] have already shown that the states of control and data plane of certain switches can diverge. For instance, inserting a rule is not atomic, i.e., it might still take time for a rule to be inserted in hardware, even after having received a confirmation of the insertion from the switch. Other studies have shown that the flow table capacity of switches varies drastically among vendors [Bau+18]. Although we also cover similar flow management aspects, we provide new insights with a focus on predictability. For instance we show that besides not being atomic, rule insertion can even be ignored by some switches, thereby leaving the data plane configuration permanently inconsistent with the control plane. We further show that certain switches exhibit aging effects reducing their table size over time and thereby making it unpredictable. Similarly, while some studies have measured switch buffer sizes [Bau+18] or revealed the importance of buffer management strategies for latency-sensitive applications, our work sheds light on how switches actually manage their buffer: most architectures are based on a shared buffer dynamically allocated to queues or ports.

Numerous works have also provided insights into the data plane performance of programmable switches [HYS13]; [PMK13]; [Bia+10]; [Jar+11]; [Nao+08]; [Emm+14]; [GYG13]; [Lin+18]; [Bau+18]; [Van+19b]. For instance, [HYS13]; [Bau+18]; [Lin+18] revealed important latency, throughput and buffer size metrics in particular scenarios. Our work focuses more on predictability by investigating the same metrics but evaluating them in variable scenarios. Loko [Van+19b] also focuses on predictability but derives a completely new model for a low-cost switch for which the SoA models investigated here are not valid [Jan+15]; [Gro+15]; [KCL14]; [GVK17]. Software implementations

Switch	**ASIC**	**CPU**	**Firmware** *(release date)*	**Ports**
HP E3800	HPE ProVision	Freescale P2020	KA.16.04.0016 *(2018-06-22)*	48×1G-RJ45 + 4×10G-SFP+
HP 2920	HPE ProVision	Tri Core ARM1176	WB.16.08.0001 *(2018-11-28)*	24×1G-RJ45
Dell S3048-ON	Broadcom StrataXGS	*undisclosed*	DellOS 9.14 *(2018-07-13)*	48×1G-RJ45 + 4×10G-SFP+
Dell S4048-ON	*undisclosed*	*undisclosed*	DellOS 9.14 *(2018-07-13)*	48×10G-SFP+ + 6×40G-QSFP+
Pica8 P3290	Broadcom Firebolt 3	Freescale MPC8541CDS	PicOS 2.10.2 *(2018-01-19)*	48×1G-RJ45 + 4×10G-SFP+
Pica8 P3297	Broadcom Triumph 2	Freescale P2020	PicOS 2.11.19 *(2019-02-27)*	48×1G-RJ45 + 4×10G-SFP+
NEC PF5240	*undisclosed*	*undisclosed*	OS-F3PA 6.0.0.0 *(2014-06)*	48×1G-RJ45 + 4×10G-SFP+

Table 4.2: Specifications of the investigated switches: names, application-specific integrated circuit (ASIC), CPU, firmware and ports.

have also been investigated [DDC18]; [MP18]; [Laz+14]; [Rah+16]; [Gal+15]. However, as suggested by these works, our measurements confirm that OS-based software processing in the CPU is not a viable solution for predictable performance.

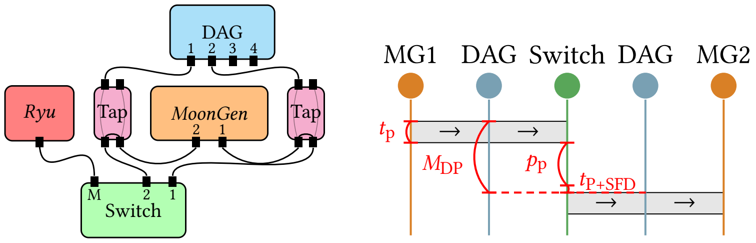
Regarding PQ, Durner et al. [DBK15] conducted an interesting measurement study on its impact on network performance, however, with a focus on flow-level aspects while our analysis focuses on per-packet delays for assessing the predictability of priority schedulers with respect to the latency of individual data plane packets. Some of our presented results also show that previous studies [Bau+18] contain even incorrect data, mostly due to device misconfiguration.

4.1.2 Measurement Study: Performance Predictability

This section (Sec. 4.1.2) and the next section (Sec. 4.1.3) report on our predictability analysis of different switches. Whereas this section focuses on forwarding performance, the correctness of assumptions with respect to management tasks is analyzed in Sec. 4.1.3.

Tab. 4.2 lists the seven investigated switches, representing a wide range of devices: 4 different vendors, different switches per vendors; both SDN-tailored (e.g., Pica8) and general (e.g., HP) switches; both 1G and 10G devices and both high-end (e.g., Dell) and lower-end switches (e.g., NEC and Pica8). The challenge in benchmarking switches is that information about their internal functioning, i.e., what exact components are traversed by packets when they are forwarded, is not publicly available. Manufacturers are reluctant to open their architecture, as illustrated by the fact that we sometimes do not even know the ASIC or CPU model of a switch (Tab. 4.2). Besides, when manufacturers actually describe internals, we will see that such documentation can be erroneous or outdated (Sec. 4.1.3.2). This suggests that switches have to be considered blackboxes for our study.

The results of the empirical predictability study of this section are summarized in Tab. 4.1, structured into the different requirements and assumptions made: whereas some switches like the NEC support more assumptions, *none of the investigated switches exhibit predictable behavior along all the investigated dimensions.* The requirements, discussed and justified in separate subsections within this (Sec. 4.1.2) and the next section (Sec. 4.1.3), are selected based on the assumptions made by SoA E2E strict latency models regarding the behavior of forwarding devices [Jan+15]; [Gro+15]; [KCL14]; [GVK17].



(a) Processing time measurement setup.

(b) Processing time sequence diagram.

Figure 4.1: (a) Switch processing time measurement setup and (b) corresponding sequence diagram.

Throughout the section, we use OF v1.0 for configuring the switches because it supports all the features required to deploy the SoA latency models [Jan+15]; [Gro+15]; [KCL14]; [GVK17], including those described in chapter 2.

4.1.2.1 *PT* — Processing Time

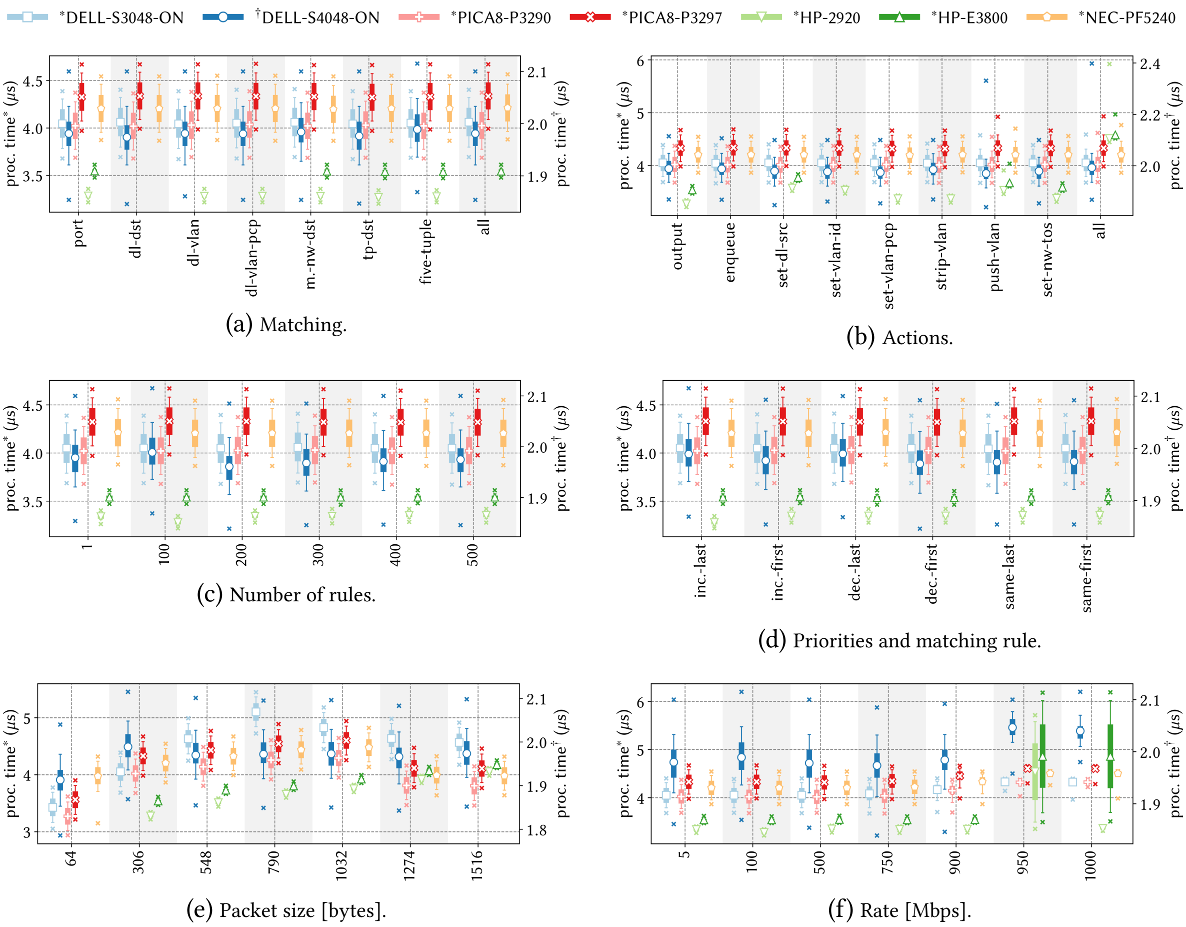
End-to-end delay is the sum of propagation, transmission, processing, and queuing delay. The propagation and transmission delays are physical values directly computed from the physical link properties. The processing and queuing delays are the critical components: they are determined by switch-internal functions. While queuing delay is computed using mathematical models (see Sec. 4.1.2.2), SoA approaches assume that the processing time of the switches is deterministically bounded by a *constant* value [Gro+15]; [GVK17].

We evaluate the processing time of our switches in different settings to assess whether it is indeed bounded by a predictable value for different modes of operation. We vary the *matching* and *actions* properties of rules, their *number*, their *priority*, the *matching rule*, the *rate* of (data plane) packets and the *packet size*. These considered dimensions and their respective values are shown in Tab. 4.3. For the *matching* values, combination of fields are also considered: *five-tuple* includes layer-3 (L3)/L4 source/destination and L3 protocol; *all* includes layer-2 (L2)/L3/L4 source/destination, L3 type of service (ToS) and L3 protocol. For each *action* type, an additional *output* action is included. The *all* action consists of *output*, *set-dl-src*, *push-vlan*, *set-vlan-id*, *set-vlan-pcp*, and *set-nw-tos* (as shown later, L3/L4 modifications are never realized in hardware).

Measurement setup. A *Ryu*-based [Ryu17] controller generates a flow table according to the selected dimension values (Fig. 4.1). The matching flow entry is configured to be forwarded to port 2 of the switch. We use *MoonGen* [Emm+15] to generate packets with the appropriate header fields, packet size and rate. Packets arriving (port 1) and leaving (port 2) the switch are mirrored using network taps to a nanosecond-precise Endace data acquisition and generation (DAG) 7.5G4 measurement card. The card timestamps packets upon arrival of the start frame delimiter (SFD) [Don02]. The processing time p_{P} of a packet p can be obtained by

$$p_{\mathrm{P}} = M_{\mathrm{DP}} - t_p - t_{\mathrm{P+SFD}}, \tag{4.1}$$

where M_{DP} is the measured latency, t_p is the computed packet transmission time, and $t_{\mathrm{P+SFD}}$ is the computed transmission time of the Ethernet preamble and SFD (8 bytes) (Fig. 4.1b).



(a) Matching.

(b) Actions.

(c) Number of rules.

(d) Priorities and matching rule.

(e) Packet size [bytes].

(f) Rate [Mbps].

Figure 4.2: Impact of the different considered dimensions on the hardware processing time of the different switches. The number of rules, their priority, the matching rule and its matching structure do not impact the processing time. The packet size is the main influential factor, while rate and action type impact only a subset of the investigated switches. Overall, the processing time of switches is predictable and can be deterministically bounded by a constant.

Results. While investigating a specific dimension, the other dimensions are kept constant with the default values in Tab. 4.3. The boxplots in Fig. 4.2 show the measured processing times p_{p} for 100,000 packets per scenario. Note that, because of its lower processing time, a different scale is used for the Dell S4048-ON switch. We first consider only hardware rules and cover software rules later.

General. Fig. 4.2 shows that switches have similar processing times (around 4 μs), except the Dell S4048-ON (around 2 μs). Also, for a given case, the variance in processing time exhibited by the switches is always the same (around 0.2 μs for the Dell S4048-ON and 0.6 μs for the others).

Matching. In Fig. 4.2a, it can be seen that the complexity of the matching structure does not affect the processing time of the switches. The HP 2920 does not support matching in hardware for *all* and

Dimension	Values
num. of entries	1, **100**, 200, 300, 400, 500
match type	*port, dl-dst, dl-vlan, dl-vlan-pcp, masked-nw-dst, tp-dst,* ***five-tuple****, all*
action	***output****, enqueue, set-dl-src, set-vlan-id, set-vlan-pcp, strip-vlan, push-vlan, set-nw-src, set-nw-tos, set-tp-src, all*
matching rule	*first,* ***last***
priorities	***increasing****, decreasing, same*
packet size [bytes]	64, **306**, 548, 790, 1032, 1274, 1516
rate [Mbps]	5, **100**, 500, 750, 900, 950, 1000

Table 4.3: Dimensions (and their values) considered for the processing time measurements. Bold values correspond to the default values.

dl-dst. Also, the HP E3800 does not support *dl-dst* in hardware, and interestingly, does not support *dl-vlan-pcp* and *dl-vlan* matchings at all[1].

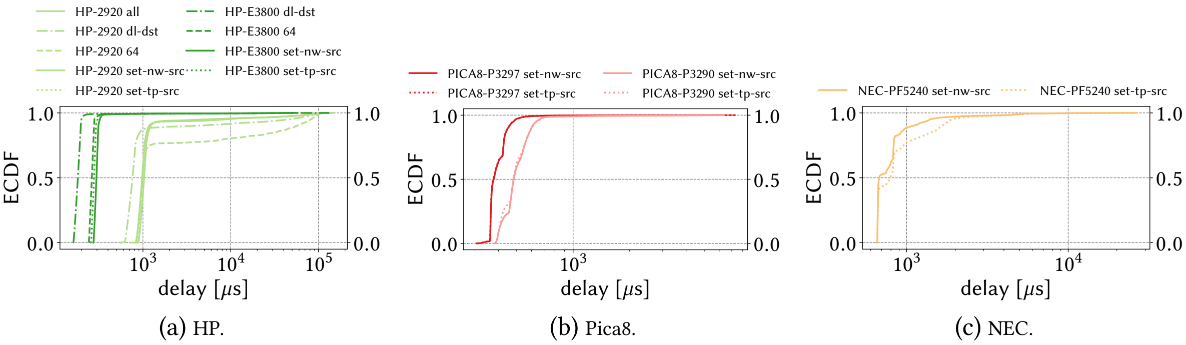
Actions. Similar to the matching, the *action* types do not affect the processing time of the switches (see Fig. 4.2b). This behavior remains true, even when packet rewriting and checksum recomputations are involved. Again, we omit the unsupported actions (i.e., *strip-vlan*, *set-vlan-pcp* and *set-vlan-id* for the HP E3800 and *set-nw-src* and *set-tp-src* for all switches). For the *push-vlan* action case, though the processing time minimum, average and median values are the same as for the other actions, we observe that its maximum processing time is slightly higher. Similarly, this behavior is observed for the *all* action (as it includes *push-vlan*).

Number, priority, and order of rules. Fig. 4.2c and 4.2d show that the number, priority and order of rules do not impact the switch processing time.

Packet size. As can be seen in Fig. 4.2e, the Pica8 P3290, Pica8 P3297, Dell S3048-ON, and NEC PF5240 switches show almost the same behavior: the switch processing time increases with the packet size up to a certain threshold (e.g., 790 bytes for the Dell S3048-ON), thereafter, it starts to decrease (e.g., from 1032–1516 for the Dell S3048-ON). This is surprising when compared to existing literature results, that additionally consider transmission latency [Bau+18] when measuring processing time. Due to missing insights on the exact details of the switch architecture and its ASIC, we can only speculate on the reasons which looked most reasonable to us. As these switches mainly use Broadcom chips, we guess that this is indeed an ASIC-dependent behavior[2]. Such a behavior can be due to the traffic manager implementation, since this ASIC module is usually responsible for buffering packets before sending them. Specifically, to achieve high utilization and efficiency, the traffic manager typically waits until a certain number of cells of data are filled to continue processing. While this can present some explanations for the increasing trend, the decreasing trend remains unclear. The two HP switches show a mostly (actually piecewise) linear behavior. We suspect this is because of the specific way bytes are buffered in the HPE ASIC. In contrast, the processing time of the Dell S4048-ON is mostly constant except for smaller packets, where it is lower.

[1] We note that the HP E3800 documentation states that *dl-vlan* and *dl-vlan-pcp* matchings are supported. However, with a configuration identical to the one used for HP 2920, the switch never successfully matched on these fields. We tried to contact the HP support but unfortunately did not receive any reply. Thus, we consider these fields as *not supported*.

[2] We have tried to contact Broadcom to obtain explanations on this behavior, however without any success.



(a) HP. (b) Pica8. (c) NEC.

Figure 4.3: Processing time of software rules for different vendors. Note that Dell does not support any software rules. We observe that software processing is much less predictable than hardware processing. Software processing is also orders of magnitude slower.

Rate. Fig. 4.2f indicates that the rate also does not influence processing time of the switches, except for very high rates. In these cases, both HP switches see their overall processing time increasing, while other switches only see their minimum processing time increased.

Software rules. In this part, we present the results of measuring the processing time of software rules (excluding Dell, because they do not support any software processing capabilities). Because packet loss is observed for higher rates, we use a rate of 0.1 Mbps for this experiment. We point out three main observations (Fig. 4.3). First, none of the switches support L3/L4 rewriting in hardware. The HP switches additionally do not support L2 matching in hardware. More surprisingly, the HP switches cannot forward 64-byte packets in hardware: even if the rule is stored in hardware and operates properly for bigger packets, we observed that 64- and 65-byte packets are always processed in software. Interestingly, this does not happen when the switch runs in legacy mode or with OF rules matching on the physical input port. We suspect that this is due to the fact that the HP switches use a different TCAM table for OF processing with complex matching, which might not be able to process packets of these sizes. Second, software processing time is up to 5 orders of magnitude higher than in hardware. Third, processing time in software is much less predictable: it varies by nearly two orders of magnitude compared to the hardware case (see Fig. 4.2). This is due to the fact that the switch CPU is also performing other interfering tasks (e.g., running its OS and the OF agent).

Outcomes. The processing time of switches mostly depends on packet size. Moreover, the *action*, *matching*, *priorities*, and the *number* of rules do not influence processing time. The processing time of switches can be considered predictable and bounded only for hardware rules.

4.1.2.2 *PQ* — Priority Queuing Overhead

Predictable latency works consider the availability of PQ [Jan+15]; [Gro+15]; [KCL14]; [GVK17]. Queuing delay is then computed using mathematical models, e.g., DNC [LT12]; [VK16].

We explore whether our switches support PQ and if their behavior can be verified by DNC models. Regarding the first question, we found that only the two HP switches do not support PQ in OF mode (i.e., the *enqueue* action).

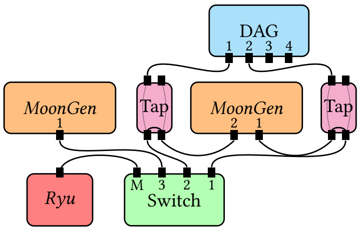


Figure 4.4: Measurement setup for PQ investigation.

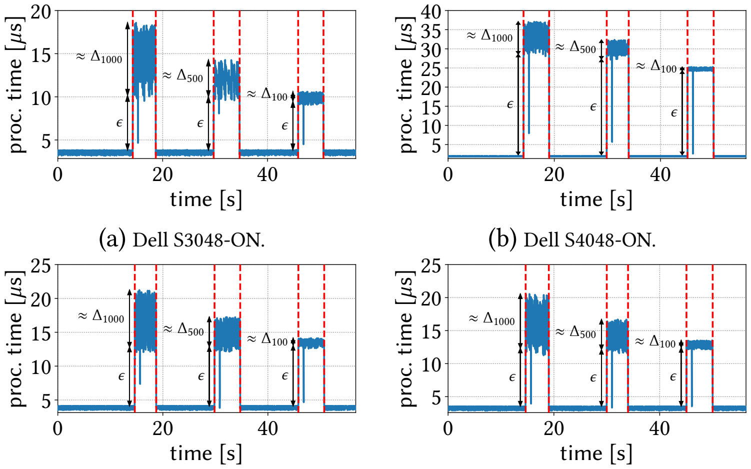
Measurement setup. Two *MoonGen* instances send low and high priority flows to the switch through ports 1 and 3, respectively (Fig. 4.4). The flows are forwarded to port 2. We start a high priority flow sending bursts of 100-byte packets. Then, in order to quantify the overhead of PQ, we subsequently send three low priority flows with different packet sizes (1000, 500, and 100 bytes) also at line rate. The switch processing time is then measured using the measurement card and a setup identical to Sec. 4.1.2.1.

Deterministic network calculus prediction. DNC states that processing of a high priority (H) packet can be delayed by a lower priority (L) packet by at most the transmission time of the largest packet in lower priority queues (thereby considering the non-preemptive property of priority schedulers). Therefore, DNC calculates the worst-case delay bound D_H experienced by a high priority flow as

$$D_\mathrm{H} = p_\mathrm{p} + \Delta_{l_\mathrm{L}} = p_\mathrm{p} + l_\mathrm{L}/R, \tag{4.2}$$

where p_p is the pure processing time of packet p (as measured in Sec. 4.1.2.1), the parameter Δ_{l_L} is the transmission time of the largest packet in lower priority queues, l_L is the size of this packet, and R is the link rate [LT12]; [VK16]. In the following, we will show that surprisingly, in practice, processing time can be higher than this mathematical prediction.

Results. In the worst case, we observe a total delay increase of $\Delta_{l_\mathrm{L}} + \epsilon$ (Fig. 4.5). That is, there is a delay increase of ϵ in addition to the increase predicted by DNC, independent of the size of the interfering packets. In fact, ϵ is the overhead of the PQ implementation. The reason is that the switch is not able to determine the next queue without spending a minor processing overhead. We confirmed this observation by reducing the rate of the interfering flows. In this case (not shown), the maximum processing time of the high priority flow looks exactly the same but we observe all the intermediate values. This is because, when the rate is lower, it can happen that a high priority packet arrives just before the next check of the switch, hence not having to wait. In our plots, these values are not visible because cross-traffic is sent at line rate; hence, the scheduler always switches back to low priority flows. All switches exhibit this behavior. Further measurements, not shown here, show that the ϵ value depends on the switch model but stays constant for different scenarios (different high priority packet sizes, additional concurrent lower priority flows). From our measurements, we have 9 μs for both Pica8 switches and the NEC PF5240, 6 μs for the Dell S3048-ON, and 27 μs for the Dell S4048-ON. The relatively low overhead and its stability tells us that the scheduling operation is not performed by the central CPU of the switch but by a specific component responsible for this, e.g., a micro-controller.



(a) Dell S3048-ON.

(b) Dell S4048-ON.

(c) NEC PF5240.

(d) Pica8 P3290.

Figure 4.5: Processing time of a high priority flow (bursts of 100-byte packets) with 3 subsequent interfering low priority flows of, respectively, 1000, 500 and 100-byte packets. DNC predicts an increase of Δ_{l_L} in processing delay. We observe that the processing time increases more (by ϵ) than what DNC predicts. We note that the behavior of the Pica8 P3297 switch is very similar to the Pica8 P3290 and hence not shown.

Outcomes. While the two HP switches do not support PQ, all the other switches implement PQ by updating the queue to send from only every ϵ microseconds. This invalidates DNC predictions (Eqn. 4.2). However, once the modeling correction is done, the performance of the switches is stable and predictable.

4.1.2.3 *TP* — Line Rate Throughput

Queuing models used by SoA approaches [Jan+15]; [Gro+15]; [KCL14]; [GVK17] assume that queuing does not happen at the ingress of switches but at their egress. Accordingly, switches need to be fast enough to process packets at line rate on all ports simultaneously in both directions.

We verify that switches indeed can process packets at line rate in both directions on all their ports simultaneously without any loss. Interestingly, most existing works measure only the throughput of a single port of a switch [Bau+18]; [PMK13]; [Bia+10]; [Emm+14]; [Rau+16]; the complete saturation of a switch backplane has not yet been targeted in the literature.

Setup: the shoelace measurement. The traditional approach to measure throughput is quite simple: send a high load to a switch and measure the output load from it. In practice, however, to saturate the switch is a challenging task: a 48-port switch demands producing a rate of up to 48 Gbps, which, with a simple approach of connecting each port to a traffic source, would require 48 servers or networking cards. As a result, researchers then simply fallback to measuring the throughput of a single pair of ports [Bau+18]; [PMK13]; [Bia+10]; [Emm+14]; [Rau+16]. However, such a measurement does not verify that a switch is able to process packets at line rate if several ports are used simultaneously, hence not guaranteeing predictability.

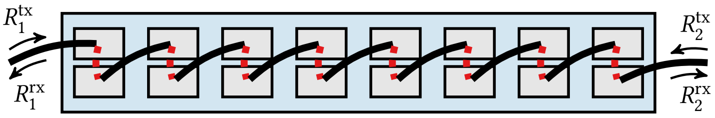


Figure 4.6: The shoelace measurement setup for the measurement of the throughput of a programmable switch. Black thick lines represent cables and red thick dashed lines represent internal forwarding rules. This setup allows to saturate all the ports of a switch with only two traffic sources.

To circumvent this issue, we propose the *shoelace measurement*[3] setup (Fig. 4.6), a methodology that makes use of the programmability of switches to saturate a switch with only *two traffic sources* — only two physical network connections instead of, e.g., 48, are needed.

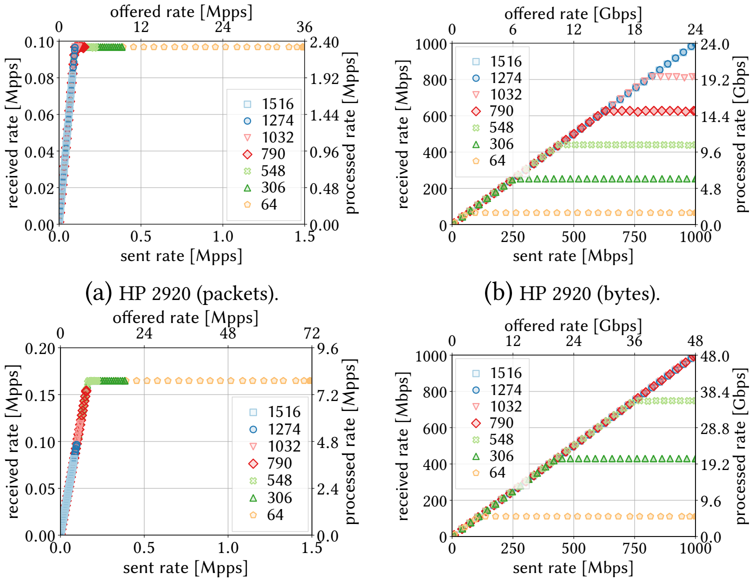
The first source saturates the first port of the switch. The switch is configured to forward all traffic entering this first port to its second one. The second port of the switch is then connected back to its third port. This port configuration and connection then goes on until it reaches the last port of the switch. In this way, all the traffic sent to the first port has to be processed n times, where n is the number of ports of the switch. Further, by having the second source sending traffic to the last port of the switch, and configuring backward forwarding rules, we effectively saturate the n ports of the switch in both directions. We then can compute the (minimum) throughput of the switch as $n \times (R_1^{\text{rx}} + R_2^{\text{rx}})$, where R_1^{rx} and R_2^{rx} are the rates received at the first and last ports, respectively. If these values are equal to R_2^{tx} and R_1^{tx}, we could not reach the throughput limit of the switch and the switch is able to process bi-directional line rate on all its ports simultaneously. For a given input rate $R_1^{\text{tx}} = R_2^{\text{tx}}$, we run the experiment five seconds and consider that a switch is able to handle a given rate only if no single packet is lost.

We use the shoelace measurement setup to investigate the throughput of all our switches for all the packet sizes in Tab. 4.3. Using *MoonGen* [Emm+15], we generate line rate traffic on both the first and last ports of the switch. We use the statistics of the two interfaces to detect packet loss. We consider rates from 8 to 1000 Mbps by steps of 32 Mbps. For the Dell S4048-ON, we also run the experiment with 10 Gbps links, ranging from 80 Mbps to 10 Gbps by steps of 320 Mbps.

Results. Among all the switches, only the HP switches lost packets. For the smallest packet size (64 bytes), sending 1 Gbps (resp. 10 Gbps) corresponds to 1.5 Mpps (resp. 15 Mpps). That is, for our 48-port switches, all but the HP switches can process at least $48 \times 1.5 = 72$ Mpps (resp. 720 Mpps) simultaneously. This is confirmed by the datasheets of the switches, which all announce values higher than this.

For the HP switches, the datasheets also announce that the switches can process line rate on all their ports. However, this is not the case (see Fig. 4.7). We observe that, independently of the packet size, the switches can process up to 2.32 Mpps and 7.91 Mpps. Depending on the packet size, this leads to different data rates. The HP 2920 can process line rate for packets of at least 1274 bytes and the HP 3800 for packets of at least 790 bytes. This stands in contradiction to the datasheets of the switches. In order to investigate this further, we conduct a measurement run with the HP E3800 in legacy mode. As the shoelace measurement setup cannot be used for legacy switches, we only

[3] Taking its name from how the cabling of the switch looks like in this setup.



(a) HP 2920 (packets).

(b) HP 2920 (bytes).

(c) HP E3800 (packets).

(d) HP E3800 (bytes).

Figure 4.7: Throughput of the HP switches. Lower (resp. left) axes correspond to the traffic sent (resp. received) on the first and last ports, i.e., R_1^{tx} and R_2^{tx} (resp. R_1^{rx} and R_2^{rx}). The upper (resp. right) axes correspond to the values scaled by the number n of ports to represent the total data rate actually processed by the switch.

measure the throughput on two ports. We use the 10G SFP+ ports to be able to reach more than 7.91 Mpps. In such a setup, the HP E3800 switch processes correctly 10 Gbps at line rate, i.e., from 0.8 Mpps to 14.8 Mpps depending on the packet size. Using a bidirectional measurement, the switch is also able to process all packets, but the network card (Intel 82599ES) used was only able to generate up to $9.2 \times 2 = 18.4$ Mpps for 64-byte packets. We conclude that the throughput that the switch can handle depends on whether it operates in OF or legacy mode.

Our explanation here is as follows: the L2/L3 TCAM tables used for legacy switching/routing cannot store arbitrary matching fields, which means they cannot support the OF features. To support OF, the HP switches use their traditional so-called "access control list (ACL)" TCAM tables, which provide higher flexibility. Hence, in OF mode, we measure the throughput of the "ACL" table, while in legacy mode we measure the throughput of the L2/L3 tables. It turns out that the "ACL" table of the switches was not dimensioned for handling line rate, which hence impacts the performance of the switch when used in OF mode[4].

Note that we also investigate unidirectional throughput by sending data only on the first port of the switch. The obtained values were exactly the half of those obtained for bidirectional testing, showing that using ports in a single direction or in both does not impact the achievable throughput.

Outcomes. Using our proposed shoelace setup, we observed that only the HP switches do not behave as expected and are hence not predictable, as they occasionally can lose packets. They cannot process packets at line rate (Tab. 4.4), even though they can in legacy mode: the TCAM table used for OF and legacy modes are different and exhibit different throughputs.

[4] Note that, here, 64-byte packets are still processed in hardware, while they were processed in software in Sec. 4.1.2.1. This is because we are here matching on physical port while we used *five-tuple* matching in Sec. 4.1.2.1.

Switch	**[pps]**	**[bps]** (64 – 1516 bytes)
HP E3800	7.91 Mpps	5.31 Gbps – *line rate*
HP 2920	2.32 Mpps	1.56 Gbps – *line rate*
Dell S3048-ON	≥ 72 Mpps	*line rate – line rate*
*Dell S4048-ON**	≥ 720 Mpps	*line rate – line rate*
Pica8 P3290	≥ 72 Mpps	*line rate – line rate*
Pica8 P3297	≥ 72 Mpps	*line rate – line rate*
NEC PF5240	≥ 72 Mpps	*line rate – line rate*

Table 4.4: Measured throughput for the different switches. Values in bps are given for the smallest and biggest packets. *line rate* means that the switch handles line rate on all its ports simultaneously.
*measured at 10 Gbps.

4.1.3 Measurement Study: Management Predictability

Next, we consider the management predictability. We analyze two aspects which are relevant for SoA approaches concerning predictable latency: the flow management, in Sec. 4.1.3.1, and the buffer management, in Sec. 4.1.3.2, of switches.

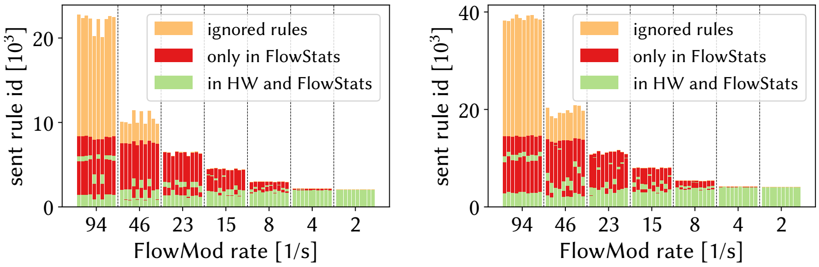
4.1.3.1 *FM* — Flow Management

SoA approaches rely on fine-grained traffic engineering (one rule per single flow) in order to provide their strict guarantees, e.g., deterministic latency [KCL14]; [GVK17].

Therefore, the number of flow rules on a single switch in a network can grow up to several thousands of flows [GVK17]. Hence, the first requirement is a sufficient flow table capacity. As flow requests arrive during runtime, they have to be inserted live in the corresponding hardware tables by the controller. Hence, each switch should have synchronized data and control planes, e.g., it should not state that certain flow rules are embedded if they are actually not. Note that it is known that adding a rule into the hardware table of a switch can cause a wide variety of issues [KPK14]; [KPK15]. For instance, the state of the data plane can lag behind the state of the control plane for a certain amount of time [KPK15]. However, we here only require the switch to add the rule in its hardware table *at some moment*, and we do not consider delay as an issue.

Note that in this section, the OF version in use may have an impact. We, however, stick to OF 1.0 as it provides the necessary features for the SoA E2E latency models and it is fully supported by all switches.

Measurement setup. We connect the target switch to a *Ryu*-based controller and connect a dual-port data plane host running *MoonGen* [Emm+15] to the switch. First, we install rules at a given rate on the switch until the latter returns an OF *Error* message indicating that its flow table is full. We consider the same match and action parameters as listed in Tab. 4.3. The rules' *output* actions direct to the second *MoonGen* interface. Second, the controller queries the state of the switch with *TableStatsRequest*/*FlowStatsRequest* messages. Finally, the *MoonGen* host generates one packet per rule on its first interface and checks if it is received on the second interface, i.e., if it is correctly forwarded.



(a) Pica8 P3290.

(b) Pica8 P3297.

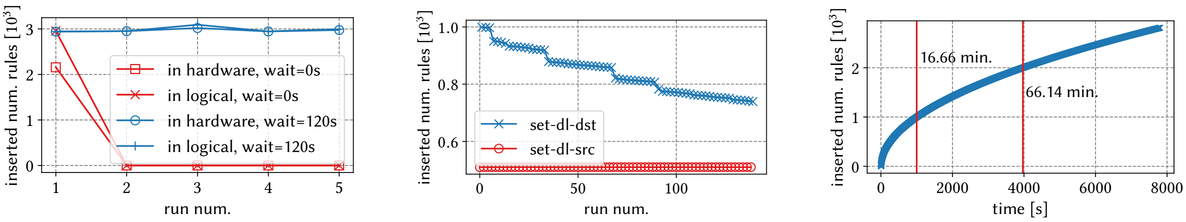
Figure 4.8: Divergence of the flow table state of the Pica8 (a) P3290 and (b) P3297 switches for different *FlowModAdd* rates. For each rate, each stacked bar corresponds to a distinct run. Rules that are only in the *FlowStats* response of the switch are critical: while the controller thinks the packets will be forwarded, they will actually not be. The switch is then unpredictable.

Results. Generally, the results per switch vary; hence, we report on each manufacturer separately.

Pica8 switches. We consider *five-tuple/output* flow rules. The following behavior is the same for other combinations of matching and actions. Fig. 4.8 shows the total number of sent rules before receiving the first *Error* message for different *FlowModAdd* rates and different runs per rate (one bar corresponds to one run). The different colors identify rules that were correctly added in the hardware table, those that were only in the stats reply of the switch (and hence are reported to be added but are actually not) and those that were simply ignored. The total number of rules in hardware is always the same: the Pica8 P3290 stores 2046 rules and the P3297 stores 4094. However, for increasing *FlowModAdd* rates, we observe that the number of ignored and incorrectly added rules increase. While ignored rules are not too problematic, as the controller can react to it, rules that are reported to be added but are not are critical: the controller assumes that packets will be forwarded, while they will not be.

Although both switches can store enough rules, they fail to report a correct state to the controller, which can be dramatic for predictability: packets of a theoretically accepted flow can never be forwarded. We believe that the reason for this is synchronization issues between the open vswitch (OVS) [Pfa+15] instance, which realizes OF on both switches, and the ASIC managing the hardware table. OVS might not be fast enough to insert all the rules, although it previously confirmed them to the controller. This leads to an inconsistency between OVS and what is actually inserted into the hardware table. It is important here to make sense of two aspects. First, the considered rates: inconsistencies appear already for as low as four new flows per second. Second, the correctly installed rules are not the first ones and vary depending on the run. That means that the switches show here a completely unpredictable behavior: we cannot know in advance which rules will be correctly added, except if we reduce the addition rate to a single flow per second, which is not feasible. We believe that such observations are not only relevant for operators, but also researchers when experimenting with these switches.

HP switches. Fig. 4.10a shows the hardware table size of the HP E3800 switch for the different match/action combinations. A size of zero (white) indicates that a certain match and action combination is not supported. Depending on the match and action type of the embedded rules, the table



(a) HP E3800. (b) Dell S3048-ON. (c) NEC PF5240.

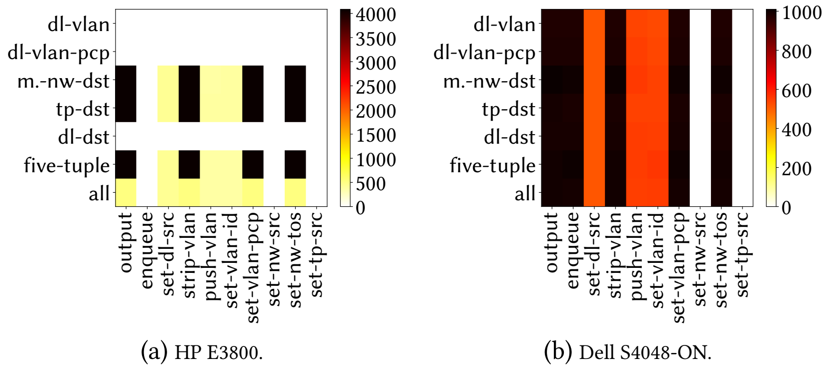
Figure 4.9: (a) HP E3800 switch, *five-tuple-output* as the match-action, with rate of 85 messages per second. (b) Aging of the Dell switches. The measurement procedure is as follows, we use *five-tuple* matching and we perform 200 runs, 100 runs with *set-dl-src* and 100 runs with *set-dl-dst* actions. The runs are performed consecutively, first one runs with *set-dl-src* and then with *set-dl-dst*. We notice that the maximum number of rules for *set-dl-dst* reduces at each iteration. (c) The addition of rules to the NEC PF5240 switch takes a big amount of time.

size varies from 372 to 4085. Due to space reasons, we omit showing the detailed results for the HP 2920; however, it exhibits a similar trend as the HP E3800 but supports only around 100 to 500 flows.

We additionally observed a remarkable behavior: the switches showed aging effects during our measurements. Fig. 4.9a shows the total number of rules in the *FlowStats* answer of the HP E3800 switch and its hardware table over five consecutive runs with 85 *FlowModAdd* messages per second (*five-tuple*/*output* flow rules). For the first set (red lines) of runs, the data plane test is done directly after receiving the *FlowStats* response. For the second set (blue lines), we introduce an additional waiting time of 120 seconds before the start of the date plane test. We notice that sending the *FlowModAdd* messages with a high rate triggers the switch to send the *Error* message earlier, i.e., before the hardware table is actually full. Indeed, the table size for *five-tuple*/*output* with lower rate was 4085 (see Fig. 4.10a) while it is now around 3000 (run 1 in Fig. 4.9a). Furthermore, we observe that the amount of forwarded packets is lower (around 2100 packets) than the number of rules in the logical table. Doing other consecutive runs without waiting, we observe that the switch then rejects any new rule. On the other hand, we observe that, if we wait 120 seconds before the data plane tests, the switch does not show some aging effects (blue lines). This aging leads to an unpredictable behavior from the switch: the controller can never be sure whether it is able to use the complete hardware table space.

Dell switches. Similar to the HP switches, the number of rules which can be stored in the hardware flow table of the Dell switches varies based on the *match*/*action* combination: from 510 rules to the maximum of 1000 rules. Fig. 4.10b shows this for the Dell S4048-ON, the Dell S3048-ON behaving exactly the same. While the Dell devices are able to handle higher *FlowModAdd* rates (e.g., more than 85 *FlowModAdd* messages per second) than the other switches, their hardware tables are the smallest.

However, we noticed again flow table aging effects (shown for the Dell S3048-ON in Fig. 4.9b). We perform 140 consecutive runs measuring the available flow table size. All even runs have *five-tuple* matchings and *set-dl-src* actions, while the odd runs have *five-tuple* matchings and *set-dl-dst* actions. We observe that, for each new iteration, the number of flows that can be added with *five-tuple*/*set-dl-src* combinations stays the same, while it reduces for the runs with *five-tuple*/*set-dl-dst*



(a) HP E3800.

(b) Dell S4048-ON.

Figure 4.10: Flow table size of the (a) HP E3800 and (b) Dell S4048-ON switches for the different match/action combinations. A size of 0 identifies a combination not supported in hardware. We observe that the table size depends on both the matching type and the action and can heavily influence the number of rules that can be inserted.

combinations. The reduction is non-negligible, as the capacity of the flow table reduces from 1000 to 739 rules.

NEC PF5240. The NEC PF5240 switch is the only switch that controls the control plane message rate by throttling the transmission control protocol (TCP) connection. As we cannot control the TCP behavior with Ryu, we modify our procedure. Instead of sending *FlowModAdd* messages with a certain rate, we now send a *BarrierRequest* message after each *FlowModAdd* and wait for the response from the switch. Upon reception of the *BarrierResponse*, a new *FlowModAdd* message is dispatched after 10 milliseconds. Fig. 4.9c shows the number of inserted rules by the switch during one measurement run. By using the modified procedure, adding the rules to NEC PF5240 switch takes a very large amount of time. For instance, adding 1000 rules takes around 16 minutes, and adding 2000 rules takes more than one hour. In total, the switch accepts 2809 rules. Although the size of the table is acceptable, the rules are not inserted in a *timely* manner.

Outcomes. Unfortunately, based on our analysis, none of the switches exhibit a predictable behavior with a sufficiently-sized hardware flow table. Both Dell and HP switches suffer from unpredictable aging, i.e., the flow table size can reduce with each consecutive run. The Pica8 switches can unpredictably and silently ignore rules. For the NEC switch, reaching a high number of rules requires too much time.

4.1.3.2 *BM* — Buffer Management

In order to ensure no packet loss, SoA approaches for predictable latency rely on mathematical models, e.g., DNC, to bound the amount of backlog flows generate at each individual queue on the way to their destination [Gro+15]; [Jan+15]; [KCL14]; [GVK17]. To this end, all these approaches assume that each queue of each port of each switch is equipped with its own physical buffer, and that these buffers are managed independently. To which extent this assumption is true is still an open question. More precisely, there are several challenges still to be tackled: *i)* how do existing switches actually manage their buffers? and *ii)* are these buffers actually isolated? In this part, we intend to study these questions in detail by measuring the buffer capacity of a particular queue in different overload scenarios and assess whether the way the switches manage their buffer is predictable, and, more importantly, as predicted by SoA models.

Measurement setup. We have already seen that the output of switches is not always trustworthy (Sec. 4.1.3.1). Therefore, we design a setup relying only on data plane measurements to measure buffer capacities. Similar to [Bau+18], we infer the buffer size N of a particular queue based on observed packet delays. The total delay observed (through a measurement setup identical to Fig. 4.1a) by a packet from a high priority queue is given by

$$D_{\mathrm{H}} = p_{\mathrm{p}} + \epsilon + q_{\mathrm{p}}, \tag{4.3}$$

where, in addition to the processing time p_{p} (measured in Sec. 4.1.2.1), ϵ corresponds to the PQ overhead (measured in Sec. 4.1.2.2), and q_{p} corresponds to queuing delay. We note that both the ϵ and q_{p} parameters were not present in Sec. 4.1.2.1. This is because these parameters are larger than zero only when queuing happens, which we made sure is not the case for our measurements in Sec. 4.1.2.1. The values of p_{p} and ϵ in Eqn. 4.3 are known from previous sections. The queuing delay q_{p} can then be obtained by measuring the total delay D_{H} and used to calculate N. Indeed, the queuing delay q_{p} of a given high priority packet can be decomposed as

$$q_{\mathrm{p}} = \sum_{i \in P} l_{\mathrm{i}}/R, \tag{4.4}$$

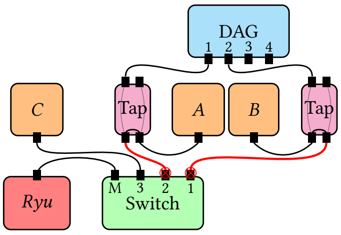
where P is the set of packets scheduled before the considered packet and R is the link rate. If all packets have the same size l, we have

$$q_{\mathrm{p}} = |P| l/R, \tag{4.5}$$

where $|P|$ is the number of packets in the high priority queue when the considered packet arrived. If a port or queue is overloaded[5], it will start queuing packets and eventually drop some of them. When we observe packet loss, we can compute the queuing delay d_{p} of the previous non-dropped packet, from which we can obtain $|P|$. As this packet was the last one to be buffered before dropping packets, we have $N = |P| + 1$, i.e., the buffer capacity corresponds to the number of packets queued before p plus one for p itself. This measurement procedure allows us to monitor only the buffer size available to high priority queues.

We use up to six different ports sending flows to each other at line rate. Each port sends and receives one flow. These flows are forwarded to the highest priority queues. We then use an additional port to send “overload” traffic. For each sending port, the overload port sends additional packets with the same headers, hence overloading the corresponding receiving queue. In order to overload low priority queues, the overload port simply sends additional flows which are forwarded to the corresponding low priority queues. This is sufficient to overload them, as they will never be served, since we send line rate of high priority flows at the same time. We use this setup to overload from 1 to 6 ports and from 1 to 4 or 8 priority queues per port (depending on how many queues the switch under test supports). An example setup to congest 2 ports and 2 queues per port is shown

[5] *overloaded* and *congested* terms are used interchangeably.



(a) Physical setup.

From	To	Priority
A	B	High
B	A	High
C	B	High
C	A	High
C	B	Low
C	A	Low

(b) Flows sent.

Figure 4.11: Setup for evaluating buffer size with 2 congested ports and 2 congested queues per port. The congested ports are marked with a red cross.

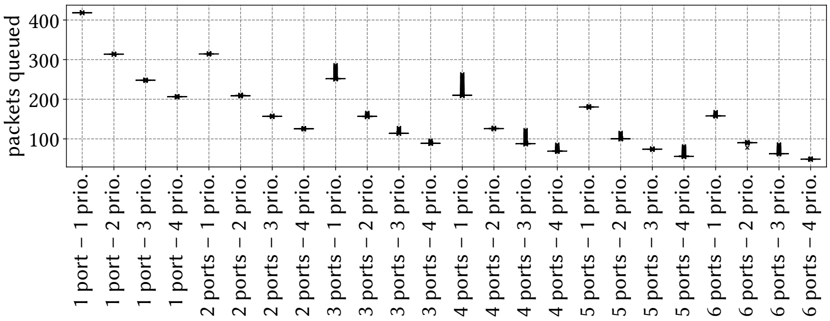


Figure 4.12: Buffer capacity made available to a given queue of the Dell S3048-ON switch for different numbers of congested ports and queues (1516-byte packets): the more ports and queues are overloaded, the less buffer is made available to a given queue.

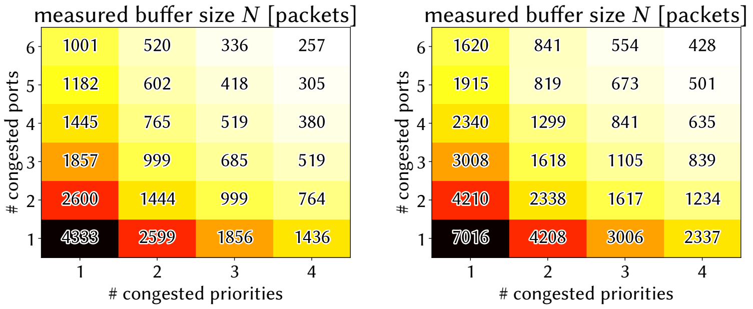
in Fig. 4.11. We monitor packet loss and packet delays of one of the high priority flows using our measurement card. We configure forwarding rules on the switch matching on IP destination and enqueuing in a specific queue. Further, we use different source MAC addresses for each packet to uniquely identify them and easily detect packet loss and compute q_p.

We detail our results for the different manufacturers separately.

HP switches. As the two HP switches do not support PQ (no separate queues), this measurement does not apply to them.

Dell switches. The Dell S3048-ON and S4048-ON switches support up to 4 priority queues. Fig. 4.12 shows the inferred buffer sizes over runs of at least three seconds with 1516-byte packets with the Dell S3048-ON switch, for different numbers of congested ports and priorities. In contrast to what other related works such as QJump [Gro+15], Silo [Jan+15] and DetServ [GVK17] assume, we observe that the buffer made available to our monitored flow depends on the buffer needed by other ports and queues. More specifically, the more ports and queues are congested, the less buffer is made available to our monitored flow. We note that the available buffer size ranges from around 420 packets to around 50 packets.

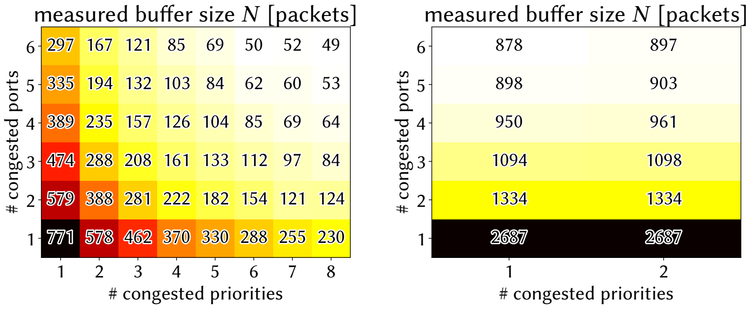
Fig. 4.12 shows that for each run, the inferred buffer size (one value per packet lost) is stable. Hence, in Fig. 4.13, we plot heatmaps of the observed median values for the Dell S4048-ON switch. The results indicate that the available buffer for the S3048-ON switch is bigger than for the S4048-ON switch: from around 10 times bigger for a single congested queue to around 5 times bigger for 6 ports



(a) 1516-byte packets.

(b) 790-byte packets.

Figure 4.13: Median buffer capacity made available to a given queue of the Dell S4048-ON switch for different numbers of congested ports and queues for different packet sizes. The Dell S4048-ON can store more packets than the Dell S3048-ON and the number of packets that can be buffered, as expected, increases with smaller packets.



(a) Pica8 P3297 (the P3290 behaves exactly the same).

(b) NEC PF5240.

Figure 4.14: Pica8 and NEC behave similarly to the Dell switches: buffers are not isolated per-queue (1516-byte packets).

with 4 congested queues. Fig. 4.13b shows that, as expected, reducing the size of transferred packets allows to increase the total number of packets in the queue. The overall behavior however stays the same.

Pica8 switches. Both Pica8 switches presented the exact same behavior, we here show only results for the Pica8 P3297 switch. Fig. 4.14a shows that the behavior is comparable to the Dell switches: the more queues and ports are congested, the less buffer becomes available to the monitored queue. Interestingly, our results invalidate the Pica8 documentation regarding buffer management. Indeed, while they indeed say the queues are based on a shared buffer, the numbers they provide do not correspond to our results.

NEC PF5240. The NEC switch supports up to 8 priority queues. The same experiment with the NEC PF5240 switch exhibits a constant buffer size for each scenario: 63 packets. Hence, the NEC switch seems to have isolated buffers for each queue. However, the NEC switch provides a global *limit-queue-length* option, 64 by default, that limits the maximum packets a queue can buffer. We set this option to its maximum value and rerun the experiment. Unfortunately, this option applies only to the two lower priority queues. As our setup can only measure the buffer size available for

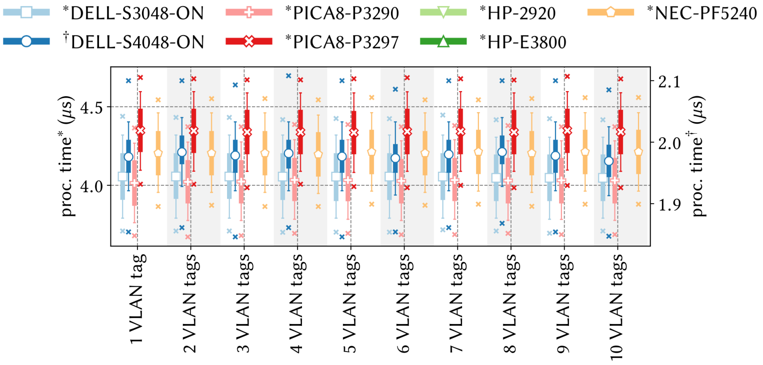


Figure 4.15: Processing time of the different switches for matching on the outermost VLAN tag, popping it and forwarding to a particular queue based on different VLAN tag stacks heights. We observe that the switches can predictably support source routing solutions based on VLAN tags.

the highest priority, we reduce the maximum number of priority levels that can be congested to 2, the two lowest priority queues. Fig. 4.14b shows that the buffer size made available to a queue then depends on the number of congested ports. On the contrary to the Dell and Pica8 switches, it does not seem to depend on the number of congested priorities. Note however that the results here are much less stable than those from Dell in Fig. 4.12. Hence, the numbers should not be taken as exact, but as insights on the buffer management strategy of the switch.

Outcomes. We observe that, while predictable latency solutions do assume that switches provide isolated per-queue buffers, the reality is the opposite: all our switches are based on shared memory for implementing queue buffers. This does actually make sense: sharing buffers among all queues and ports enable work-conservation: if a queue does not use its space, it can be used by another one. However, from a predictability point of view, there is a major issue: a burst in a low priority queue, or even on another port, can suddenly reduce the buffer space available to a given queue and hence potentially lead to unpredictable packet loss.

4.1.4 Insights and Discussions

Let us recap some of our insights on the predictability of SDN switches. On the bright side, the observed processing times of the switches, at least when done in hardware, is fairly predictable. We also observed that most switches successfully process packets at line rate.

On the dark side, we identified several challenges. First, we found that PQ adds non-negligible overhead to processing time. While this is not what is predicted by SoA models (e.g., DNC), we observe that this overhead is actually predictable: the observed ϵ values, which are static values for each switch, can simply be added as constants to the modeling formulas of DNC. Second, we observed that switches have some very unpredictable behavior regarding their flow management. For example, the Pica8 switches can unpredictably drop rules without leaving any means to the controller to be aware of it. Would there be a way to avoid sending new rules to the switches at runtime? That would prevent such unpredictable artifacts to happen.

A potential solution is to use source routing, e.g., based on a stack of VLAN tags. This way, the forwarding table of the switches has to be programmed only once and embedding additional flows does not require to interact with the flow table of switches but only requires to implement the tagging on the source end host. As a small prototype, Fig. 4.15 shows that all our switches (but the HP switches) indeed support such as setup: forwarding to a particular queue based on the outermost VLAN tag and pop it.

Finally, we observe that the switches are based on a shared buffer infrastructure, in opposition to what traditional latency models assume [Gro+15]; [KCL14]; [GVK17]; [Jan+15]. We indeed observe that a given queue gets a different buffer capacity based on the current congestion state of the switch. While it may seem that the buffer capacity made available to a queue is unpredictable, thereby making packet loss prevention complicated, our measurements give some hope. Indeed, we observe that the buffer capacity of a port, though variable, depends only on the number of congested ports and queues. Usually, one does not use all the ports of a switch: for example, in a $k = 4$ fat-tree topology, one only needs 4 ports. We can then use the values resulting from our measurements as worst-case buffer capacity, instead of the very pessimistic strategy of dividing the total buffer memory by the total number of queues (number of priority queues times the number of ports).

Generally, while our results refer to the investigated specific SDN switches, we expect many of our results to also be valid for P4 programmable devices [Bos+14]. Indeed, in both cases, forwarding is done by TCAM tables and a firmware (independent of the P4 code) which implements buffer and flow management. The need for predictability for P4 devices might, however, be even more stringent, as a P4 programmer expects to have full control over its device, and hence expects a strictly predictable behavior based on its implemented logic.

4.2 Chameleon: High Utilization with Queue-Aware and Adaptive Source Routing

The previous section was motivated by the observation that a predictable network behavior critically depends on the underlying hardware. We presented a methodology and reported on our measurement study using different switches from different vendors, and identified several shortcomings, in terms of performance but also in terms of correctness. We understand this work as a first step and believe that it opens several interesting avenues for future research. In particular, more research is needed on how to design and model network systems toward more predictable and deterministic network architectures meeting the requirements of future applications. This is the gap we fill in this section.

This section is further motivated by the unprecedented routing flexibilities provided in modern networks, which in principle allow networks to autonomously and dynamically re-evaluate resource allocation decisions, and hence enable novel opportunities navigate the tradeoff between predictability, performance, and resource efficiency. In particular, these routing flexibilities enable networks to become *demand-aware*: network configurations can be adapted toward the workload they serve, potentially accounting for current delays along specific paths and exploiting currently underutilized links. The challenge, however, remains how to account for such information, and how to exploit routing flexibilities while maintaining predictability.

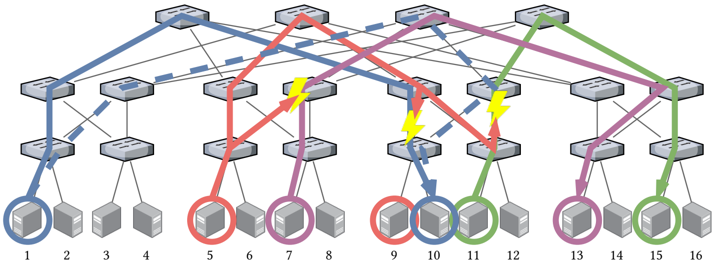


Figure 4.16: The blue (between VMs 1 and 10), purple (between VMs 7 and 13) and green (between VMs 11 and 15) flows are already embedded. In this situation, the red flow (between VMs 5 and 9) cannot be embedded by Silo. Rerouting (which Silo does not do) the blue flow on the dashed path would however make space for the new flow and allow to embed it.

A key observation of this section is that rendering networks more dynamic and adaptable does not have to contradict predictability. In particular, if done right, principles of network calculus can still be employed and the resulting performance guarantees along routing paths maintained. That is, networks can be adapted to flows arriving over time while still providing hard guarantees at all times.

4.2.1 Motivation: Unexploited Opportunities

A main motivation for our work are the unexploited optimization opportunities available in current networks: SoA networks are operated in a fairly inflexible and demand-oblivious manner. We argue that this can lead to both suboptimal network performance and low predictability of performance (in terms of latency and throughput), which leads to unnecessarily low utilization. In the following, we identify and discuss such missed optimization opportunities. Later in this section, we will show that it is indeed possible to exploit these opportunities and operate networks in a dynamic and demand-aware manner, without sacrificing predictability.

4.2.1.1 The Price of Static Allocation

State-of-the-art approaches for providing predictable network performance have the common feature that they are fairly static: embedding decisions (e.g., related to the route or per-flow rate), once taken, are usually not reevaluated nor adapted later: SoA solutions are not designed for reacting to flows arriving over time. In environments where networks need to provide guarantees and, hence, perform admission control, this can lead to unnecessarily high network flow rejection rates. For example, if the network configuration chosen earlier does not fit the characteristics of arriving flows, these flows need to be rejected. In contrast, in a dynamic and demand-aware network, it may still be possible to accept these flows, using reconfigurations. To be more concrete, let us consider the two main solutions providing predictable latency in the cloud: Silo [Jan+15] and QJump [Gro+15].

QJump. QJump [Gro+15] relies on information about application performance requirements, related to latency, rate and packet size, at network initialization time. This information is then used to compute the QJump formula: a maximum latency of $2nP/R+\epsilon$. Here, n is the number of applications using the system, P the packet size, R the links rate, and ϵ the cumulative processing time, which is guaranteed to all applications, assuming that they transmit at most one packet per each of these time periods, i.e., at a rate of at most $P/(2nP/R + \epsilon)$ [Gro+15]. While the ϵ and R parameters are constant and dependent on the physical topology only, the n and P parameters must be defined upfront, at network initialization time; this is necessary to be able to compute the maximum latency guarantee that the system will provide to flows.

Let us consider a $k = 4$ fat-tree topology with 10 Gbps links and with a cumulative E2E processing time of 4 μs. Here we have $R = 10$ Gbps and $\epsilon = 4$ μs. If the network operator decides to authorize 10 VMs per server and packets of at most 300 bytes, we have $n = 16 \times 10 = 160$ applications and $P = 300$ bytes. As a result, the QJump system guarantees a maximum latency of 80.8 μs at a rate of at most 29.7 Mbps for each VM.

While providing predictable performance, these static allocations can lead to unnecessary request rejections and as a result low utilization. For example, even if the network is unused, QJump would in this situation not admit a tenant request for a flow with a latency requirement of 50 μs. Similarly, a request for a 50 Mbps flow would be rejected unnecessarily. If an applications needs much less bandwidth than 29.7 Mbps, say 3.11 Mbps, and tolerates a higher latency guarantee than 80.8 μs, say 772 μs, the system will accept only 160 flows, while 1600 of these flows could have been accepted if the network operator initially decided to define $n = 1600$. Similar observations can be made for applications that send packets smaller or greater than 300 bytes.

This is a real concern, especially in cloud environments where tenant applications are typically unknown, requirements are hard to estimate, and elasticity (pay as you grow) is a key advantage.

In ***conclusion***, a greedy and inflexible parameter selection at network startup time can lead to unnecessary flow rejections and low network utilization.

Silo. Silo [Jan+15] also provides latency guarantees by leveraging admission control and relying on DNC modeling. At network startup, Silo assigns a delay D_i to each link in the network. Then, upon a new flow request, the admission control logic of Silo uses DNC to calculate the worst-case delays d_i of each link if the flow was to be accepted. The flow is rejected if $d_i > D_i$ for a link i. If a flow can be accepted, its latency guarantee is $\sum_i D_i$ for all links i traversed by the flow. As a result, the number of possible delay guarantees for a given application corresponds to the number of different paths between the two VMs of the flow, i.e., $(k/2)^2 = 4$ for a fat-tree topology with $k = 4$.

To give an example, we consider a fat-tree topology with $k = 4$. We allocate a delay D_i of $D_R = 20$ μs to rack links, $D_P = 50$ μs to intra-pod links and $D_A = 80$ μs to aggregation links. Without taking detours, there is only one possible delay between any pair of VMs. Between VMs in the same rack, the delay is $2D_R = 40$ μs. Between VMs in different racks but the same pod, the delay is $2(D_R + D_P) = 140$ μs. Between VMs in different racks and different pods, the delay is $2(D_R + D_P + D_A) = 300$ μs. The situation is then similar to QJump: if a tenant needs a latency guarantee lower than these values, say 30 μs, the flow will have to be rejected unnecessarily. Similarly, if tenant applications tolerate higher latency guarantees, say 10 ms, the admission control logic of Silo will start blocking flows to avoid reaching the predefined limits, even though guarantees would still be fulfilled. By increasing the allocated delays at each link, more flows could have been accepted.

Once Silo embeds a flow, there is no reevaluation of its decision. This is illustrated in Fig. 4.16. Let us assume that the blue path (between VMs 1 and 10), the purple path (between VMs 7 and 13) and the green path (between VMs 11 and 15) are already embedded. Furthermore, let us assume, for simplicity of the example, that these flows consume the entire capacity on their links. In this situation, the red flow (between VMs 5 and 9) cannot be embedded: it is blocked by all the other flows. However, rerouting the blue flow on the dashed path, i.e., reevaluating a decision taken previously, would make space for the new flow and would actually make it possible to admit and embed it.

In ***conclusion***, similarly to QJump, Silo's predictability guarantees can come at the price of low utilization: resource allocation decisions related to link delays and flow embeddings are performed greedily, and never reevaluated again. If applications have requirements that do not match the defined link delays, Silo will reject them while they actually could have been accepted, as we will show. As a result, Silo's resource allocation and embedding approach leads to a bias in terms of the types of flows that can be accepted — and to unnecessary rejections and hence low utilization.

4.2.1.2 Unexploited Path Diversity

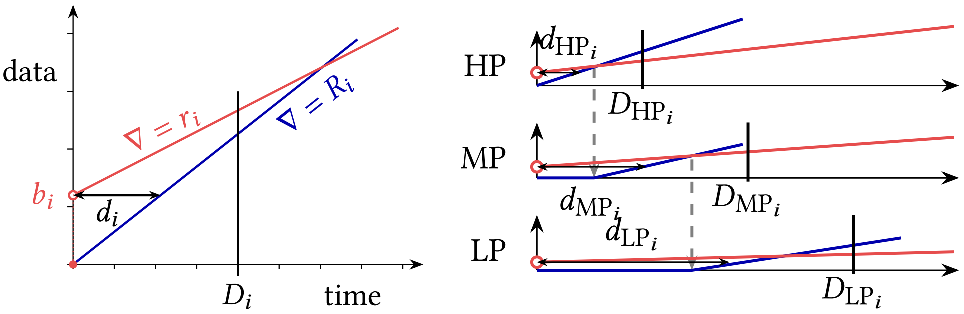
We see a great potential to exploit path diversity and more fine-granular routing to improve the efficiency and performance predictability of networks. In fact, even for the same physical path, multiple performance characteristics may be experienced: as the switches and routers along the physical route typically have multiple queues, the delay often depends not only on the specific router but also on the specific queue that is traversed. This motivates us to consider a finer granularity of routing: based on a "queue-level topology" (as defined in Sec. 2.3.3) rather than just a "router-level topology".

Surprisingly, SoA solutions do not exploit physical path diversity. For example, QJump's admission control algorithm does not account at all for the specific paths on which flows can be routed. As a consequence, QJump does not reserve network resources per switch or per link, but for the whole network, which can be inefficient. It implies that QJump assumes that two flows, even if they are disjoint, consume the same resources.

Let us illustrate the problem again with a fat-tree topology with $k = 4$ as in Fig. 4.16. Taking the parameters as in Sec. 4.2.1.1, QJump will accept up to 160 flows with delay requirements of at most 80.8 μs and a rate of at most 29.7 Mbps. Let us consider that the 160 flows accepted by QJump are located in the two leftmost pods, which is possible with a simple first-come first-serve VM allocation algorithm. In this case, QJump will reject any new flow request because it reached the maximum of n resources. At the same time, half of the network is unused although it could actually accommodate more flows: the lack of routing knowledge leads to unnecessarily rejections. Similarly, while Silo reserves resources per link, there is no optimization of routes nor of priorities assignment in the network. By not optimizing nor accounting for the routes where flows are embedded, such approaches are not demand-aware, as the network state and performance characteristics depend on the specific route taken.

4.2.2 *Chameleon* System Design

Motivated by the above shortcomings and opportunities, we now describe the design of *Chameleon*, which combines adaptive source routing and priority queuing. The latency modeling of QJump is topology-agnostic and assumes the same traffic envelope for all the flows with deterministic require-



(a) Silo: per link.

(b) *Chameleon*: per queue.

Figure 4.17: Silo and *Chameleon* modeling for access control. Silo models links while *Chameleon* goes one level lower and defines one model per priority queue. This offers higher delay diversity to applications.

ments. These fundamental assumptions prevent it from being adapted to solve the above-mentioned shortcomings without a complete redesign of the system. As a result, we choose to build upon Silo. *Chameleon* is based on four building blocks: it builds upon Silo, leverages priority queuing, and applies fine-grained source routing and adaptive reconfigurations. We discuss these building blocks in turn. These correspond to the architecture defined in chapter 2. The routing procedure just has the additional task of performing adaptive reconfigurations.

4.2.2.1 Building Block 1: Silo

Like Silo, *Chameleon* leverages three basic components: the resource allocation that is run at network startup (Sec. 4.2.2.1), the access control logic that ensures that flows are embedded only if all delay requirements are satisfied (Sec. 4.2.2.1) and the resource reservation (Sec. 4.2.2.1) responsible for keeping track of resources usage at runtime.

Resource allocation. Silo keeps track of resource consumption and per-link worst-case delays using DNC. Each link i is assigned a maximum delay D_i. We call this the *resource allocation*, as each link is allocated a delay budget. Then, Silo always makes sure that the DNC worst-case delay d_i of each link remains lower than its maximum budget D_i, i.e., the admission control of Silo ensures that $d_i \leq D_i \quad \forall i \in \mathcal{G}$, where $\mathcal{G}$ represents the set of links in the network. This is illustrated in Fig. 4.17a for a particular link. Based on the token-bucket burst and rate parameters of each flow, Silo keeps track of the total burst b_i and rate r_i consumed at each link, forming the DNC arrival curve for this link. The DNC service curve of the link corresponds to the rate R_i of the link. The horizontal deviation $d_i = b_i/R_i$ between these two curves represents the worst-case delay at this link.

Access control. A new flow request f consists of token-bucket parameters b^f and r^f, and of a maximum delay requirement d^f. At each link where the flow shall be embedded, Silo checks whether adding the token-bucket parameters b^f and r^f to the already used resources b_i and r_i would result in $d_i > D_i$. If this does not happen for any of the links supposed to be traversed by the flow and if the sum of all the delays D_i of these links is lower or equal to the delay requirement d^f of the flow, the flow is accepted. Otherwise, it is rejected.

Resource reservation. When a flow is accepted, its token-bucket parameters are simply added to the b_i and r_i parameters of each link it traverses. Note that, per DNC, the burst b^f requirement of a flow increases at each hop. Indeed, at a switch, the burst of a flow increases for each new packet that arrives while previous packets are still queued. Formally, the burst increases by $r^f D_i$ at each hop; the maximum amount of data that can accumulate while packets are queued. This is taken into account by Silo when checking resource consumption at each link, as well as when reserving resources to account for the embedding of a new flow.

We note that Silo also incorporates a VM placement algorithm. In this thesis, we focus on the embedding task and hence assume that a flow already has a source and destination server assigned.

4.2.2.2 Building Block 2: Priority Queuing

In order to increase the delay diversity offered to applications, i.e., to offer different service levels, *Chameleon* uses priority queuing. Each output link i now corresponds to n_i priority queues. In order to ease the parallelism with Silo, we present subsequently how the resource allocation, access control and resource reservation mechanisms are changed. These mechanisms strictly correspond to those of the TBM described in Sec. 2.4.

Resource allocation. Delays D_q are assigned by a resource allocation algorithm to each queue q. The set of different delays that a physical path can offer is now multiplied by the number of combinations of priority levels at each hop.

Access control. The process is illustrated in Fig. 4.17b for $n_i = 3$ priority queues. *Chameleon* keeps track of token-bucket resource consumption parameters for each individual priority queue. The service curves offered to each queue are governed by DNC. Non-preemptive priority queuing scheme requires high priorities to wait for at most one packet of a lower priority before being transmitted. As a result, the highest priority queue service curve is identical to the Silo case (i.e., it is the link rate) but is shifted towards the right by $l_{max}/R + \phi$, where l_{max} is the maximum packet size in lower priority queues, R is the link rate and ϕ is a parameter for accounting for the overhead of the priority queuing implementation in the switch[6]. As measured in Sec. 4.1.2.2), ϕ is typically in the order of microseconds. The service curve of a low priority queue then corresponds to the difference between the service curve of the higher priority queue and the arrival curve of the traffic traversing the latter. This is shown in Fig. 4.17b.

When trying to embed a flow on a path (of queues), the access control must be slightly adapted to account for the presence of lower priority queues. Indeed, adding a flow to a queue modifies the service curve offered to lower priority queues, and could hence violate the $d_q \leq D_q$ constraint for these queues and hence lead to the violation of the guarantees provided to already embedded flows. When checking if a flow can be added to a particular queue, the delay of lower priority queues also has to be checked. Additionally, the access control must check that the buffer capacity of each queue at the link is not violated. DNC provides a bound on the worst-case buffer occupancy at a system:

[6] Note that, for computing *per-packet* delays, DNC requires service curves to be shifted down by the maximum packet size of the flow [VK16]; [LT12]. While our implementation takes this into account, for simplicity, and because this is a very small value, we omit this in our description.

it corresponds to the vertical deviation between the arrival and service curves. The access control simply checks that this deviation stays lower than the buffer capacity of each queue. In order to reduce the computed bounds, arrival curves are shaped by the rate of the input link where they are coming from.

Resource reservation. When a flow is accepted, its token bucket parameters are simply added to the b_q and r_q parameters of each queue it traverses. Additionally, the service curves of the lower priority queues also have to be updated (as described in Sec. 4.2.2.2) to reflect the change in the arrival curve of a higher priority queue. The burst increase of flows is of course handled in the same way as for Silo.

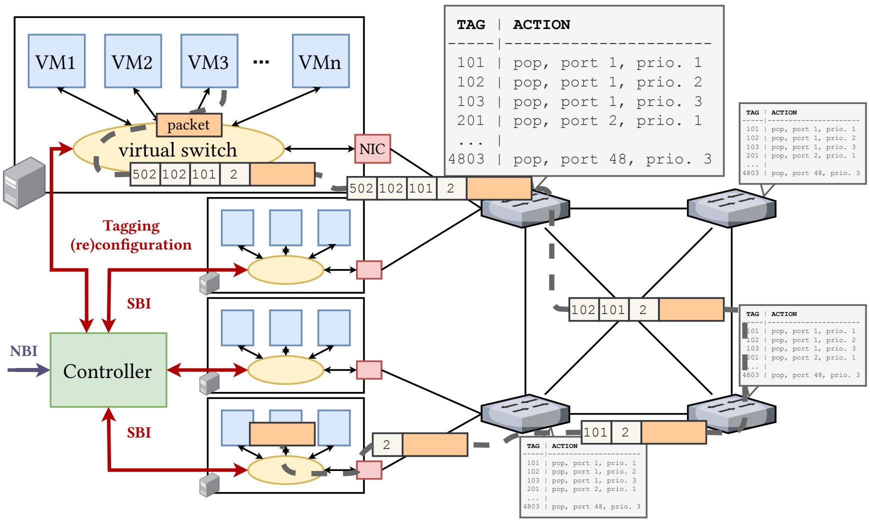
4.2.2.3 Building Block 3: Routing

The introduction of priority queuing changes the path finding problem. Besides the physical path, also priority queues have to be selected. This corresponds to finding a path in the *queue-level* topology defined in Sec. 2.3.3. In this queue-level topology, each queue/edge q has been allocated a delay D_q by the resource allocation algorithm. Finding a path for a flow request then consists in finding a path $\mathcal{P}$ such that $\sum_{q \in \mathcal{P}} D_q \leq d^f$, and for which the access control allowed access to the flow. If we introduce a *cost function* that defines a metric value for each queue in the network, this corresponds to a DCLC routing problem, the problem we investigated in chapter 3. The cost function must be defined in a way that makes the routing algorithm consume the least amount of resources for each flow and hence maximizes the probability of accepting future demands. We will come back to this in Sec. 4.2.2.4. As we have seen in chapter 3, the DCLC routing problem is NP-complete and optimal routing algorithms exhibit too high memory consumption and runtime to be used as online routing algorithms. Hence, a sub-optimal, yet fast and complete algorithm has to be used, e.g., LARAC [Jüt+01]. As we have seen in Sec. 3.6, the access control, because it depends on the burst of a flow, is an M∞ metric and impacts the completeness of the routing algorithm. In particular, as shown in Tab. 3.4, this makes the routing procedure both sub-optimal *and* incomplete. We discuss the impact of this and how we cope with it in the next section.

4.2.2.4 Building Block 4: Reconfigurations

When a flow is routed, it is the role of the cost function to direct the routing algorithm such that the least amount of resources is consumed. However, the cost function is not aware of upcoming requests and, as we highlighted in Sec. 4.2.1.1, a low-cost path for routing a flow f might happen to block a later flow g. Finding a cost function that is good for any network scenario is a challenging problem, as whether a choice now is good for later is only defined by the upcoming flows, which are unknown to the cost function. As we leave a more detailed study of cost functions for future work, we instead use a dummy cost function (e.g., least-delay) and when the routing procedure fails at embedding a new routing request, it can analyze the current network state, reroute already embedded flows to make space for the new flow and then embed the original flow.

However, reconfiguring running flows constitutes a big challenge. Algorithmically, consistent network updates is a complex task, especially in the presence of strict latency guarantees [FSV18].



```
TAG | ACTION
-----|----------------------
101 | pop, port 1, prio. 1
102 | pop, port 1, prio. 2
103 | pop, port 1, prio. 3
201 | pop, port 2, prio. 1
... |
4803 | pop, port 48, prio. 3
```

Figure 4.18: Example of *Chameleon* in operation: VM 3 on the first server sends a packet to VM 2 on the fourth server. Hypervisors in servers tag packets of the different VMs to define the path they take and their priority level at each hop. This enables easy reconfigurations and circumvents traditional issues in distributed network reconfigurations.

Additionally, we have shown in Sec. 4.1.3.1 that the management interface exposed by existing programmable devices is not always predictable, the controller hence being unsure whether its desired configuration update is indeed implemented in the data plane. Furthermore, other measurement studies have shown that updating forwarding rules on programmable devices can lead to transient phases during which packets are forwarded on both paths [KPK14], an unacceptable situation for predictability.

To circumvent these issues, we propose to use source routing for configuring forwarding decisions, similarly to what Microsoft uses in their datacenters [Fir17]; [Fir+18]. This is illustrated in Fig. 4.18. Instead of forwarding based on a five-tuple matching, the forwarding elements in the network use a tag in the packet to define their behavior. Each tag corresponds to a port-queue combination. When sending packets from VMs, hypervisors push a stack of tags corresponding to the path the packet has to follow and the priority levels at which it should be enqueued. For example, if a priority level p and output link l correspond to tag $t = 100l + p$, a stack of tags 101, 503 means that the packet should be forwarded to port 1 of the first switch and with priority 1 and then to port 5 of the second switch with priority 3. The switches simply match on the tag to perform the corresponding action and then pop the outermost tag out of the stack to permit the next-hop switch to read the next tag.

This approach solves the above-mentioned issues. Indeed, the forwarding behavior of switches is configured once at startup and never has to be updated. This eliminates the unpredictability problem of the management interface of switches and the transient phase issue when updating flow tables. Further, routes are configured on end-hosts, which eliminates the problem of consistently updating the network configuration, as the network is configured centrally.

This however brings another important challenge. The tagging in the hypervisor virtual switch, and the updates of its tagging rules must be predictable, as this new component adds an additional delay to the packets. However, in contrast to blackbox forwarding devices based on closed implementation, the virtual switch is under our complete control. That allows us to specifically design it for satisfying these requirements. In particular, several recent technologies such as P4 and DPDK offer the potential for achieving this predictability. We describe in detail in Sec. 4.2.3.2 how we implement the tagging component and confirm in Sec. 4.2.4.2 that this implementation is predictable and fast enough for predictable latency use cases.

4.2.3 *Chameleon* Implementation

We separate our description of the implementation of *Chameleon* into the control plane and the data plane parts.

4.2.3.1 Control Plane

The control plane is implemented as a multi-threaded set of Java 8 libraries implementing all the controller functionalities. The code consists of around 30k lines of code.

Interfaces. The controller implements a NBI that exposes a representational state transfer (REST) application program interface (API) to users (Fig. 4.18). This API allows tenants, VMs, and flows to be created and deleted through hypertext transfer protocol (HTTP) *POST* requests. A tenant is a logical abstraction that supports users to create flows between VMs that they created, i.e., VMs of the same tenant. All created VMs are identical and allocated to a randomly selected physical server. VM placement is outside the scope of this work. The creation of a flow requires the specification of source and destination VMs, of burst, rate, and latency requirements, and of a five-tuple matching structure. A counterpart southbound interface (SBI) module implements the OF 1.0 protocol (Fig. 4.18). The module discovers the network topology at startup using link-layer discovery protocol (LLDP) packets and configures the static forwarding rules on switches (see Sec. 4.2.3.2). Upon a VM creation request, the SBI module triggers the actual creation on the chosen server via secure shell (SSH). Upon a flow embedding request, the NBI module forwards the request to the routing procedure. If the routing procedure returns an embedding, or if it requests the reconfiguration of a previous embedding, the SBI configures the corresponding tagging rules on the source server via SSH (Sec. 4.2.3.2). We do not aim at providing strict guarantees for request processing times. As a result, the communication between the SBI and the servers does not need latency guarantees and can happen over a traditional control network.

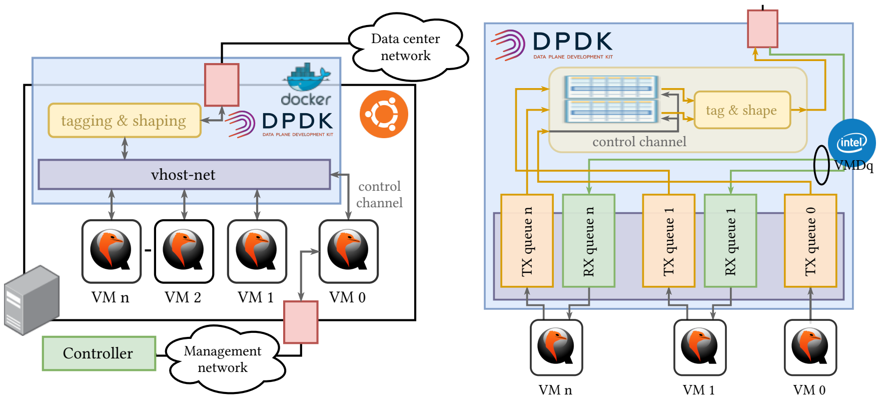
Resource allocation and reservation, and access control. The resource allocation simply assigns a maximum delay to each queue upon discovery of a new link (as described in 4.2.2.2). For 8-queue ports, it assigns the following delays: 0.1 ms, 0.5 ms, 1 ms, 1.5 ms, 3 ms, 6 ms, 12 ms, and 24 ms. For 4-queue ports, it assigns 0.1 ms, 1 ms, 6 ms, and 24 ms. Host ports towards their access switches are assigned 0.5 ms. This somewhat arbitrary delay assignment is chosen to be able to span delay requirements from sub-milliseconds to hundreds of milliseconds. The optimization of the maximum delays assignment is left for future work. Access control and resource reservation are implemented as described in Sec. 4.2.2.2.

```
 1: function EMBEDDINGSTRATEGY(request)
 2:     response ← ROUTE(request)
 3:     if response $\neq$ NULL then
 4:         RESERVE(response), return response
 5:     for each flowToReroute in SORT(GETFLOWSTOREROUTE(request)) do
 6:         INCREASEGRAPHCOSTS(flowToReroute, request)
 7:         reroutingResponse ← ROUTE(flowToReroute)
 8:         if reroutingResponse $\neq$ NULL then
 9:             RESERVE(reroutingResponse)
10:             FREE(flowToReroute.originalPath)
11:             response ← ROUTE(request)
12:             if response $\neq$ NULL then
13:                 RESERVE(response), return response
14:     return NULL
```

Figure 4.19: Pseudo code of the flow embedding and reconfiguration.

Routing and rerouting strategies. The routing procedure for finding a DCLC embedding is implemented using the LARAC algorithm [Jüt+01] as described in Sec. 4.2.2.3. The complete routing and rerouting logic is shown in Fig. 4.19. First, the procedure tries to find a path for the flow request using a least-delay search (line 2). If it fails at finding a valid embedding (either because of its incompleteness – Sec. 4.2.2.3 – or because of previous flows poorly embedded – Sec. 4.2.1.1), the procedure tries to reroute already embedded flows to make space for the new one. First, in line 5, it selects a set of sorted candidate flows to be rerouted and iterates through them. In our implementation, it selects all the flows traversing at least one edge of any of the equal-length SPs in the physical topology from the source server to the destination server of the new flow to embed. Those paths are found using Yen's algorithm [Yen71]. Other flows not traversing these SPs are indeed not expected to prevent the new flow from being embedded. In our implementation, we sort flows according to the number of physical links they share with the SPs of equal length in the physical topology between the source and destination servers of the new flow to embed. Then, for a given candidate flow to reroute, based on the current state of the network, the procedure tries to re-embed the selected flow. In line 6, to direct the routing procedure toward a path that potentially allows the new flow to be embedded, we increase the cost (see Sec. 4.2.3.1) of the previous queues in which the flow was embedded (to move it somewhere else) and of all the queues of all the equal-length SPs between the source and destination servers of the new flow to embed (to prevent the rerouted flow to interfere with the new flow). The cost is increased by multiplying the original cost value by an arbitrary high value (30 000 in our implementation). If the flow cannot be re-embedded, the procedure continues to the next candidate flow. If the flow can be re-embedded, the procedure notifies the SBI to reserve the new embedding (line 9) and then to free the resources reserved for the previous embedding (line 10), and then retries to add the new flow. If that fails, the procedure continues to the next candidate flow to reroute. If that succeeds, the new flow is successfully embedded thanks to the reconfiguration of previous flows. If the list of candidate flows is exhausted without any success, embedding failed and the new flow is rejected (line 14).

Cost function. As described in Sec. 4.2.2.4, the design of a good cost function is a very challenging task. In some sense, the proposed rerouting strategy is a way of adapting the cost function to



(a) Server.

(b) DPDK application.

Figure 4.20: *Chameleon*'s end-host implementation (a) and zoom in the DPDK application. QEMU VMs are connected to the DPDK application using a *vhost-net*/*virtio-net* architecture and communicate through distinct receive and transmit queues. A control VM allows the configuration of the tagging/shaping rules.

future requests, as we reroute an old flow with the knowledge of the flows that were accepted later. Unfortunately, also the rerouting procedure needs a cost function. We decide to simply use the *delay* as cost, thereby effectively degenerating the DCLC problem into a simple least-delay routing problem solved by Dijkstra's algorithm [Dij59]. The rationale behind this decision is that the burst increase of a flow at each hop is proportional to the delay of this hop (see Sec. 4.2.2.1). This cost function hence minimizes the resource (burst) consumption of flows.

4.2.3.2 Data Plane

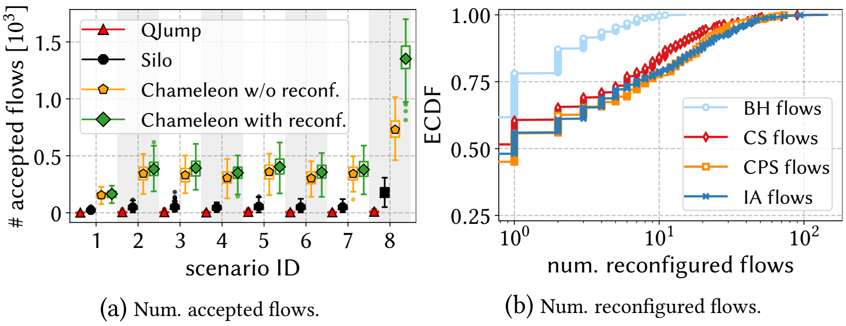
We consider 1 Gbps OF 1.0 devices: Dell S3048-ON and S4048-ON (four priority levels per port), and Pica P3297 (eight priority levels) switches. We use servers running Ubuntu 18.04 (4.15.0-66-generic kernel) with 64 (Dell servers) or 128 GB (Dell and Supermicro servers) of random access memory (RAM), an Intel Xeon Silver 4114 @ 2.2 GHz (20 cores, Supermicro servers) or an Intel Xeon E5-2650 v4 @ 2.2 GHz (24 cores, Dell servers) as CPU, and an Intel X550 (Supermicro servers) or X540 (Dell servers) NIC towards the data network. We use VLAN tags to implement the tag stacks. While any other stackable tagging mechanism can be used (e.g., multi-protocol label switching (MPLS)), we use VLAN tags because of its low header overhead and its more widespread support. Furthermore, we have shown in Sec. 4.1.4 that matching on VLAN tags to output to a particular port and queue and popping the outermost VLAN tag can be done at line rate and with a predictable performance by simple OF 1.0 devices.

In the following, we describe the end-host implementation (Fig. 4.20), the cornerstone of our solution. This consists of a *tagging* part, responsible for pushing tag stacks to packets, and of a *shaping* module, responsible for ensuring that applications do not exceed their negotiated b^f and r^f token-bucket parameters. We implement the virtual switch of the VMs hypervisor as a DPDK 19.08 application running in a privileged *docker* container. The general architecture of the virtual switch is shown in Fig. 4.20a. The different VMs run in QEMU 2.11.1 with KVM. The VMs and the DPDK application are connected through *virtio* using a *vhost-net*/*virtio-net* para-virtualization architecture [AIN].

How to ensure predictability? The processing of the virtual switch has to be predictable, in terms of latency. To do so, the DPDK application is pinned to specific cores of the server and we prevent the kernel of the server to use these cores using the kernel *isolcpus* parameter. To avoid unpredictable performance variations, we further disable hyper-threading, turbo-boost, and power saving features of the CPU. This ensures that the DPDK application runs isolated on dedicated CPU cores that operate at a constant and stable speed. We use Intel's cache allocation technology (CAT) to allocate a specific portion of the CPU last level cache (LLC) to the cores used by the DPDK application. As level-one and level-two caches are per core, this prevents other applications from interfering with DPDK through the memory caches. We use three cores for the application: one sending core, one receiving core, and one master core for the DPDK master process. Both sending and receiving cores process batches of packets for the different VMs in a round-robin fashion. Each VM is assigned a sending and a receiving queue (see Fig. 4.20b). The sending queues are filled by the VM *virtio* drivers and emptied by the sending core, which is then responsible for tagging and shaping before sending out the packets to the NIC. The receiving queues are filled by the receiving core. The destination VM of a packet is identified by its MAC destination address and VLAN tag. Doing this separation in software would prevent batch processing, a major enabler of the fast software processing performance of DPDK. Indeed, a series of packets received from the NIC is not necessarily entirely destined for the same VM. Hence, we use the virtual machine device queues (VMDQ) technology of Intel NICs. Packet separation is done in hardware and packets for the different VMs are automatically stored in separate physical queues that are then simply pulled by the receiving core and sent to the different VMs *virtio* drivers.

Tagging. The sending core, after pulling a batch from a VM sending queue, is responsible for tagging the packets. The program maintains tagging rules with the following fields: *protocol*, *source IP*, *destination IP*, *source port*, *destination port*, *number of tags to push*, *tags to push*. The entries are stored in a two-dimensional array indexed by the VM ID and the rule ID for a given VM. The maximum number of VMs (64 in our implementation) and of rules per VM (3 in our implementation) is fixed and the array hence does not require dynamic memory allocation. Within a processed batch, for each packet, the core traverses the 3 rules of the VM it is currently serving. If a five-tuple match is found, the tags stored in the corresponding entry are directly copied between the Ethernet and IP headers of the packet. If no match is found, the packet is dropped. Once all packets of a batch are processed, the program sends the batch of tagged packets to the NIC.

Shaping. The sending core must ensure that flows do not exceed the token-bucket parameters that have been reserved for them. Indeed, a violation of these parameters invalidates all the DNC computations and can hence lead to delay guarantees violations. We add four fields to the tagging rules: *rate*, *burst*, *number of tokens*, and *last timestamp*. The two first fields store the token-bucket parameters of the entry, the third and fourth fields store the number of tokens in the token bucket when they were last computed and the corresponding timestamp. For each packet within a processed batch, the sending core computes the updated number of tokens based on the current timestamp and the rate parameter of the token bucket. The packet is dropped if there are not enough tokens for sending the packet. Otherwise, the number of tokens corresponding to the packet size are removed,



(a) Num. accepted flows.

(b) Num. reconfigured flows.

Figure 4.21: Simulation results. (a) indicates the increased number of accepted flows in *Chameleon* compared to the SoA systems and (b) shows the number of reconfigured flows per flow type and the effect of the characteristics of flows on their reconfigurability.

the timestamp is updated and the packet is kept. Timestamps are obtained using the timestamp counter (TSC) register of the CPU. Having disabled the turbo-boost and power-saving features of the CPU ensures that this counter measures real time and not simply the number of instructions.

Configuration of tagging/shaping rules. In order to communicate with the virtual switch without creating unpredictability and synchronization issues, we use an additional VM, the control VM (with ID 0). This VM is not allocated a receiving queue (see Fig. 4.20b). The packets sent by this VM are used to configure the rules stored in the sending core. When the sending core receives a control packet, the first two bytes of the packets are used to index the table – they correspond to the VM ID and rule ID to update. The next bytes in the packet are simply copied in the entry. The *Chameleon* controller connects to this control VM to update tagging/shaping entries and is hence responsible for sending the appropriate values in the correct order and endianness. The DPDK application then simply reinitializes the *number of tokens* and *last timestamp* fields of the modified entry.

4.2.4 Evaluation

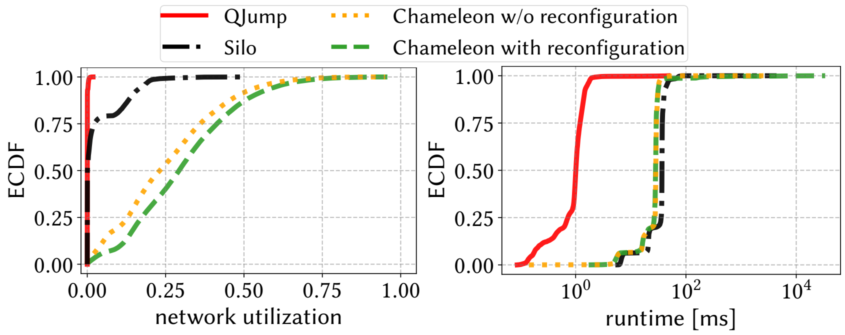
The goal of our evaluation is to show that *Chameleon* successfully provides latency guarantees and can reach higher network utilization than existing approaches. First, Sec. 4.2.4.1 evaluates the utilization and number of flows our system can accommodate by running simulations of its admission control. For different types of traffic distributions, we show that *Chameleon* reaches higher network utilization and number of accepted flows than the SoA QJump [Gro+15] and Silo [Jan+15] systems. Second, in Sec. 4.2.4.2, we perform a microbenchmark of our end-host tagging and shaping implementation. We show that our implementation is accurate in shaping flows and can tag packets at high rates. Finally, in Sec. 4.2.4.3, we deploy the *Chameleon* system in a testbed composed of ten switches and eight servers. We show that *Chameleon* can improve the performance of applications that run on a shared network infrastructure and that the packet delays guaranteed by the system are indeed not violated.

Flow description	Rate	Burst	Deadline
Category 1: Industrial applications (IA) [Kat+17a]; [Vir17]			
Database operations	[300, 550] Kbps	[100, 400] byte	[80, 120] ms
SCADA operations	[150, 550] Kbps	[100, 400] byte	[150, 200] ms
Production control	[100, 500] Kbps	[100, 400] byte	[10, 20] ms
Control and NTP	[1, 100] Kbps	[80, 120] byte	[10, 20] ms
Category 2: Clock synchronization (CS) [Pop19]			
PTP	[1, 220] Kbps	[80, 300] byte	[2, 4] ms
Category 3: Control plane synchronization (CPS) [SK18]; [IEC18]			
Eventual consistency	[2, 4] Mbps	[80, 140] byte	[50, 200] ms
Strict consistency	[5, 8] Mbps	[1000, 3000] byte	[50, 200] ms
Adaptive consistency	[2, 4] Mbps	[80, 120] byte	[50, 200] ms
Category 4: Bandwidth-hungry applications (BH) [Mel+19]; [Ali+11]; [WMZ19]; [Ali+12]			
Hadoop, data-mining	[100, 150] Mbps	[1000, 5000] byte	[10, 100] ms
Hadoop, data-mining	[100, 200] Mbps	[1000, 3000] byte	[10, 100] ms
Hadoop, data-mining	[80, 200] Mbps	[1000, 3000] byte	[50, 100] ms

Table 4.5: Considered flow types and their characteristics.

Scenario ID	Distribution (IA, CS, CPS, BH)	Scenario ID	Distribution (IA, CS, CPS, BH)
1	(0.25, 0.25, 0.25, 0.25)	2	(0.2, 0.2, 0.5, 0.1)
3	(0.2, 0.5, 0.2, 0.1)	4	(0.5, 0.2, 0.2, 0.1)
5	(0.1, 0.4, 0.4, 0.1)	6	(0.4, 0.1, 0.4, 0.1)
7	(0.4, 0.4, 0.1, 0.1)	8	(0.33, 0.33, 0.33, 0.01)

Table 4.6: Flow request distributions used in the simulation.



(a) Network utilization.

(b) Runtime.

Figure 4.22: (a) Improved network utilization achieved by *Chameleon* compared to QJump and Silo. (b) Runtime of embedding one flow in the network.

4.2.4.1 Network Utilization

We conduct a comprehensive simulation study comparing *Chameleon* with the two main SoA approaches for predictable latency: QJump [Gro+15] and Silo [Jan+15]. We consider a $k = 4$ fat-tree topology with 1 Gbps physical links and 16 servers.

Configuration of the systems. For *Chameleon*, at each physical port, we consider 8 queues each with 97 KB buffer size, according to our results from Sec. 4.1.3.2 for Pica8 P3290 and P3297 switches. For Silo, because it does not use priority queuing, we set a single queue with 590 KB of buffer size, still according to our results from Sec. 4.1.3.2 (Fig. 4.14a). We set the Silo per-link delay to 0.1 ms[7]. For QJump, we have $R = 1$ Gbps, $\epsilon = 4\ \mu s$ [Gro+15], we consider the maximum packet size $P = 1500$ byte, and we set $n = 32$[8].

Simulation setup. We define a set of flow requests as an input to evaluate the performance of the different systems. A flow request is defined by its source and destination nodes, and requested rate, burst, and deadline. We choose the source and destination of each flow request randomly from the hosts in the network. To specify the rate, burst, and deadline values, we define different types of application categories: industrial applications (IA), clock synchronization (CS), control plane synchronization (CPS), and bandwidth-hungry (BH) applications (see Tab. 4.5). Each category of application is defined by a set of distributions for rate, burst, and deadline values according to SoA references as reported in Tab. 4.5. This allows us to define a wide range of different scenarios and confirm that *Chameleon* performs well under any scenario. To randomly sample flow requests, we use a flow request distribution (a, b, c, d), where a, b, c, and d are the probabilities of a flow to belong to the IA, CS, CPS, and BH categories. For example, the flow distribution of scenario 1 in the Tab. 4.6 indicates that the probability of having a flow request from each category is the same and is equal to 0.25. After that, for a given flow category, we randomly select one of the distributions of this category and then randomly sample the rate, burst, and deadline values uniformly within the ranges defined in Tab. 4.5. We define eight different scenarios as shown in Tab. 4.5. For each scenario and system, we perform 100 runs for which we add flows until a rejection happens. The simulation was performed on a machine equipped with Intel Core i7-6700 CPU @ 3.40 GHz, 16 GB of RAM, running Arch Linux x64 with kernel version 5.4.15-arch1-1.

Results: number of accepted flows. The comparison of number of accepted flows is shown in Fig. 4.21a for all the scenarios described in Tab. 4.6. We consider two cases for the *Chameleon* admission control system, with and without reconfiguring previously embedded flows. Yet, in the case without reconfigurations, *Chameleon* is able to accept between 2× and 10× more flow requests compared to the two SoA approaches. Additionally enabling reconfigurations allows to accept even more flow requests. The big performance difference between the SoA and *Chameleon* is due to the flexible and demand-awareness design of *Chameleon*, while SoA relies on static and greedy decisions (see Sec. 4.2.1). In particular, QJump is blocked to a maximum of $n = 32$ flows because of the necessity to define n beforehand.

We observe that the benefit provided by reconfigurations depends on the input traffic distribution. For instance, scenario 8 in Fig. 4.21a benefits more from reconfiguration than scenario 1. In fact, as it is depicted in Fig. 4.21b, flow types appear to exhibit different levels of reconfigurability. In particular, Fig. 4.21b shows that flows from the BH category are reconfigured less than the other flow types. This is due to the fact that the rate and burst of BH flows are significantly higher than

[7] We compared the performance of Silo with each of the delays we used for *Chameleon* (see Sec. 4.2.3.1) and selected the best performing value.

[8] Again, we evaluated QJump with different n values and we chose the best performing one.

other types, hence having less chance to be reconfigured (especially in a highly utilized network, see Fig. 4.22a). However, in addition to rate and burst, flow deadline plays an important role in the reconfigurability of the flows. For example, in Fig. 4.21b, it can be seen that CS flows have been reconfigured less than IA and CPS, mostly due to their tight latency requirements.

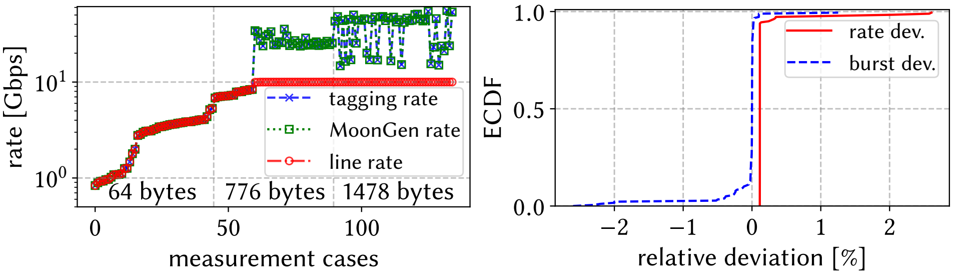
It is worth to note that according to Fig. 4.21b, although the reconfiguration operations bring a great benefit in terms of number of accepted flows, we only reconfigure a few percent of the accepted flows (less than 100 flows on average).

Results: network utilization. Fig. 4.22a shows the ECDF of the network link utilizations achieved by the different systems for all the considered scenarios. Note that we excluded the host to top-of-rack switch links from the figure. It can be seen that *Chameleon* is able to significantly increase the network utilization compared to other approaches, to reach close to line rate utilization for some links. High utilization and predictable latency are hence not anymore exclusive objectives. For network operators, that means *Chameleon* has the potential of achieving greater revenue.

Results: runtime. Fig. 4.22b depicts the comparison of runtime for embedding one flow request in the network for the different systems. We measure the time between a flow request and the reception of a response (whether positive or not). For *Chameleon*, this includes routing and reconfiguration operations. We observe that, despite the greater complexity in *Chameleon*'s logic, it achieves better runtime performance than Silo at the median. This is due to the fact that Silo runs a DCLC algorithm for finding the SP satisfying the delay requirement while *Chameleon* simply runs a Dijkstra least-delay search. However, at the tail, *Chameleon* exhibits higher runtimes, due to the reconfiguration computations. Around a dozen of seconds are reached for the considered scenario. While this is unacceptable, it can be easily avoided by stopping the embedding procedure if no response is received after a pre-defined processing time threshold. Moreover, Fig. 4.22b shows that the reconfiguration has a low runtime impact, adding a few miliseconds to the runtime.

4.2.4.2 Tagger/Shaper Microbenchmark

Tagger. We connect two Dell servers (for specifications, see Sec. 4.2.3.2) directly using 10 Gbps interfaces. In the source server, we deploy a VM generating traffic using the *MoonGen* [Emm+15] traffic generator. This generated traffic is pulled in batches through the *virtio* virtual interface by the DPDK application. The combination of parameters outlined in Tab. 4.7 is used to create the evaluation scenarios. We measure the rate of traffic generated by the VM/MoonGen, pulled by the tagger, tagged and forwarded to the NIC. These values are obtained through simple packet counters in the DPDK application. The number of packets is converted into rate using the rate measured at the destination interface (not connected to DPDK) using *tcpdump*. Fig. 4.23a shows the generation, tagging and line rate for the different scenarios, ordered by packet size. We observe that the DPDK application is fast enough to tag every pulled packet from the VM, reaching up to 40 Gbps in some cases. All the tagged packets are successfully sent to the NIC. We see that the tagging rate is either bounded by the physical link rate (10 Gbps) or by the rate achieved by the traffic generator. Hence, the tagging implementation is never the bottleneck.



(a) Tagging performance.

(b) Shaping precision.

Figure 4.23: Performance evaluation of the tagger and shaper implementation of *Chameleon*.

Shaper. We connect a Dell server directly to an Endace DAG 7.5G4 measurement card [End16] through a 1 Gbps connection. We configure our DPDK application to pull packets one-by-one (batch size of one) and to add 6 tags to them. Using the parameters in Tab. 4.8, we deploy a number of VMs running *MoonGen* and generate traffic towards the DPDK application with the corresponding packet size. Based on the traces obtained by the measurement card, the actual shaped rate and token bucket size are determined. We calculate the rate (resp. burst) deviation as the relative deviation of the observed shaped rate (resp. burst) compared to the value defined in the shaping rule. As can be observed in Fig. 4.23b, *Chameleon*'s shaper implementation exhibits a precise performance, producing a maximum relative error of around 2%. Also, although not shown in the figure, we observe that shaping is more precise for a lower number of VMs. This is because the DPDK application pulls packets in a round-robin fashion from VMs: having less VMs leads to shorter pulling intervals.

Parameter	Values	Parameter	Values
Packet size [byte]	64, 776, 1478	Batch size	1, 16, 32
Num. flows	1, 2, 3	Num. tags	2, 4, 6, 8, 10

Table 4.7: Considered parameters for the tagger evaluation.

# VMs	# Flows	Packet size [byte]	Rate [bps]	Burst [bits]
10	3	78	10^5	10^5
10	1	800	10^7	10^4, 10^5, 10^6
5	3	800, 1522	10^7	10^5
5	1	78	10^3, 10^7	10^3, 10^4, 10^5
1	3	800	10^5, 10^7	10^5
1	1	78, 800, 1522	10^7	10^6

Table 4.8: Measurement scenarios for the shaper evaluation.

4.2.4.3 Testbed Measurements

We verify the *Chameleon* system in a real $k = 4$ fat-tree testbed as shown in Fig. 4.24. The *Chameleon* controller connects to all the servers and switches through an out-of-band management network not shown in the figure. In the first experiment (Sec. 4.2.4.3), we confirm that the delays guaranteed by *Chameleon* are indeed not violated throughout the whole lifetime of flows, even when flows

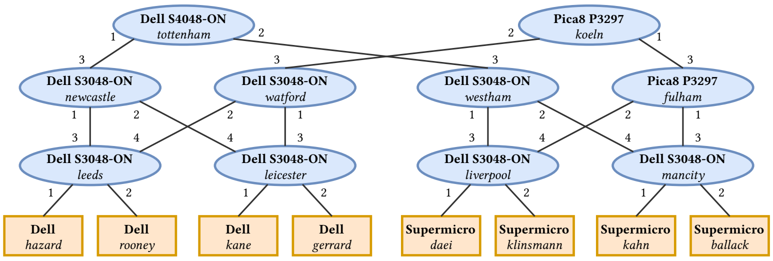
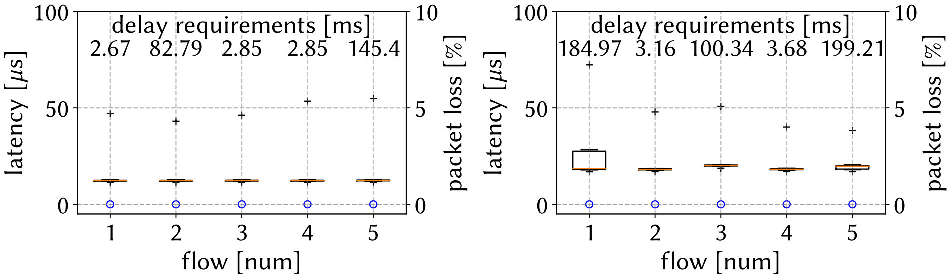


Figure 4.24: Testbed for our experiments.



(a) 4-server experiment.

(b) 8-server experiment.

Figure 4.25: End-to-end latency measured for 5 different flows. The crosses depict the maximum observed latencies. Whiskers of the boxplots show the 10% and 90% percentiles.

are rerouted or new flows are embedded in the network. In the second experiment (Sec. 4.2.4.3), we illustrate that *Chameleon* helps resolving network interference and can provide guarantees to applications even in presence of bandwidth-hungry adversarial traffic. The buffer capacities of the switches are defined as measured in Sec. 4.1.3.2.

Verification of E2E latency guarantees. In this part of our evaluation, we run the complete *Chameleon* system in the testbed depicted in Fig. 4.24. We consider two scenarios, in the first one we use the full testbed (with 8 servers), while in the second one, we use only the left pod of our topology (4 servers). To perform our experiments, we consider 31%, 31%, 31%, and 11% of, IA, CS, CPS, and BH applications and generate flow requests as in Sec. 4.2.4.1. These particular scenarios accepted a total of 298 and 218 flow requests. Using two network taps mirroring traffic to an Endace DAG 7.5G4 measurement card [End16], we measure the packet delay experienced by five random accepted flows, while ensuring that at least one of these flows was reconfigured. Fig. 4.25 presents the observed E2E packet delay of the selected flows in both scenarios. It can be seen that the delay requirements of flows are met and there is no packet loss occurring in the system. It is interesting to notice that there is very little queuing happening: most packets experience the same delay (only due to processing in the switches) and only a few packets are delayed due to queuing. This shows that, to keep queues nearly empty, a drastic and very conservative approach like QJump (which allows to send at most one packet at the same time in the network) is not necessary and that a precise DNC modeling can achieve low and predictable latency while still achieving high utilization.

Resolving network interference. Precise clock synchronization is often a requirement of distributed systems [Cor+13]. In LANs, the precision time protocol (PTP) is a master–slave protocol

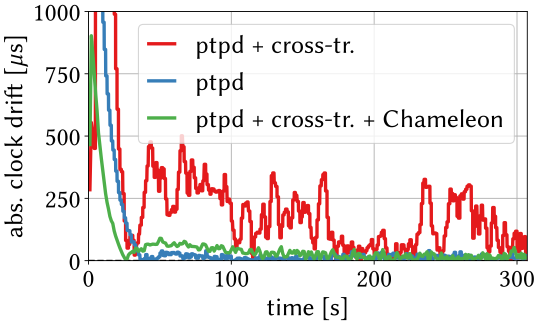


Figure 4.26: ptpd experiment. *Chameleon* resolves the interference introduced by sharing the network with bandwidth-hungry applications like Hadoop.

that is widely used for clock synchronization. It offers microsecond-granularity from a master server to other slave machines. In Fig. 4.26, we show the clock offset between a slave VM on the server *kane* and a master VM on the server *gerrard* in our testbed when both are running ptpd (version 2.3.1), an open-source implementation of PTP. The PTP application shares the network with two flows (one from a VM on server *rooney* and one from a VM on *kane*) that send Hadoop-like traffic to the VM on *gerrard* that runs ptpd: the traffic flows are competing for bandwidth with the PTP flows. The traffic is generated using *MoonGen* [Emm+15] on both VMs and emulates Hadoop traffic by sending bursts of line rate traffic, at an average rate of around 480 Mbps. We observe in Fig. 4.26 that this cross-traffic causes ptpd to fall out of synchronization in the order of hundreds of microseconds, while the clock offset of ptpd on the same idle network remains in the order of dozens of microseconds. When we introduce *Chameleon* to reserve network resources and route flows on appropriate queues through VLAN-based source routing, the interference is resolved and the ptpd synchronization offset remains as in the idle network scenario.

4.3 Summary

The main contribution of this chapter is *Chameleon*, a demand-aware cloud network that combines adaptive source routing with priority queuing to meet both performance *and* resource efficiency objectives. *Chameleon* dynamically reevaluates routing decisions, performing adjustments while maintaining network calculus invariants to ensure strict latency guarantees are provided and preserved. In extensive experiments conducted in a testbed based on real data center topologies and using large-scale simulations, we found that *Chameleon* can significantly outperform the SoA: *Chameleon* is able to admit significantly more flows, and hence increase network utilization and operator revenue, without sacrificing performance guarantees.

This chapter has shown that demand-aware and adaptive networks, leveraging source-routing and queuing flexibilities, introduce an opportunity to improve cloud network utilization while providing a predictable performance, in particular, latency. Our approach buils upon network calculus concepts while accounting for such flexibilities.

We understand our work as a first step and believe that our approach introduces several interesting avenues for future research. In particular, our algorithms are currently fairly simple, and we expect further utilization gains with improved algorithms. More generally, while we have focused on datacenters, it will be interesting to explore opportunities of self-adapting networks, based on priority reconfigurations, in wide-area networks as well. We believe that the self-adapting approaches considered in this chapter can also serve as a stepping stone toward the self-driving networks [FR17] envisioned by the networking community.

Chapter 5

Measurements and Testbed Implementation for Small Networks

Communication networks do not only form the backbone of our digital society, connecting users to the world's datacenters as well as datacenters to each other, but also are becoming ubiquitous in "small scale" environments, such as airplanes [SV08], cars [Som+10] and industrial production sites [GJF13b]. These environments often come with particular constraints: such networks have to provide stringent latency guarantees to operate properly (e.g., time-critical control loops) and their workload often has specific characteristics (e.g., related to the rate and burstiness of the arriving demands). In general, such networks rely on significantly smaller and more lightweight equipment than other networks [Tri11]; [SV08]. At the same time, small networks can still benefit from emerging flexible communication technologies, and in particular *programmability*. We define "small networks" as networks of low capacity that connect small, lightweight, and low-cost equipment. We observe that such networks have received little attention in the literature. In particular, while there exist various low-cost programmable switches based on simple hardware, such as the Zodiac FX [Nor19] or the Banana Pi R1 [Sin18a] and R2 [Sin18b], it is unexplored today to which extent such hardware can be used to provide predictable performance, and in particular latency. At first sight, it may seem challenging to provide deterministic latency guarantees with low-cost and hence low-performance and less reliable devices.

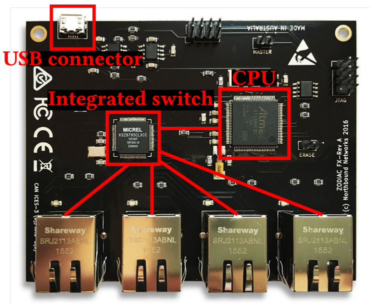
This chapter is motivated by the observation that we lack performance models for low-cost programmable switches used in small networks – a prerequisite for predictability. Indeed, as we show in this chapter, the few SoA models for predictable latency which do exist today, such as QJump [Gro+15] and Silo [Jan+15], are a poor match for such switches. Our analysis of the reasons behind this mismatch shows that existing models rely on architectural and performance assumptions that turn out to be invalid in this context. In particular, processing time on low-cost programmable devices is *not* negligible and can create interferences among the different, and up to now considered independent, switch ports. This analysis is also valid for our TBM and *Chameleon* models described in chapters 2 and 4. Indeed, as extensions of Silo, they suffer from the same invalid assumptions for small networks.

In this chapter, we observe that low-cost programmable devices also provide great opportunities for predictable performance, *because* they are simple. For example, the Zodiac FX runs a single-threaded OS-free packet processing loop. More expensive devices, such as carrier-grade switches, typically rely on multi-core architectures and OSs with complex schedulers and optimizations [KPK14]; [KPK15]; [Kuź+18], making it difficult to devise models: a prerequisite for predictability. Another opportunity, besides architectural simplicity, comes from the fact that low-cost programmable switches are often based on *open* architectures, in contrast to high-end switches that have black-box architectures. As we show, this allows us to derive fundamental benchmarking dimensions. Besides, industrial applications that demand predictability typically impose relaxed bandwidth (up to hundreds of kilobits per second) and latency guarantees on the order of milliseconds [Kat+17a], which can potentially be achieved by low-capacity hardware.

The main contribution of this chapter is the design, implementation, and evaluation of *Loko*, a system providing E2E latency guarantees for networks based on low-cost programmable switches serving token-bucket traffic patterns. *Loko* relies on a measurement-based approach to derive accurate performance models for such switches, and manages the network accordingly in order to ensure deterministic latency. Our approach to design *Loko* leverages principles of DNC, and proceeds in three steps: First, through a profound measurement campaign, we derive the necessary parameters for modeling switching performance, leveraging knowledge of the (open) architecture of low-cost devices. Second, based on these measurement inputs, we construct a switch model that avoids traditional assumptions that are invalid for low-cost devices. Third, we extend the switch model to a network model, which forms the basis for the design of admission control and resource allocation strategies enabling *Loko* to provide latency guarantees. This model can be used as part of the DetServ architecture defined in chapter 2 to replace the TBM, invalid for small networks. We evaluate *Loko* in a real testbed using a proof-of-concept implementation with Zodiac FX switches. Our experiments confirm the correctness and applicability of our approach and its underlying models: our testbed measurements show that the guarantees provided by *Loko* are indeed not violated. We observe predictable E2E latencies, including guaranteed throughput, guaranteed packet delivery and burst allowance.

This chapter tackles the challenge of providing predictable latency guarantees for token-bucket traffic patterns with low-cost and hence low-performance (and presumably less reliable) devices. Succinctly, the contributions of this chapter are:

- We demonstrate the limitations of existing performance models in the context of low-cost and low-capacity programmable switches. To this end, we pinpoint the assumptions taken by SoA approaches, including the TBM of chapter 2, which are invalid in this context.
- We present the first measurement-based methodology to realize networks providing deterministic QoS guarantees. Our approach relies on DNC principles. We also give insights on the performance of low-cost programmable switches.
- We design, implement, and evaluate *Loko*, a system based on the derived models and resource allocation algorithms which provides latency guarantees for small-scale programmable devices serving token-bucket traffic patterns and that follows the DetServ architecture defined in chapter 2.



IS *Microchip KSZ8795CLX*
1× 1000 Mbps data port (to CPU)
4× 100 Mbps data ports

CPU *Atmel SAM4E8CA*
1× 100 Mbps port (to IS)
120 MHz single-core
128 KB of RAM
512 KB of prog. flash memory

Figure 5.1: Physical layout of the Zodiac FX and main specifications of its integrated switch (IS) and CPU.

- Using operational traces from a world-leading industrial network operator, we confirm the practical value of *Loko*: it can satisfy delay and bandwidth requirements of existing industrial applications using low-cost hardware.

In order to ensure reproducibility, we have released our research artifacts (data sets, traces, configuration files and source code) at [Van19a].

Content and outline of this chapter. We present an empirical motivation for our work in Sec. 5.1. We then introduce our measurement-based methodology (Sec. 5.2) and derive a switch performance model accordingly (Sec. 5.3). In Sec. 5.4, we describe the *Loko* system, its network model and how it fits in the DetServ architecture, and we report on results of our proof-of-concept measurements in Sec. 5.5. We discuss the generality of our solution and its applicability in Sec. 5.6. Finally, we summarize and conclude the chapter in Sec. 5.7. This chapter relies on the content of [Van+19b].

5.1 Motivation: SoA Falls Short for Low-Cost Devices

We start with an empirical motivation showing the shortcomings of SoA performance models when applied to networks based on low-cost and hence low-capacity programmable switches. As a case study throughout the chapter, we will consider the $70 Zodiac FX switch, which is archetypical for switches used in such networks, e.g., the Banana Pi R1 and R2 [Sin18a]; [Sin18b].

The Zodiac FX relies on a CPU for packet processing. Such kind of architecture, also used by the $90 and $125 Banana Pi R1 and R2, is typical for low-cost programmable devices. Indeed, such devices do not have the ability to build a programmable processing into the switch chip, as can easily be done for switches implementing only a static behavior, e.g., L2 switching. As a result, the only option is to use a CPU for processing.

We will then show that SoA systems [Gro+15]; [Jan+15] (see Sec. 2.1) providing latency guarantees are a poor match for such architectures, and provide an analysis for the underlying reasons. We find that such architectures invalidate several assumptions, as *(a)* in contrast to high-end devices, packets cannot be processed at line rate; and *(b)* the switching capacity is shared among ports, thereby leading to inter-port interferences which are ignored for high-end devices.

```
 1: while true do
 2:     processFrame()
 3:     processCLI()
 4:     protocolTimers()
 5:     checkOFConnection()
 6:     if +500 ms since last OFChecks() then OFChecks()
 7: function processFrame()
 8:     if packet from native port then
 9:         if HTTP packet then sendToHttpServer()
10:         if OF packet then sendToOFAgent()
11:     if packet from OF port then sendToOFPipeline()
```

Figure 5.2: Main loop of the Zodiac FX firmware.

5.1.1 Hardware Architecture

The Zodiac FX is equipped with four 100 Mbps Ethernet ports connected to an integrated 5-port L2 switch (Fig. 5.1) with a 64 KB internal frame buffer (composed of 512 buffers of 128 bytes) shared among all ports. The fifth port of this IS is connected through a 100 Mbps link to an ARM Cortex-M4 single-core 120 MHz micro-controller (CPU) with 128 KB of RAM. The IS and the CPU are further connected through an out-of-band universal synchronous/asynchronous receiver/transmitter (USART) interface (not shown in Fig. 5.1) to allow the CPU to configure the forwarding behavior of the IS (Sec. 5.1.2.1) and to fetch status/statistics information (Sec. 5.1.2.2). The flash memory of the CPU, storing the firmware, can be programmed through a universal serial bus (USB) connector which also serves as a power input. The switch consumes up to 200 mA, making it perfect for small-scale, low-energy environments.

5.1.2 Firmware Architecture

The Zodiac FX ships with an open-source firmware supporting OF versions 1.0 and 1.3 [Nor18]. We focus on OF version 1.0 and on version 0.84 of the firmware.

5.1.2.1 Behavior of the Integrated Switch (IS)

Through the USART interface, the CPU configures the IS during boot-up based on a configuration stored in electrically erasable programmable read-only memory (EEPROM). The firmware distinguishes *native* and *OF* ports — *native* ports are (management and) CP ports and *OF* ports are DP ports. If a port is configured as a *native* port, the IS processes the packets using its internal L2 switching engine. If a port is configured as an *OF* port, the IS directly sends the packets to the CPU with a 1-byte tail tag identifying the port where the packet came from. By default, ports 1–3 (the three leftmost ports in Fig. 5.1) are *OF* ports and port 4 is a *native* port. The fifth port (towards the CPU) is always configured as *native*.

5.1.2.2 Behavior of the CPU

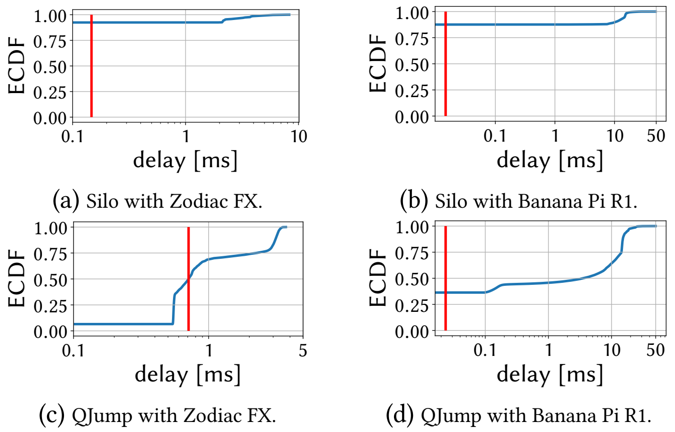
After configuring the IS, the CPU runs the single-threaded infinite loop shown in Fig. 5.2. The processFrame() function (line 2) processes, if present, one Ethernet frame. If the frame comes from an *OF* port, it is forwarded to the *OF pipeline*. If the frame comes from a *native* port, it is forwarded, based on the L4 destination port, either to the *OF agent*, or to an HTTP server hosting a user interface. If the CPU sends a packet coming from the *OF agent* or the HTTP server, it is sent through the normal L2 switching engine of the IS. If the packet comes from the *OF pipeline*, the output port is defined by the *OF pipeline* through a 1-byte tail tag appended to the packet. The processCLI() function (line 3) processes, if present, a command sent via USB on the command line interface (CLI). Both processFrame() and processCLI() functions are non-blocking and return only when processing is completed. The protocolTimers() function (line 4) handles the timers of the TCP/IP stack and the checkOFConnection() function (line 5) handles the OF connection. Finally, the OFChecks() function (line 6) alternates between updating the port statistics, updating the status (up/down) of ports (both through the out-of-band connection) and checking entries timeouts (each one executed at most every 1500 ms).

OpenFlow agent. We detail here the flow table management behavior of the OF agent (line 10). The table is stored as an ordered list of up to 128 entries. Upon receipt of a *FlowMod Add* message, the new flow entry is stored directly at the end of the table. Upon receipt of a *FlowMod Delete* message, the agent goes through all entries one by one. If an entry matches the flow deletion request, it is deleted and the table is *consolidated* by replacing the removed entry with the last entry in the table. The process upon receipt of a *FlowMod Modify* message is similar, except that matching flow entries are overwritten with the new received flow.

OpenFlow pipeline. The DP processing logic (line 11) goes through the flow entries one by one. If a flow entry matches, only subsequent entries with higher or equal priority are checked. While checking if an entry matches the incoming packet, all fields belonging to the match structure are considered, independently of whether another field in this match structure matched or not. If no flow entry matches, a *PacketIn* message is sent to the controller. Otherwise, the counters of the highest priority matching entry are updated and the corresponding action(s) is (are) performed.

5.1.3 Why Does SoA Fail?

We demonstrate the need for a system able to provide guarantees for low-cost programmable switches by deploying two SoA systems for guaranteed latency, QJump [Gro+15] and Silo [Jan+15], on the Zodiac FX and Banana Pi R1 switches and by showing that these systems fail to provide their latency guarantees. We choose these two solutions as the two only existing SoA systems for strict latency guarantees without hardware modifications (see Sec. 2.1). We consider a topology of two switches, both of which have two hosts attached. Each host sends traffic to its symmetrical counterpart on the other switch. For making the experiment comparable to the final evaluation of Loko (Sec. 5.5), we consider 306-byte packets and configure 17 flow entries on the switches (4 used ones and 13 dummy ones). For traffic shaping, we use the *tc* Linux utility with its *tbf* queuing discipline [Lin]. We observe the packet delays of two of the four flows for 20 runs of 10 seconds using an Endace DAG 7.5G4 card [End16].



(a) Silo with Zodiac FX.

(b) Silo with Banana Pi R1.

(c) QJump with Zodiac FX.

(d) QJump with Banana Pi R1.

Figure 5.3: State-of-the-art approaches fail at providing their guarantees for low-cost devices. The vertical red line corresponds to the guaranteed latency, zero delay corresponds to packet loss.

Silo. The guarantees provided by Silo [Jan+15] are based on an admission control scheme. In our scenario, with Zodiac FX switches, Silo would, for example, allow each host to send traffic at a rate of 45 Mbps, with a maximum burst of 306 bytes and would provide a 146.9 μs latency guarantee for these flows. We describe how we compute these values in appendix Sec. 5.8. Fig. 5.3a shows that Silo fails at providing its guarantees for the Zodiac FX: for this amount of traffic, the switches drop 92.4% of the packets and 7.6% of the packets arrive delayed. For the Banana Pi R1, we show in the appendix that Silo would allow each host to send traffic at a rate of 450 Mbps, with a maximum burst of 306 bytes and would provide a latency guarantee of 14.7 μs. Fig. 5.3b shows that sending this amount of traffic leads to 87.5% of lost packets and 12.5% of delayed packets: Silo also fails at providing its guarantees for the Banana Pi R1.

QJump. QJump [Gro+15] guarantees a maximum latency of $2nP/R+\epsilon$ – where n is the number of hosts, P denotes the packet size, R represents the link rate, and ϵ refers to the cumulated processing time – if all hosts send at most one packet during this time period, i.e., at a rate of at most $P/(2nP/R+\epsilon)$. We have n =4 and P =306 bytes. For the Banana Pi R1, R =1 Gbps. The ϵ parameter is not known. If we use the value in [Gro+15], i.e., 4 μs, Fig. 5.3d shows that QJump fails at providing its guarantees: 36.3% of the packets are lost and 63.7% are late. This shows that a proper modeling of the processing time of the switches is needed in order to determine ϵ. This is the gap we address in this chapter (Sec. 5.2) for the Zodiac FX. For the specific case considered (packet size, number of entries) and with at most two switches between any pair of hosts, we have $\epsilon = 2 \times p_{\mathrm{FX}} = 257$ μs. Fig. 5.3c shows that, even with this ϵ modeling, QJump fails: 6.5% of packets are lost and 49.9% are late.

Failure reasons. With the previously described hardware architecture in mind, we advocate the following explanations for the failure of Silo and QJump. First, both approaches assume that switches can process packets at line rate. For carrier-grade switches, that is usually correct. For example, the Dell S4048-ON switch provides a 1080 Mpps throughput [Del] and at most 48 ports ×10 Gbps/64 bytes =938 Mpps can be sent on its input ports. With four 100 Mbps ports, the Zodiac FX can receive up to 0.781 Mpps but its throughput can go down to 0.3 Mbps (Sec. 5.2.4), i.e., as low as 586 pps. Similarly, with four 1 Gbps ports, the Banana Pi R1 can receive up to 7.81 Mpps

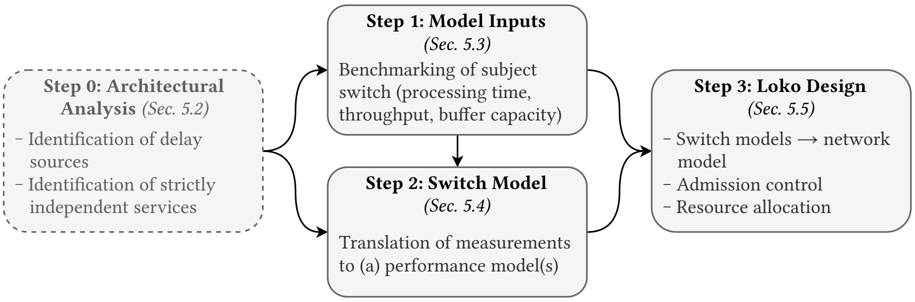

Step 0: Architectural Analysis (Sec. 5.2)
- Identification of delay sources
- Identification of strictly independent services
Step 1: Model Inputs (Sec. 5.3)
Benchmarking of subject switch (processing time, throughput, buffer capacity)
Step 2: Switch Model (Sec. 5.4)
Translation of measurements to (a) performance model(s)
Step 3: Loko Design (Sec. 5.5)
- Switch models → network model
- Admission control
- Resource allocation

Figure 5.4: Our approach for the design of Loko.

but, through a setup similar to Sec. 5.2.4, we evaluated its throughput at around 645 Mbps for 1470-byte packets, i.e., as low as 59 kpps. Hence, the Zodiac FX and the Banana Pi R1 cannot always process packets at line rate and must buffer at the ingress. Second, because of the centralized CPU of small-scale programmable switches, the processing of packets from a given port can interfere with other ports. Because carrier-grade switches can process packets at line rate, Silo and QJump assume independent services for each port, while, seemingly, a shared per-switch service definition is required for low-cost programmable switches.

To summarize, the assumptions of existing approaches that turn out to be erroneous for low-cost programmable switches are:

- *Assumption 1.* Switches can process packets at line rate and hence queuing happens mostly at the egress.
- *Assumption 2.* Ports do not interfere with each other.

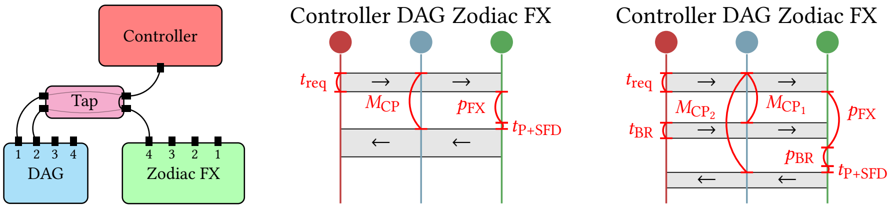
Therefore, our approach for *Loko* proceeds in three steps (cf. Fig. 5.4). First (Sec. 5.2), to avoid *Assumption 1*, we comprehensively evaluate the performance of a given low-cost programmable switch. Second (Sec. 5.3), we use our measurements results to derive a shared per-switch forwarding performance model based on DNC, thereby avoiding *Assumptions 1 and 2*. Finally, in Sec. 5.4, we describe the overall *Loko* system using the switch performance model to design a network-wide model for predictable latency with resource allocation and admission control that fits within the DetServ architecture described in Sec. 2.3.

5.2 Step 1: Switch Benchmarking

To realize predictable performance, *Loko* leverages DNC concepts. DNC modeling requires the determination of the worst-case processing time and throughput, for delay computation, and of the buffer size for packet loss prevention (see Sec. 2.2). In this section we present a measurement-based methodology, along with its results, to determine these switch performance parameters.

5.2.1 Ensuring Deterministic Performance

First, we must ensure that all influential factors can be gathered as benchmarking dimensions. In this section, we cover the influencing factors that cannot be controlled and must be deactivated. As



(a) CP setup. (b) With reply. (c) No reply.

Figure 5.5: (a) CP processing time measurement setup and corresponding sequence diagrams for CP messages (b) with reply and (c) without reply.

a single CPU processes all received packets, for CP (resp. DP) processing, we make sure that no DP (resp. CP) packets are sent and that the CLI and HTTP server are not used.

The CHECKOFCONNECTION() function (line 5) sends *EchoRequest* messages if no CP messages were exchanged for *n* seconds. We set *n* to infinity and consider that the controller checks the liveness of the OF connection.

The OFCHECKS() function (line 6) performs three different operations every 500 ms (Sec. 5.1.2.2). Fig. 5.6a shows the processing time of an *EchoRequest* over time. We observe spikes every 1.5 s, corresponding to the time needed for the IS to return port statistics updates to the CPU. Port status updates are also fetched from the IS, thus are costly as well. In order to prevent these interferences, we completely disable the OFCHECKS() function. First, port statistics are not needed, as our architecture is solely based on a centralized admission control strategy that keeps the network state in the CP (see Sec. 2.3). Second, we assume a static network topology, thereby not needing port status updates. Finally, flow timeout management is also not needed. Indeed, flow entries are proactively and completely managed by a controller.

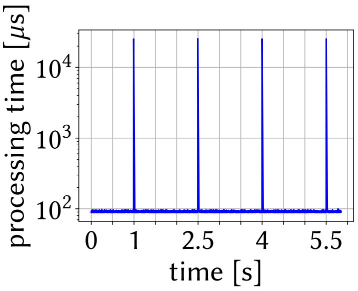
The PROTOCOLTIMERS() function (line 4) hence remains the only source of interference. Its impact will be considered by taking the worst-case value among our samples.

5.2.2 Control Plane Processing Time

The architecture of low-cost programmable switches (Sec. 5.1.3) leads to interferences between the processing of CP and DP packets. We first consider the CP, an essential component of any programmable network. We describe in this section our measurement-based methodology for determining CP processing times and report on its results for our case study.

5.2.2.1 Setup

A *Ryu*-based [Ryu17] controller is connected to the Zodiac FX and a network tap mirrors the frames of this connection to an Endace DAG 7.5G4 measurement card [End16] which timestamps packets upon arrival of the SFD [Don02] (Fig. 5.5a). We construct two measurement procedures: for CP messages with reply (e.g., *EchoRequest*) and for CP messages without reply (e.g., *FlowMod Add*).



(a) Native code impact.

OF 1.0 Message	**avg. ± std. dev.** [μs]
EchoRequest	90.91 ± 1.71
BarrierRequest	91.27 ± 1.74
FeatureRequest	295.55 ± 1.76
DescStatsRequest	187.64 ± 1.98
GetConfigRequest	91.92 ± 1.60
PortStatsRequest	125.70 ± 1.93
AllPortStatsRequest	173.93 ± 1.94
FlowStatsRequest*	297.80 ± 65.20

(b) CP processing times.

Figure 5.6: (a) *EchoRequest* processing time with native code and (b) processing time of CP messages except *FlowMod*.
* indicates dependency on the number of flow entries and actions.

CP messages with reply. Here, the controller simply sends a CP message and its processing time p_{FX} can be obtained from M_{CP}, the time difference between the DAG timestamps (see Fig. 5.5b), as

$$p_{\mathrm{FX}} = M_{\mathrm{CP}} - t_{\mathrm{req}} - t_{\mathrm{P+SFD}}, \tag{5.1}$$

where t_{req} is the computed transmission time of the request and $t_{\mathrm{P+SFD}}$ is the computed transmission time of the Ethernet preamble and SFD (8 bytes) sent before the response.

CP messages without reply. Here, we send an additional *BarrierRequest* directly after the subject CP message (see Fig. 5.5c). This way, the processing time p_{FX} of the subject CP message can be obtained from the measured delay M_{CP_2} until the *BarrierReply* is received as

$$p_{\mathrm{FX}} = M_{\mathrm{CP}_2} - t_{\mathrm{req}} - p_{\mathrm{BR}} - t_{\mathrm{P+SFD}}, \tag{5.2}$$

where t_{req} and $t_{\mathrm{P+SFD}}$ are computed and p_{BR} corresponds to the measured processing time of a *BarrierRequest*. This is only valid if the *BarrierRequest* is received by the Zodiac FX before it finished processing the subject CP message, i.e.,

$$M_{\mathrm{CP}_1} + t_{\mathrm{BR}} < M_{\mathrm{CP}_2} - p_{\mathrm{BR}} - t_{\mathrm{P+SFD}}. \tag{5.3}$$

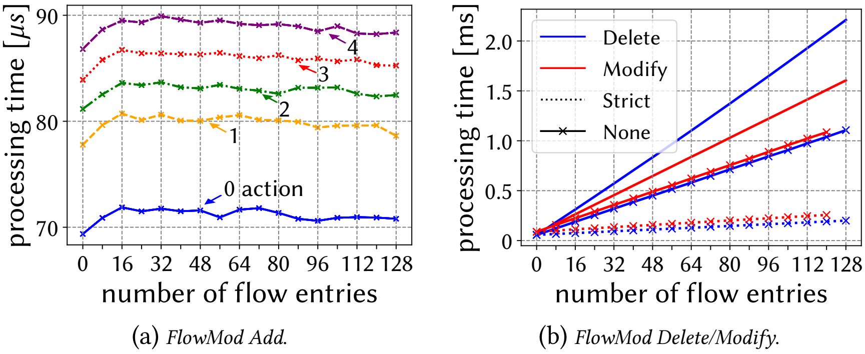
To ensure this, we implement a Linux *tc* queuing discipline that delays OF CP messages without reply (e.g., *FlowMod*) until a subsequent *BarrierRequest* is sent.

5.2.2.2 Scenario

For the *FlowMod* and *FlowStatsRequest* messages, we consider flow tables with 1 to 128 entries and 0 to 4 actions per entry. For other messages, based on our analysis of the OF agent implementation, we consider an empty flow table because the processing time is independent of the switch state. We gather 100 samples for each CP message. For *FlowMod* and *FlowStatsRequest*, we gather 100 samples for each combination of the numbers of entries and actions.

5.2.2.3 Results

Fig. 5.6b shows that the processing time of the Zodiac FX for CP packets is very stable: less than 2 μs of standard deviation. The higher variation for the *FlowStatsRequest* message is due to its dependency on the numbers of flow entries and actions considered. Surprisingly, the Zodiac FX actually



(a) *FlowMod Add.*

(b) *FlowMod Delete/Modify.*

Figure 5.7: Average (100 samples per point) processing time of *FlowMod Add/Delete/Modify* messages.

outperforms carrier-grade devices in some cases. For instance, it needs around 92 μs to process a *BarrierRequest* message, while the Pica8 P-3290 and Dell 8132F switches need 100–700 μs [KPK14].

FlowMod Add/Delete/Modify. Fig. 5.7a shows that the number of flow entries has no significant impact on the average processing time of *FlowMod Add* messages: the Zodiac FX always directly adds new entries at the end of the table. The processing time is only influenced by the number of actions, as a higher number of actions requires copying more data into memory. For *FlowMod Delete/Modify* messages, we consider several cases: with or without the *strict* option and deleting/modifying all (lines without markers) or *none* (lines with markers). Fig. 5.7b shows that, in general, the average processing time increases linearly with the number of entries, reaching up to 2.3 ms for deletion. We further observe that requests with the *strict* option are processed faster than without. This is due to the fact that, for *strict* deletion/modification, matches only have to be compared bitwise, while without the *strict* option, more costly masking operations are required. Further, deleting/modifying all flow entries requires more time than *none*, as the switch additionally has to perform the deletion (i.e., consolidate the table) or modification for each flow entry. The consolidation operation appears more costly than the modification, as for this case, processing a *Delete* request takes more time than a *Modify* request. However, when *none* of the entries match, the *FlowMod Modify* message requires the switch to add the entry, which in this case, leads to a slightly higher processing time for *Modify* requests.

Outcomes. Given the knowledge of the Zodiac FX state, the processing time of CP messages is highly predictable, allowing us to deterministically model it and include it in our shared per-switch model.

5.2.3 Data Plane Processing Time

As processing time is not negligible for small-scale programmable switches (Sec. 5.1.3), an important step towards the computation of worst-case switch traversal times is to precisely quantify the processing time of DP packets. We present our detailed and comprehensive measurement methodology and report on its results and insights.

Dimension	Values
nb. of entries	1, 17, 33, 49, 65, 81, 97, 113, 128
match type	*port*, *tp-dst*, *dl-dst*, *masked-nw-dst*, *five-tuple*, *all*
action	*output*, *set-vlan-id*, *set-vlan-pcp*, *strip-vlan*, *set-dl-src*, *set-nw-src*, *set-nw-tos*, *set-tp-src*
used entry	*first*, *last*
priorities	*increasing*, *decreasing*
packet size	64, 306, 548, 790, 1032, 1274, 1516

Table 5.1: Considered dimensions for the DP processing time.

5.2.3.1 Evaluation Dimensions

Based on the knowledge of the OF pipeline implementation (Sec. 5.1.2.2), we identify all the parameters influencing the processing time of the Zodiac FX. We define them below, with Tab. 5.1 reporting the full lists of considered values.

- *number of entries*. More entries means more comparisons. The maximum number of entries is 128 (see Sec. 5.1.2), so we explore values from 1 to 128 by steps of 16.

- *match type*. Since only fields belonging to the match structure are checked, the number and type of fields in the match structure impact the processing time of the Zodiac FX. In addition to the *port*, *tp-dst*, *dl-dst* and *masked-nw-dst* match types, we consider the *five-tuple* (*ip-src*, *ip-dst*, *tp-src*, *tp-dst*, *nw-proto*) and *all* (*five-tuple*, *in-port*, *nw-tos*, *dl-src*, *dl-dst*) combinations. Because all fields of the match are always checked, the *way* in which an entry does or does not match (e.g., how many fields fail) has no influence.

- *action*. Besides the single *output* action, we consider different modification actions followed by the *output* action.

- *used entry*. Because the switch can avoid checking flow entries if a match was already found, the position of the matching entry can have an impact. We consider cases with only one matching entry: the *first* or the *last* one.

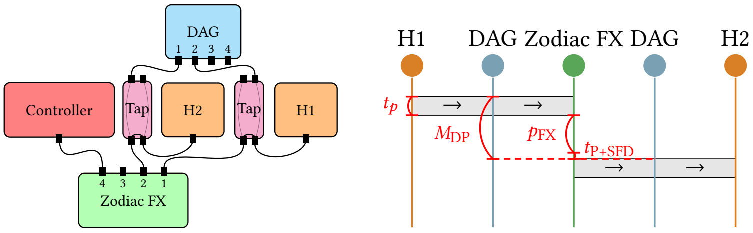
- *priorities*. We consider two different orderings of flow entries: *increasing* and *decreasing* priorities. In the former case, all flow entries will be checked, while in the latter, entries are not checked anymore as soon as an entry matches.

- *packet size*. Many components of delay in a switch are likely to be proportional to packet size [KS98].

Because of the centralized CPU architecture, the simultaneous usage of several ports (including the CP port) also impacts processing time. This will be taken into account in our model by defining a shared per-switch service (Sec. 5.3). We hence do not include it in our processing time measurements.

5.2.3.2 Setup

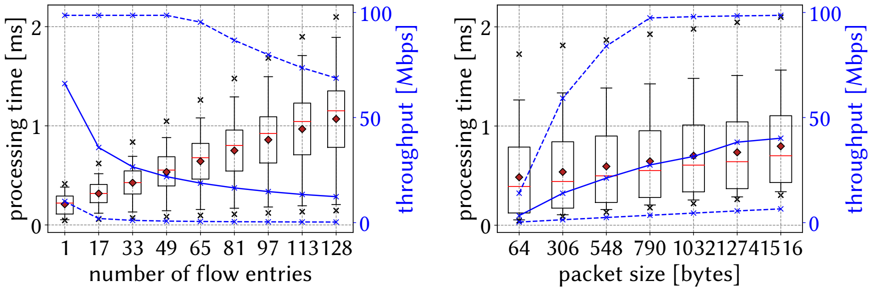
A *Ryu*-based [Ryu17] controller generates a flow table according to the selected values of the dimensions. The matching flow entry is configured to forward to port 2. Using *scapy* [BS18], Host 1 (*H1*) sends packets with the appropriate header fields and packet size. Packets coming in (port 1) and out (port 2) of the Zodiac FX are then mirrored using two network taps to the Endace DAG 7.5G4 measurement card (Fig. 5.8a). The processing time p_{FX} of the switch can be obtained from the measured



(a) DP setup.

(b) DP sequence diagram.

Figure 5.8: (a) DP processing time measurement setup and (b) corresponding sequence diagram.



(a) Number of flow entries

(b) Packet size

Figure 5.9: Processing time and throughput of the Zodiac FX based on the number of entries (left) and packet sizes (right).

M_{DP} by subtracting the computed transmission time t_p of the packet (Fig. 5.8b), i.e.,

$$p_{FX} = M_{DP} - t_p - t_{P+SFD}. \tag{5.4}$$

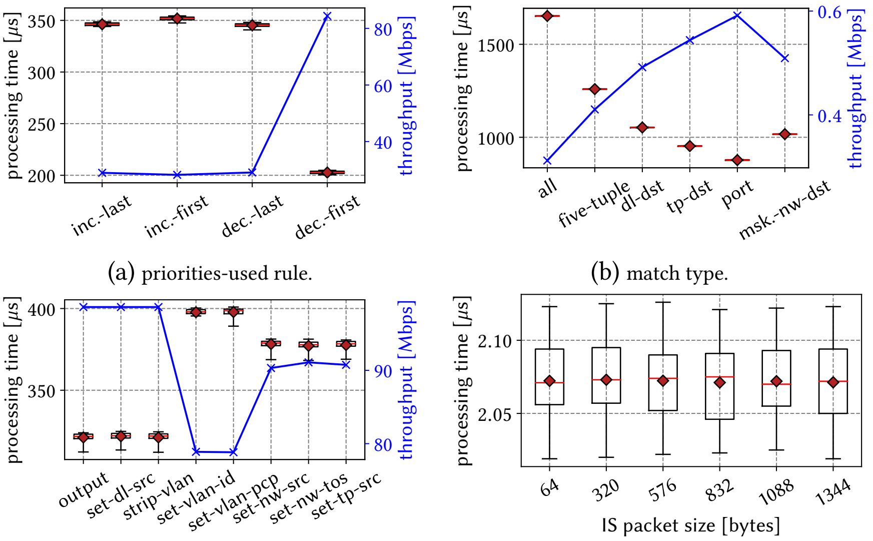
For each possible combination of the different dimensions in Tab. 5.1, we measure the processing time of 100 packets in order to reach sufficient statistical significance.

5.2.3.3 Results

The results are represented as boxplots in Figs. 5.9 and 5.10. Whiskers show the 5% and 95% percentiles and the minimum and maximum values are shown as crosses. The figures also show the throughput values covered in Sec. 5.2.4.

Number of entries. Fig. 5.9 shows that the processing time increases linearly with the number of entries. In order to show the whole range of processing time values achieved by the switch, all the other dimensions are aggregated in the boxplots. We see that the processing time of the Zodiac FX ranges from around 50 μs to 2.1 ms.

Packet size. Similarly, because of memory copy operations, the measured processing time increases linearly with the packet size (Fig. 5.9). We observe that the increase is smaller than for the number of entries, i.e., the latter has a higher impact. For a similar packet size range, the Pica8 P-3297, Dell S4048-ON and NEC PF-5240 carrier-grade switches have a processing time of around 3–5 μs (see Sec. 4.1.2.1). Compared to them, the Zodiac FX performs poorly: up to three orders of



(a) priorities-used rule.

(b) match type.

(c) action.

(d) IS packet size [bytes].

Figure 5.10: (a)–(c) Processing time and computed throughput of the Zodiac FX and (d) processing time of the IS. (a) *17* entries, *five-tuple* matching, *790*-byte packets and *output* action, (b) *128* entries, *increasing* priorities, *last* entry used, *64*-byte packets and *output* action, (c) *1* entry, *port* matching, *decreasing* priorities, *last* entry used, *1516*-byte packets and *output* action.

magnitude slower. This is in line with our motivational experiment of Sec. 5.1.3: processing time is not negligible for low-cost devices.

Used entry and priorities. For a single selected case within our dimensions, Fig. 5.10a shows that the processing time is the lowest when the priorities are *decreasing* and the *first* entry matches. In this case, a full comparison against the other entries is not necessary. For all other cases, the switch compares the packet against each entry in the table. Compared to Fig. 5.9, other dimensions are not aggregated. We see that, in this case, the switch performance is highly predictable: the processing time variance is negligible.

Match type. Naturally, as the match structure becomes larger, the processing time increases (Fig. 5.10b). For instance, for this selected case, *port* matching requires around 0.88 ms and *all* around 1.65 ms. We again observe that the switch performance is predictable for a specific investigated case.

Action type. Actions that require the re-computation of L3/L4 checksums are slower (Fig. 5.10c). For instance, for this selected case, *set-nw-src* requires 380 μs while the simple *output* action is the fastest with around 322 μs. We observe that the action type has a much lower impact than the match type: the match type influences the time needed to check each entry, while the action is only executed once. As for Fig. 5.10a and 5.10b, we observe that the switch performance is highly predictable.

Outcomes. Our results show that, for a single case among our dimensions, the processing time of the Zodiac FX is very stable, enabling us to precisely and deterministically model the DP performance. We also see that, for different cases, the performance of the Zodiac FX can highly vary.

Finally, we observe that the processing time of the switch creates a potential for satisfying the latency requirements of typical industrial applications, which are in the order of milliseconds [Kat+17a].

5.2.4 Data Plane Throughput

As we have shown in Sec. 5.1.3, small-scale programmable devices are presumably not able to process packets at line rate. In this section, we present our methodology to quantify the actual rate at which packets are processed. We detail how this rate can be computed based on the processing time (Sec. 5.2.3), demonstrate that this computation is indeed correct, and give insights on the throughput achieved by the Zodiac FX.

5.2.4.1 Mathematical Computation

Generally, throughput TP can be computed from packet size l_p and packet processing time p_{FX} through

$$TP = l_p / p_{\mathrm{FX}}. \tag{5.5}$$

However, if the switch is able to process several packets simultaneously, e.g., through a pipeline, Eqn. 5.5 becomes a lower bound on the throughput. The Zodiac FX forms a pipeline composed of the IS, the link IS–CPU, and the CPU; it can hence process packets simultaneously. The throughput of the Zodiac FX then corresponds to the throughput of the bottleneck element in the pipeline. Hence, we determine the throughput of these three elements.

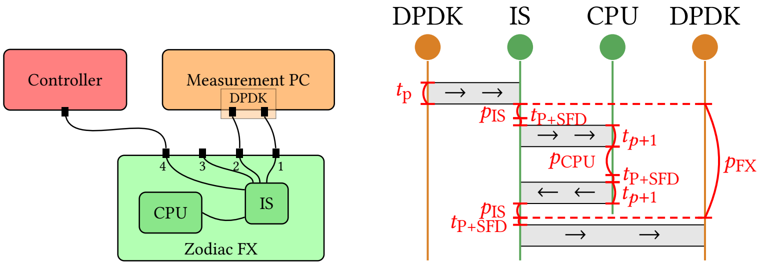
Through a setup identical to Fig. 5.8a, we measure the processing time of the IS (Fig. 5.10d) by configuring it in L2 learning mode, hence not using the CPU. The results show a stable processing time independent of the packet size of $p_{\mathrm{IS}} = 2.07\ \mu$s on average. The IS is never the bottleneck. Indeed, although it is traversed twice in the pipeline, its maximum processing time corresponds to a minimum throughput (through Eqn. 5.5) which is greater than twice the throughput of the link IS–CPU[1]. The latter is given by $l_p/(l_p + 21) \times 100$ Mbps, as the link has to transport, besides the p-byte packet, the 1-byte tail tag, the preamble (7 bytes), SFD (1 byte) and interframe gap (IFG) (12 bytes). The throughput of the CPU can be computed with Eqn. 5.5. As a result, the throughput can be computed as

$$TP = \min\left\{l_p / p_{\mathrm{CPU}}, l_p/(l_p + 21) \times 100 \text{ Mbps}\right\}, \tag{5.6}$$

where

$$p_{\mathrm{CPU}} = p_{\mathrm{FX}} - 2p_{\mathrm{IS}} - 2p_{\mathrm{P+SFD}} - 2t_{p+1} \tag{5.7}$$

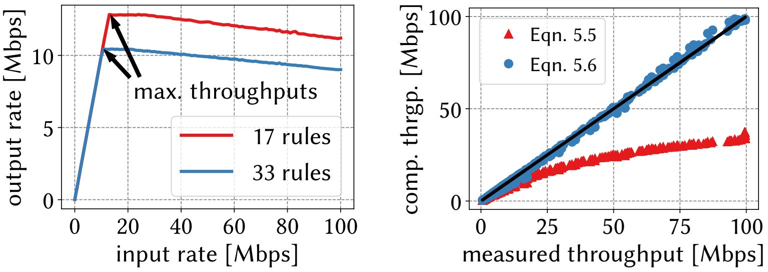
is inferred from Fig. 5.11b and t_{p+1} corresponds to the transmission time of a $(l_p + 1)$-byte packet. We measured p_{IS} in L2 learning mode, which is slower than when it is used with the Zodiac FX switch [Mic17]. Hence, to avoid taking any too optimistic assumption for throughput computation, we neglect this term in Eqn. 5.7. p_{FX} is obtained from Sec. 5.2.3.



(a) DP throughput setup.

(b) Sequence diagram.

Figure 5.11: (a) Setup for the measurement of DP throughput and (b) corresponding sequence diagram.



(a) Finding max. throughput.

(b) Verification of Eqn. 5.5 & 5.6.

Figure 5.12: (a) Output rate based on the input rate for *five-tuple* matching, 64-byte packets, *decreasing* priorities, the *first* entry matching and 17 or 33 entries. (b) Measured throughput compared to Eqn. 5.5 & 5.6.

5.2.4.2 Empirical Verification

Setup. We use DPDK [DPD18] and our own modified version of its *pktgen* application to generate traffic on one port and to log the received throughput on another other port (Fig. 5.11a)[2]. Fig. 5.12a shows the output rate of the Zodiac FX for two specific cases. When sending not more than the maximum throughput, the CPU is fast enough to process all packets, and the output rate equals the input rate. Interestingly, when the transmission exceeds the maximum throughput, the output rate reduces linearly. This is because, the CPU sends pause frames if it cannot process all the packets. As a result, the IS starts buffering the packets. As such, the buffer at the IS grows, and packets sent back by the CPU might be dropped, decreasing the throughput. As a result, we use a binary search in order to find the maximum input rate that can be processed, i.e., to find the maximum of the curves in Fig. 5.12a. Due to the precision of the DPDK sending rate and statistics reports, we use 650 kbps as the precision for the binary search.

Evaluation dimensions. For the sake of brevity, we only consider the *output* action, the *five-tuple* and *port* matchings and the *increasing-last* and *decreasing-first* priorities and matched entry combinations. The numbers of flow entries and packet sizes of Tab. 5.1 are all considered.

[1] This is not true for very small packets (e.g., 48 bytes). However, in this case, the throughput of the CPU is lower and the IS is still not the bottleneck.

[2] The IS is by default configured with “half-duplex back-pressure collision flow control” [Mic17], a L2 mechanism instructing a device to reduce its sending rate if congestion happens. To prevent this from forcing DPDK to throttle, we deactivate this feature on the IS.

Results. Fig. 5.12b shows that Eqn. 5.5 underestimates the actual throughput and that the error increases with the throughput. On the other hand, we observe that Eqn. 5.6 corresponds closely to the actual throughput. The relative error remains below 6%.

5.2.4.3 Results

We use Eqn. 5.6 for all cases of Tab. 5.1. The results are shown as blue curves in Fig. 5.9 and Fig. 5.10a–c. In Fig. 5.9, several cases are aggregated. The corresponding minimum and maximum throughputs are shown with dashed lines and the average throughput with a full solid line. Fig. 5.9 shows that delivery at line rate can be achieved only for less than 65 flow entries and packets of more than 790 bytes. Depending on the scenario, the throughput varies from less than 1 Mbps to line rate. Fig. 5.10b shows a bad case, as the number of installed entries is high (128) and the packet size is small (64 bytes). We observe that the throughput is very low, i.e., can be as low as 0.3 Mbps. Reversely, Fig. 5.10c shows one of the best cases for throughput, as there is only one rule installed, and packets are big (1516 bytes). In this case, line-rate throughput can be reached only for the *output*, *set-dl-src* and *strip-vlan* actions, while other actions are limited to around 80 Mbps.

Outcomes. We observe that the throughput of the Zodiac FX can greatly vary, from low values (300 kbps) to line rate. This is in line with our experiment of Sec. 5.1.3: for low-cost devices, the assumption that packets are always processed at line rate is not valid. However, these values create a potential for fulfilling the throughput requirements of typical applications with predictable latency requirements, which are usually no more than hundreds of kilobits per second [Kat+17a].

5.2.5 Buffer Capacity

Ensuring no packet loss with DNC concepts requires knowing the maximum number of packets that can be buffered at each switch. This section describes our methodology and applies it to our case study.

Setup. We adopt an approach similar to the one proposed in request for comments (RFC) 2544 [BM99]; [Mor17]. The setup is shown in Fig. 5.8a. We generate packets as fast as possible on port 1 of the switch. The switch is configured by the controller with 128 entries with *five-tuple* matching and *output* action to port 2. As the buffer size surely does not depend on these parameters, we do not vary them. Using the taps, we monitor both ports at packet level. Thereby, we can *(i)* determine, over time, the number of packets backlogged in the switch, and *(ii)* identify when a packet gets lost. The number of backlogged packets when the first packet gets lost is the buffer capacity of the switch.

Results. All obtained results are consistent with the following elaboration. The IS does not buffer packets and forwards data directly to the CPU which has a one-packet receive buffer and 24 buffers of 128 bytes in memory. As a result, the buffer size b_{FX} available at the switch can be computed, based on the packet length l_p, as $b_{\text{FX}} = 1 + \lfloor 24/\lceil l_p/128 \rceil \rfloor$. This value ranges from 3 packets for 1516-byte packets to 25 packets for 64-byte packets.

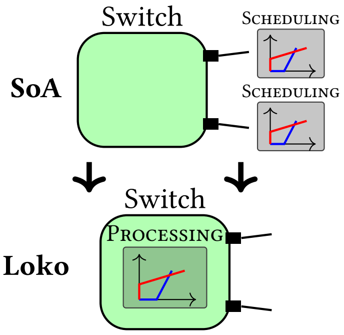


Figure 5.13: On top, SoA approaches modeling independently each port. On the bottom, *Loko*'s approach: the forwarding performance is modeled as a service shared among all ports.

5.3 Step 2: Switch Model

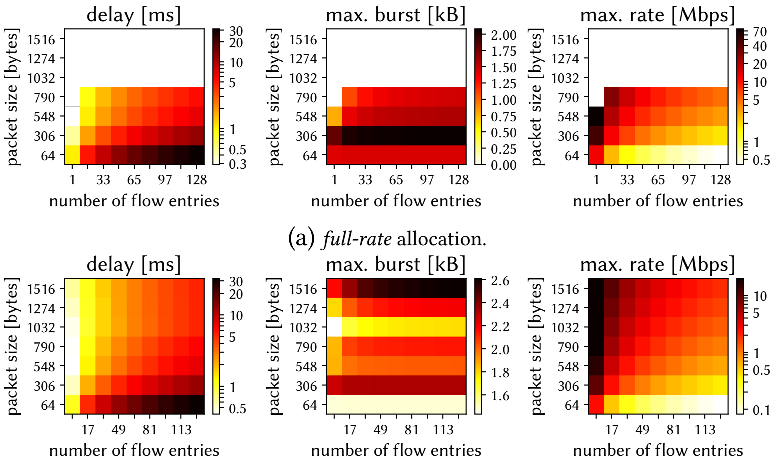
Based on our measurement results, we can now construct an accurate performance model for the switch. In particular, we aim to derive the DNC service curve (see Sec. 2.2.5 and 2.2.8) of the switch. As shown in Sec. 5.1.3, because of their architectures based on a central CPU for packet processing, low-cost programmable switches should be modeled using a single service curve, rather than using independent per-port models, as done by SoA approaches for carrier-grade switches (Fig. 5.13).

We propose to use a $\beta_{R,T}$ rate-latency service curve (see Sec. 2.2.5.2). The R and T parameters intuitively correspond to the worst-case throughput (Sec. 5.2.4) and processing time (Sec. 5.2.2 and Sec. 5.2.3) of the switch for the considered scenario. This is how the model handles varying traffic conditions: the entire space is grasped by considering the worst-case scenario, which has to be determined beforehand based on the given dimensioning of the network. In order to account for per-packet delay and not per-bit delay, l/R (where l is the largest possible packet size) has to be added to the obtained T value (see Sec. 2.2.7). By considering that the service is shared among all the different flows entering the switch, the model automatically takes into account not only inter-port interferences (and hence possible interferences of CP packets) but also buffering inside the switch.

For example, consider our case study of the Zodiac FX. If we investigate a scenario with *five-tuple* match types, *output* actions, packets of 306 bytes, and all other parameters unknown, we have $R = 1.88$ Mbps and $T = 1.35$ ms $+ l/R = 2.65$ ms. Indeed, the processing time and throughput values for 128 flow entries with *increasing* priorities and the *last* entry matching have to be used, as this is the worst case. Note that the worst-case for processing time and throughput can be different. For example, if the packet size is unknown, then the smallest and largest possible packet sizes have to be considered for the throughput and processing time respectively. Note that the processing time and throughput of CP packets also have to be considered for defining the worst-case values.

5.4 Step 3: Network Model

Having established the DNC switch model, we now describe how *Loko* builds on top of it to define a network model and provide E2E latency guarantees. As described in chapter 2 and Fig. 2.1, we consider a proactive scenario where a routing procedure residing in a centralized controller is contacted to find a delay-constrained path for a given flow. We consider the DetServ architecture described in



(a) *full-rate* allocation.

(b) *fifth-rate* allocation.

Figure 5.14: Guaranteed delay and maximum allowed burst and rate at each switch for two resource allocation schemes and combinations of packet size and number of flow entries.

Sec. 2.3. That is, the routing procedure relies on a network model for admission control, obtaining worst-case delay values, resources reservation, and computing cost values for path optimization. In order to avoid having to reroute already accepted flows, the worst-case delays provided by the network model for each switch must be valid for the whole lifetime of the network, i.e., even if other flows are added later on. The admission control mechanism is then responsible for preventing the routing procedure from using a switch if this violates the provided delay bound or if it can lead to buffer overflow.

Following the DetServ architecture, *Loko* achieves this by using a resource allocation algorithm. *Loko*'s resource allocation algorithm assigns maximum allowed token-bucket parameter values (burst size b and rate r) for each switch. In conjunction with the service curves of the switches (see Fig. 2.20), this defines the worst-case delay for each switch. The admission control then rejects a flow if adding it to the currently used token bucket parameters exceeds the allocated maximum values. That requires applications to always comply with their requested burst and rate parameters, which is typically the case for industrial applications [Kat+17a]. The admission control ensures that the per-switch worst-case delays are never violated and valid for the whole lifetime of the network.

For example, consider our case study of the Zodiac FX and a scenario where all flow entries match on *five-tuple* and have a single *output* action. Considering the worst-case of *increasing* priorities with the *last* entry matching, this defines a $\beta_{R,T}$ service curve for each packet size and number of entries combination. Fig. 5.14 shows two different resource allocation strategies (referred to as *full-rate* and *fifth-rate*) and how they lead to different delay, burst and rate values depending on the number of flow entries and packet sizes. In Fig. 5.14a, the full rate R of the service curve is allocated, and the maximum burst is chosen so that no buffer overflow occurs, i.e., the maximum backlog computed through DNC is equal to the buffer capacity. White areas show cases where this is infeasible. That is, while the switch can handle such high throughput, the maximum burst has – in order to avoid buffer overflow – to be lower than the considered packet size, which is not possible. In order to avoid such cases, one can rather assign a fraction of the service curve rate. Indeed, DNC establishes that making the rate r smaller leads to a lower maximum backlog $b + rT$ and hence allows to increase the

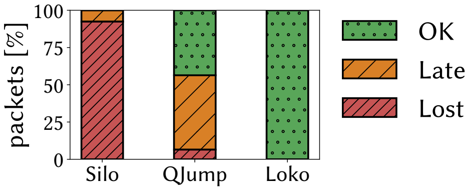


Figure 5.15: Performance of Silo [Jan+15], QJump [Gro+15] and Loko. Only Loko provides predictable latency.

Service curve	Res. all.	max. rate	max. burst	max. delay
$R = 11.8$ Mbps	*full-rate*	11.8 Mbps	2.02 kB	1.86 ms
$T = 0.46$ ms	*fifth-rate*	2.37 Mbps	2.32 kB	2.07 ms

Table 5.2: Loko configuration for the final evaluation.

maximum allowed burst b (see Fig. 2.20). Doing so, the total burst that can be accepted is higher at the price of a lower total maximum rate. Fig. 5.14b shows an exemplary allocation where the maximum rate is fixed to one fifth of the throughput of the switch. The resource allocation algorithm has to make such an a priori decision between delay, buffer and rate at each switch. We later only consider the two strategies shown in Fig. 5.14.

5.5 Loko: Evaluation

In this section, we empirically verify *Loko* and its underlying modeling with a proof-of-concept implementation with the Zodiac FX. We send several traffic flows through a network of switches and verify that guaranteed delay bounds are not violated and that no packets are lost. Fig. 5.15 shows the main result: with a setup identical to the scenario considered in Sec. 5.1.3, Loko successfully provides latency guarantees, while SoA approaches fail. Because DNC is known as a conservative approach, we further quantify the overprovisioning of the model, i.e., how much additional traffic can be sent until delay violations or packet loss actually happen. Finally, through simulations, we show that the network utilization and rejection rates achieved by *Loko* allow to support typical industrial applications with latency requirements and that they scale to network sizes typically seen in industrial scenarios.

***Loko* configuration.** We consider *five-tuple* matching, *output* action, *increasing* priorities, *last* entry matching (as this is the worst-case), 306-byte packets (typical for industrial scenarios) and 17 flow entries (as our experiment consists of four flows). For this case, the switch service curve is given by $R = 11.8$ Mbps and $T = 257\ \mu s + l/R = 464\ \mu s$. Tab. 5.2 shows the corresponding maximum rates, bursts and per-switch delays for the two different resource allocation schemes. As our main goal is to show that *Loko* works and that guarantees are indeed fulfilled, we focus on a simple configuration for our experiments. While the specific values of the bandwidth and delay in other configurations are different (in accordance with Fig. 5.14), the qualitative behavior of the system remains the same.

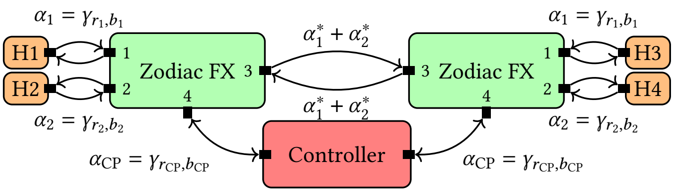

$\alpha_1 = \gamma_{r_1,b_1}$
H1
H2
1
2
Zodiac FX
3
4
$\alpha_2 = \gamma_{r_2,b_2}$
$\alpha_1^* + \alpha_2^*$
$\alpha_1^* + \alpha_2^*$
3
Zodiac FX
1
2
4
$\alpha_1 = \gamma_{r_1,b_1}$
H3
H4
$\alpha_2 = \gamma_{r_2,b_2}$
$\alpha_{\mathrm{CP}} = \gamma_{r_{\mathrm{CP}},b_{\mathrm{CP}}}$
Controller
$\alpha_{\mathrm{CP}} = \gamma_{r_{\mathrm{CP}},b_{\mathrm{CP}}}$

Figure 5.16: *Loko* evaluation setup. Only the arrival curves entering the switches are annotated.

5.5.1 Measurements: Proof-of-Concept Testbed Implementation

Network setup. We interconnect two Zodiac FX switches and connect each of them to two hosts (Fig. 5.16). This corresponds to the scenario investigated in Sec. 5.1.3. For simplicity, we consider a symmetrical scenario where both switches receive flows with arrival curves α_1 and α_2 on their ports 1 and 2 respectively. This traffic is then forwarded to port 3 of the switches, and then further forwarded by the other switch to the corresponding symmetrical hosts. The controller proactively adds these flow entries and places them at the end of the table (with increasing priorities). To account for runtime programmability, we further consider a given traffic α_{CP} from the controller which does not generate DP traffic but potentially generates a CP response (e.g., *EchoRequest*). As a result, the total traffic entering both switches is given by $\alpha_1 + \alpha_2 + \alpha_1^* + \alpha_2^* + \alpha_{\mathrm{CP}}$ where $\alpha_i^* = \gamma_{r_i,b_i+r_iD}$ $\forall i \in \{1,2\}$, where D is the worst-case delay of the switch as computed by the resource allocation algorithm (leftmost heatmaps in Fig. 5.14). We then define r_1, b_1, r_2 and b_2 such that the total amount of bursts and rates entering the two switches are accepted by *Loko* (four rightmost heatmaps in Fig. 5.14). Several rate and burst distributions are possible. For simplicity, and to be able to conduct a parameter-based study, we define N via $b_1 = Nb_2$ and $r_1 = Nr_2$. This leads to

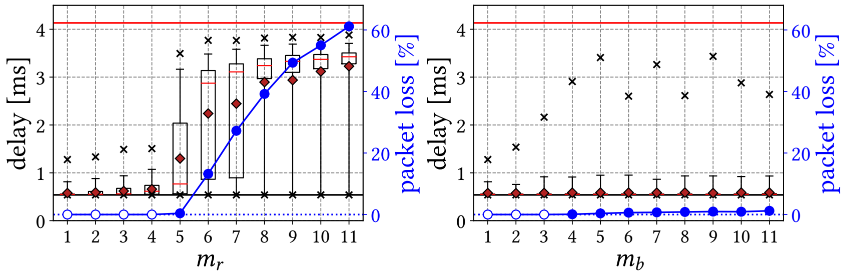
$$r_2 = \frac{R - r_{\mathrm{CP}}}{2N+2}, \qquad b_2 = \frac{B - b_{\mathrm{CP}} - r_2 D(N+1)}{2N+2}. \tag{5.8}$$

We consider that the controller sends c_{pps} *EchoRequest* packets per second. That is, $r_{\mathrm{CP}} = b_{\mathrm{CP}} \times c_{\mathrm{pps}}$ and $b_{\mathrm{CP}} = 66$ bytes if $c_{\mathrm{pps}} \neq 0$, $b_{\mathrm{CP}} = 0$ otherwise.

Traffic generation. In terms of delay and packet loss, the worst case occurs when all the allowed bursts arrive at the same time at a switch. To maximize the probability of this to happen, we use *mgen* [US] to generate randomly separated bursts at line rate and the Linux *tc* utility and its *tbf* queuing discipline [Lin] to shape these bursts so that they follow the computed token-bucket parameters (Eqn. 5.8). We further define the rate multiplier m_r and the burst multiplier m_b to adjust the sending behavior of the hosts. Values greater than 1 imply that the hosts send more than allowed by *Loko*.

Delay measurement. Through a setup similar to Fig. 5.8a, we measure the E2E delay of each packet for the two α_1 flows between $H1$ and $H3$. We then compare the observed delays to the guaranteed latency $2D$ (as each flow traverses two switches): 3.72 ms for the *full-rate* allocation strategy and 4.13 ms for the *fifth-rate* strategy. The traces allow to detect packet loss.

Plots. We plot the packet delays for different parameter combinations as boxplots. The whiskers correspond to the 1% and 99% percentiles. The minimum and maximum outliers are shown as crosses.



(a) $m_b = 1$. (b) $m_r = 1$.

Figure 5.17: *fifth-rate* allocation with $c_{\text{pps}} = 0$ and $N = 2$.

Each boxplot corresponds to the delays observed for 30 runs of 10 seconds, i.e., for a total of 5 minutes, and between 150k and 2M packets observed depending on the case, which we believe is sufficient for statistical significance. A red horizontal line identifies the delay guarantee and a black horizontal line the minimum possible delay based on processing time. The packet loss rate for the 30 runs is further shown in blue. Empty bullets identify cases for which no packets were lost, and full blue bullets identify packet loss.

5.5.1.1 Infeasibility of Some Scenarios

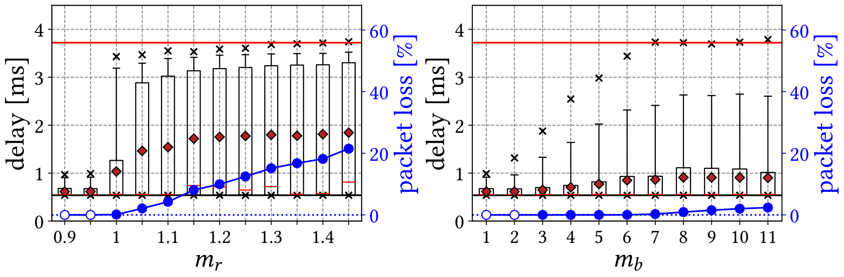
Because of our predefined setup, some cases can be infeasible, i.e., lead to per-flow burst values which are lower than the considered packet size. Indeed, since we have four flows sharing the burst allocated by the resource allocation algorithm, more infeasible cases than in Fig. 5.14 can happen. This is just a property of our simple evaluation setup and is unrelated to *Loko* and its models. The infeasible cases arise because the Zodiac FX buffer is a scarce resource. We always consider 306-byte packets and 17 flow entries because this scenario is always feasible.

5.5.1.2 *fifth-rate* Resource Allocation

We first consider the *fifth-rate* resource allocation scheme (Fig. 5.14b), do not send CP traffic ($c_{\text{pps}} = 0$), and use $N = 2$.

Impact of sent rates. Sending only the allowed bursts ($m_b = 1$), Fig. 5.17a shows the packet delays and packet loss rates observed for different rate multiplier values (m_r). We see that when the *Loko* admission control is respected ($m_r = 1$), no packets are lost, and the delay guarantee is not violated. Increasing m_r, we observe losses starting from $m_r = 5$. Then, the loss rate increases, e.g., to around 60% for $m_r = 11$. We do not observe any delay violation.

Impact of sent bursts. With $m_r = 1$, i.e., sending only at the allowed rate, Fig. 5.17b shows the packet delays and packet loss rates observed for different values of m_b. Again, when the *Loko* admission control is respected ($m_b = 1$), we observe no packet loss and no delay violations. Starting from $m_b = 4$, we observe packet loss, even though less than for $m_r > 1$ (Fig. 5.17a). This is because reaching the throughput limit is easier than reaching the buffer capacity limit. Indeed, since a burst is an instantaneous event, the buffer capacity of the switch is challenged only when the bursts are



(a) $m_b = 1$.

(b) $m_r = 0.95$.

Figure 5.18: *full-rate* allocation with $c_{\text{pps}} = 0$ and $N = 0$.

synchronized, which is probabilistically rare. We also observe that m_r must be increased more in order to observe losses. This is because we are using the *fifth-rate* resource allocation scheme. While losses could have happened for $1 < m_r < 5$, the limit was the buffer capacity, which was not reached because bursts were never synchronized. Reaching $m_r = 5$, since a fifth of the real throughput was allocated, the throughput of the switch also becomes the limit, which leads to more lost packets (and even larger delays).

5.5.1.3 *full-rate* Resource Allocation

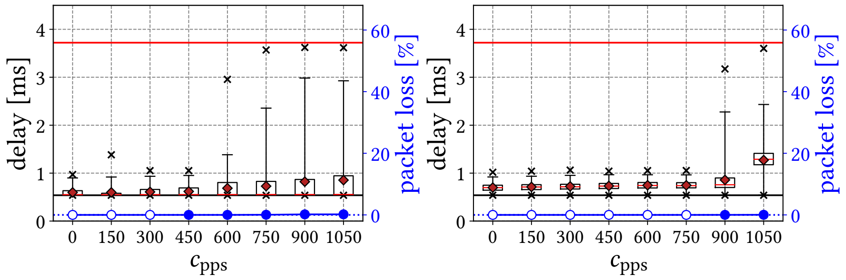
We now consider the *full-rate* resource allocation scheme (Fig. 5.14a) without CP traffic ($c_{\text{pps}} = 0$). In this case, the allowed burst is smaller and hence the infeasibility problem mentioned in Sec. 5.5.1.1 is exacerbated. As a result, we now consider $N = 0$, thereby effectively having only two flows.

Impact of sent rates. With $m_b = 1$, Fig. 5.18a shows the results for different values of m_r. Because of our 6% error in throughput computation (Sec. 5.2.4.2), we observe losses for $m_r = 1$. Using $m_r = 0.95$ allows to account for this error: we then observe no packet loss. Compared to Fig. 5.17a, we see that packet losses happen earlier, i.e., when increasing the sent rate by 5% only. This is because we allocated the full throughput of the switch: again, in practice, the throughput limit can be reached faster than the buffer limit. In contrast, with the *fifth-rate* allocation, while an increase by 5% can theoretically fill the buffer, we do not observe loss because such cases are rare in practice. We also observe delay violations for $m_r = 1.45$.

Impact of sent bursts. With $m_r = 0.95$, Fig. 5.18b shows that delays increase with m_b and packet losses happen starting from $m_b = 3$: again, practically reaching the buffer capacity is probabilistically rare and hence happens for bigger values. We also observe delay violations starting from $m_b = 7$.

5.5.1.4 CP Interference

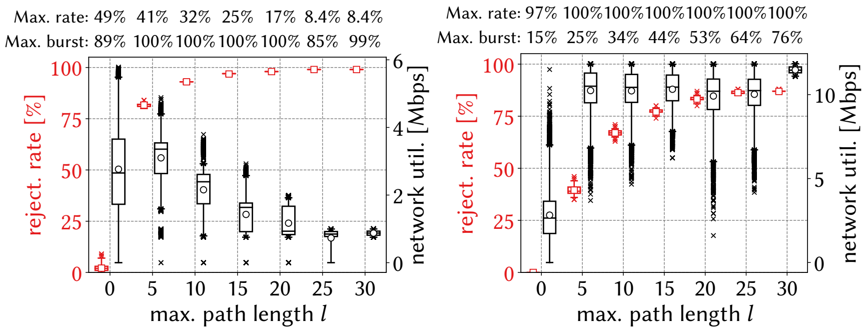
With the *full-rate* strategy and for $m_r = 0.95$ and $m_b = 1$, we introduce CP traffic. Fig. 5.19a shows the result for different values of c_{pps} without including the CP traffic in the model. We observe losses starting from $c_{\text{pps}} = 450$. Fig. 5.19b shows the same scenario but with $c_{\text{pps}} = 750$ included in the modeling. We observe that this prevents losses to happen until $c_{\text{pps}} = 750$, thereby successfully modeling the presence of interfering CP traffic.



(a) c_{pps} not modeled.

(b) $c_{\text{pps}} = 750$ included.

Figure 5.19: *full-rate* allocation with $N = 0$, $m_r = 0.95$ and $m_b = 1$.



(a) Medium-sized flows.

(b) Artificially inc. buffer size.

Figure 5.20: *Loko* scales to path lengths typical of industrial scenarios (∼5 hops). Artificially increasing (10×) the buffer capacity of the Zodiac FX (5.20b) allows to reach the maximum theoretical network utilization (11.8 Mbps, see Tab. 5.2).

5.5.2 Simulations: Scalability and Utilization

In order to assess the scalability, network utilization and rejection rates achieved by *Loko*, we run a simulation of its admission control. We consider a ring network, a typical industrial network topology. The scalability of *Loko* depends only on the burst increase of flows at each hop. Hence, *Loko* does not scale with the network size, and we consider a constant ring size of 31 switches. For a given path length l, ranging from 0 (source and destination are attached to the same switch) to 30 hops, we generate 100 flow requests from a random source node to the node l hops away. The flows have 750 kbps to 1 Mbps bandwidth requirements, corresponding to typical demands observed in traces from a wind park network from a worldwide industrial operator [Kat+17a]. The burst of a flow always corresponds to one packet size, i.e., 306 bytes. We use the *full-rate* resource allocation scheme. The delay guarantees of flows are given by $D \times (l + 1)$, where $D = 1.86$ ms is the per-switch latency guarantee (Tab. 5.2). For paths of 30 hops, this corresponds to around 56 ms, which is on the order of typical latency requirements [Kat+17a]. We run 1000 simulations for each path length l.

We then evaluate the fraction of accepted flows and report the total rate utilization of each switch (which actually corresponds to the network utilization). For each path length, boxplots and outliers show the achieved rejection rates and the utilization for each of the 31 switches over 1000 runs. The whiskers of the boxplots identify the 1% and 99% percentiles. We also show the burst and rate utilization (with respect to the maximum values defined by the resource allocation algorithm – see Tab. 5.2) of the bottleneck switch.

Fig. 5.20a shows that the maximum switch capacity can never be reached (only up to 49%, i.e., 6 Mbps). Because of the small buffer capacity of the Zodiac FX, the maximum burst allowance is always the bottleneck and the reason for rejecting flows. We observe that *Loko* can scale up to around 5 nodes, a typical maximum path length in a medium-sized industrial network [Kat+17a]. The rejection rates increase with the path lengths because, as per network calculus, a flow consumes more buffer resources at each hop it passes (the green curve in Fig. 2.20 has a higher burst than the red curve). In order to further evaluate the impact of the buffer capacity, we now hypothetically assume a switch with a buffer capacity 10 times larger. Now, flows are rejected because of reaching the throughput capacity of the switch, i.e., 11.8 Mbps in this scenario (Fig. 5.20b). As a result, though the rejection rates still increase with the path lengths, the network utilization stays mostly around its maximum value. This artificial scenario shows that while *Loko* can, in principle, realize the maximum throughput of the Zodiac FX switch, the small size of the switch buffer causes rejection of flows to avoid buffer overflow, preventing *Loko* from reaching the full throughput.

5.5.3 Outcomes

We observe that, if the admission control of *Loko* is respected, no packets are lost and delay violations do not occur. We also see that packet loss and delay violations can indeed happen if hosts send more than allowed. We further show that interfering CP traffic can also lead to DP packet loss and that *Loko* is able to incorporate this in its modeling in order to provide its guarantees even in the presence of CP traffic. Finally, we show that the bounds, network utilization and scalability achieved by *Loko* satisfy the requirements of existing industrial applications. *As such,* Loko *successfully provides deterministic latency guarantees for low-cost programmable switches serving industrial applications, even in the presence of interfering CP traffic.*

5.6 Discussion: Generalizability

While we demonstrated *Loko* for the particular case of the Zodiac FX, we discuss whether it generalizes to other switches. In principle, *Loko* can apply to any switch that processes packets using a centralized CPU. There is a single requirement: the processing of the CPU must be deterministic, which is for instance *not the case for OS-based processing*. In such cases (e.g., the Banana Pi R1 and R2 and the new Zodiac GX), the OS and other processes can interfere with packet processing. However, alternatives such as core pinning exist and might provide performance determinism. For instance, packet processing frameworks like DPDK, which are assigned a complete CPU core, could be used: there, execution is isolated on a separate core and not disturbed by the kernel running on the other cores. Further work is needed to assess the performance predictability of DPDK or of a lightweight network driver bypassing the OS kernel. This opens a broad range of applications as DPDK-based implementations for network functions are more and more common, because they provide greater performance.

Furthermore, while we focused our implementation on an SDN/OF switch, our approach is not tied to these technologies. The only requirement is to have a programmable forwarding behavior. For example, a newly released firmware of the Zodiac FX supports P4. This could also be modeled and used by *Loko*. On the other hand, we highlight that *Loko* is designed for and tailored to low-cost low-capacity switches and, hence, could not be used for commodity networking hardware; such devices do not exhibit inter-port interferences due to centralized CPU processing.

Typically, latency-critical applications require safety, reliability, and ability to operate in harsh environments (e.g., high temperatures, dust, or humidity). We did not consider such aspects. Our work is a first step toward showing that low-cost switches can be used, at least from a networking performance point of view, for providing predictable performance. The analysis and evaluation of other aspects (e.g., the mean time between failures (MTBF) of the switch) are left for future work.

Finally, we highlight that *Loko* only supports applications with clear and constant network resource requirements in terms of token-bucket burst and rate parameters. The incorporation of rather unpredictable traffic (e.g., TCP or video streaming) or of traffic which does not require any latency guarantees, requires the design of isolation mechanisms that would prevent such applications to interfere with (or use resources of) applications with strict requirements, which also constitutes an interesting topic for future work.

5.7 Summary

A predictable network performance is mission critical for many applications and yet hard to provide due to difficulties in modeling the behavior of the increasingly complex network equipment. This chapter studied the problem of providing deterministic latency guarantees in *small networks* based on *low-capacity hardware* (e.g., in-cabin and industrial networks): such networks are of increasing importance, need to meet stringent performance requirements, but have hardly been explored so far. The main contribution of this chapter is the design, implementation, and evaluation of *Loko*, a system which provides predictable latency guarantees in *programmable* networks using *low-cost* hardware. *Loko* relies on a novel measurement-based methodology and uses DNC to derive a reliable performance model of a given switch and implement the model functions of the DetServ architecture (see Sec. 2.3.4). To this end, we also showed that SoA models in the literature like QJump and Silo fall short to model the behavior of such switches, due to incorrect architectural and performance assumptions. As a case study, we implemented *Loko* for the Zodiac FX switch. Our experiments are encouraging: we found that the derived models are indeed accurate, allowing *Loko* to provide deterministic E2E guarantees with low-cost programmable devices.

Beside illustrating the correctness of *Loko*'s operation, our results convey two main messages. First, low-cost devices should not be underestimated, as minimal but tailored implementations are sufficient to provide predictable performance: guaranteed performance and simple programmability are not mutually exclusive. Second, low-cost devices require to take precautions: traditional assumptions can become wrong and invalidate existing theories. In general, we view our work as a first step and believe that it opens several interesting avenues for future research around tailored implementations on low-cost devices such as DPDK-based packet processing on multi-port NICs.

5.8 Appendix: Silo Guarantees for our Scenario

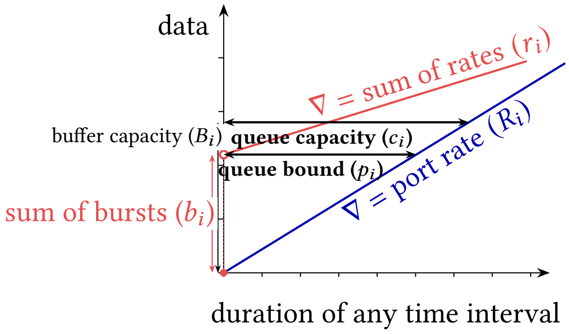


Figure 5.21: Silo's concepts of *queue bound* and *queue capacity* [Jan+15] for port i.

The guarantees provided by Silo [Jan+15] are based on an admission control scheme. It relies on the concepts of *queue bound* and *queue capacity*, defined for each port i.

- The *queue bound* p_i is the maximum queuing delay that can occur at a port i. If the total rate r_i sent to port i is greater than its output rate R_i, it is infinite. Otherwise, it is computed by dividing the total burst b_i sent to the port by the port rate R_i, i.e.,

$$p_i = \begin{cases} \infty & \text{if } r_i > R_i, \\ b_i/R_i & \text{otherwise.} \end{cases} \tag{5.9}$$

  The queue bounds are *dependent* on the traffic in the network.

- The *queue capacity* c_i is the maximum queuing delay that can occur at a port i before packets are dropped. It is computed by dividing the port buffer capacity B_i by the port rate R_i, i.e.,

$$c_i = \frac{B_i}{R_i}. \tag{5.10}$$

  The queue capacity is *independent* of the traffic in the network.

These concepts are illustrated in Fig. 5.21 for a given port i.

A new flow is accepted on a given path if the queue bounds on the output ports of this path are all lower than the corresponding queue capacities [Jan+15], i.e., if

$$p_i \leq c_i \qquad \forall i \in \text{path}. \tag{5.11}$$

Then, the latency guarantee L of the flow corresponds to the sum of queue capacities over the path of the flow [Jan+15], i.e.,

$$L = \sum_{i \in \text{path}} c_i. \tag{5.12}$$

The sum r_i of the rates at a port simply corresponds to the sum of the rates of all the flows going through this port.

The sum b_i of the bursts at a port corresponds to the sum of the bursts generated by the individual flows flowing through this port. At its first hop, the burst generated by a flow corresponds to its original burst. At each subsequent hop, this burst is increased by the rate of the flow multiplied by the queue capacity c_i of the previously traversed port [Jan+15].

Zodiac FX. With 306-byte packets, as measured in Sec. 5.2.5, the Zodiac FX has a total buffer size of 9 packets, i.e., 3 per data port. This leads to the following queue capacity at each port

$$c_i = \frac{3 \times 306 \text{ bytes}}{100 \text{ Mbps}} = 73.4\ \mu\text{s}, \quad \forall i. \tag{5.13}$$

Over our two-hop network, accepted flows receive the following guarantee on latency

$$L = 2 \times c_i = 146.9\ \mu\text{s}. \tag{5.14}$$

Silo would allow each host to send traffic at the rate of 45 Mbps and with a maximum burst of 306 bytes. Indeed, the generated queue bounds p_i are all lower than the queue capacities c_i. The ports between the switches transport two flows with their original burst. That is, the queue bound for these ports is given by

$$p_i = \frac{2 \times 306 \text{ bytes}}{100 \text{ Mbps}} = 49.0\ \mu\text{s}. \tag{5.15}$$

The ports connected towards the hosts only transport one flow, but with the burst increased by an already traversed output port. Hence, for output ports connected to hosts, the queue bound is given by

$$p_i = \frac{306 \text{ bytes} + 45 \text{ Mbps} \times 73.4\ \mu\text{s}}{100 \text{ Mbps}} = 57.5\ \mu\text{s}. \tag{5.16}$$

Both these queues bounds are lower than the queue capacities c_i, i.e., we indeed have $p_i \leq c_i$ for all ports i.

Banana Pi R1. For the Banana Pi R1, computations must be adapted to account for the 1 Gbps link rate supported by the switch. We consider the same buffer size as for the Zodiac FX: the queue capacity is given by

$$c_i = \frac{3 \times 306 \text{ bytes}}{1 \text{ Gbps}} = 7.34\ \mu\text{s} \tag{5.17}$$

and the guaranteed latency by

$$L = 2 \times c_i = 14.7\ \mu\text{s}. \tag{5.18}$$

Silo would allow each host to send traffic at the rate of 450 Mbps and with a maximum burst of 306 bytes: the queue bounds they generate is

$$p_i = \frac{2 \times 306 \text{ bytes}}{1 \text{ Gbps}} = 4.90\ \mu\text{s} \tag{5.19}$$

for the ports connecting switches (the first burst of two flows) and

$$p_i = \frac{306 \text{ bytes} + 45 \text{ Mbps} \times 7.34\ \mu\text{s}}{1 \text{ Gbps}} = 5.75\ \mu\text{s} \tag{5.20}$$

for the output ports connected to hosts (the burst of one flow increased by one hop), both of which are lower than the queue capacities c_i, i.e., we indeed have $p_i \leq c_i$ for all ports i.

Chapter 6

Conclusions and Outlook

Applications from emerging systems such as the IoT, CPSs, and cloud computing impose new requirements on the communication networks on top of which they are deployed. In particular, these applications require predictability from the networking infrastructure, both in terms of correctness and performance. Predictable network performance indeed enables applications to guarantee a given quality of service to their users and customers. For example, a safety-critical control loop in a manufacturing plant would require predictable latency, and in particular strict E2E per-packet latency guarantees, in order to ensure that safety operations and commands are triggered on time upon reception of emergency signals or events. Similarly, control flows responsible for ensuring synchronization and state replication among control VMs (e.g., SDN controllers) in a data center can provide guarantees on the worst-case synchronization time and state divergence if the underlying network infrastructure provides E2E latency guarantees.

As illustrated by the two above examples, among the different network QoS properties, latency is one of the most critical metric for applications, both in industrial and data center networks. This thesis focused on the design, implementation, and evaluation of mechanisms for providing latency guarantees to applications. In particular, we focused on strict latency guarantees, where each packet is guaranteed to reach its destination within a given latency bound. SoA systems for providing such guarantees are typically expensive, inflexible, and lead to vendor lock-in because they rely on proprietary protocols or require changes within the network protocol stack of end hosts and/or forwarding devices. Programmable networks have been considered in the recent years as a solution to overcome such protocol openness issues and improve the automation and flexibility of network configuration and management. Network operators can use open interfaces and programs written independently from the networking hardware to configure the behavior of their network. Albeit its numerous benefits towards flexibility, network programmability also introduces challenges, in particular with respect to predictability and network performance. This thesis studied the particular problem of providing predictable latency, i.e., strict E2E latency bounds, to applications in programmable networks. The next section summarizes the key contributions and outcomes of this thesis and Sec. 6.2 reports on interesting and challenging future research directions in the area of predictable latency in programmable networks.

6.1 Summary

The main outcome of this thesis consists of the design, implementation, and evaluation of two complete systems for providing predictable latency in programmable networks.

First, *Chameleon* focuses on data center networks and circumvents typical unpredictable behavior of programmable devices by pushing all its configuration to end-hosts and relying on source routing to enforce the forwarding decisions in the network. Priority levels and routes followed by flows in the network are continuously updated in order to improve resources utilization. The usage of source routing allows to easily ensure that per-packet guarantees are satisfied at any time, even in the presence of reconfigurations. We showed that *Chameleon* outperforms the SoA in terms of network utilization: some links can reach 100% utilization and up to 7 times more flows can be accepted compared to SoA approaches.

Second, *Loko* focuses on networks with low-cost and low-capacity forwarding devices, which we refer to as *small networks*. We showed with detailed measurements that traditional latency models, including those used by *Chameleon*, are not valid for low-capacity network equipment. This is mostly due to the fact that low-capacity hardware typically relies on a central CPU for packet processing, which leads to inter-port interferences, a phenomenon typically assumed non-existent by SoA latency models. The results of our detailed measurement campaign allowed to derive switch and network models for the design of access control and resource allocation routines. We showed that the resulting *Loko* system successfully provides strict latency guarantees with software-based, low-cost and low-capacity forwarding equipment. This result conveys an important message. Programmable networks which do not depend on complex switch hardware platforms can provide predictable performance and may thus find interesting applications in other contexts as well. For example, using DPDK for deploying new applications with predictable performance constitutes an attractive low-cost and more flexible alternative to the typically more expensive and less flexible hardware programmable devices.

Towards these systems, this thesis made contributions in several areas. We succinctly summarize these in the following paragraphs.

Measurements of programmable hardware and components. While programmable devices have been designed with flexibility and ease-of-configuration in mind, performance predictability remained an open question. Through detailed measurement campaigns, we investigated the predictability of programmable forwarding devices, both from a performance and a management point of view. We quantified the performance of switch hardware from many different angles, from unpredictable behaviors (e.g., flow configuration and buffer management) to predictable behaviors (e.g., processing time and throughput) and including unexpected but predictable behaviors (e.g., overhead of scheduling disciplines). These measurements and their output formed the basis for the design of (re-)configuration procedures. In the particular case of *Chameleon*, the observed unpredictable management behavior of switches were circumvented through end-host networking and source routing.

Design of strict latency forwarding models. Based on these measurements, we designed latency models that form the basis of our complete E2E provisioning systems. By allocating a time resource to individual priority queues (or switches in the case of *Loko*), our models ensure that per-priority-queue (or per-switch) latency bounds are guaranteed throughout the lifetime of the network. An access control routine is responsible for keeping this invariant. We have shown that our models allow to reach high network utilization (some links can reach 100% network utilization) while still providing low request processing time (at most hundreds of milliseconds).

Optimization of the routing procedure. As part of network QoS provisioning systems, the routing procedure plays a central role. It is responsible for finding valid embeddings in a short amount of time. This thesis thoroughly investigated the algorithmic issue of finding a route for applications. In particular, we proposed enhancements to SoA algorithms for improving their runtime, optimality, and completeness in various different network settings.

6.2 Future Work

We believe that the following research directions are of particular interest for future work.

Stochastic latency guarantees. This thesis focused on the provisioning of strict latency guarantees that are satisfied at any time on a per-packet basis. While having such guarantees presents various benefits for many applications, the quest for deterministic and strict latency guarantees leads to forced pessimistic assumptions (as part of DNC), which in turn leads to network resources that are effectively left unused while they could be used by other applications. In other words, and as shown in Tab. 2.1, ensuring strict latency guarantees prevents from achieving work-conservation. Furthermore, network traffic typically exhibits variations that can be hard to deterministically predict and ensuring that users strictly comply with agreed traffic envelopes is challenging. An interesting and promising research direction is to slightly relax the deterministic guarantees and look for guarantees that are only stochastically guaranteed. For example, an application could be guaranteed a maximum E2E latency of 120 ms for 95% of its packets. While many applications in cloud computing and CPSs would still greatly benefit from such guarantees, relaxing the deterministic aspect of the requirements has the potential of drastically improving network utilization, and hence revenue for network operators. Such a relaxation would also make it possible to deal with uncertainties in network conditions (e.g., failures) and less predictable traffic patterns that only follow stochastic distributions. To understand how the latency models, routing procedures, and results in this thesis would have to be adapted is an interesting and challenging research question.

Investigation of networks providing both deterministic and stochastic latency guarantees. Instead of focusing either on deterministic or stochastic guarantees, an interesting research direction is to design a system able to provide both QoS levels to its applications. This is a particularly challenging problem, as deterministic guarantees require the precise allocation and management of resources, while stochastic guarantees inherently introduce uncertainty and variations in the resources consumption of applications. As a result, appropriate isolation techniques have to be investigated and developed to logically separate both types of traffic and prevent any interference that would violate SLAs.

Leverage programmable data planes for latency guarantees in wide-area networks. While this thesis investigated many different types of networks, including small networks, data center networks as well as industrial networks, wide-area networks, e.g., Internet service provider (ISP) or autonomous system (AS) networks, present very interesting challenges for achieving latency guarantees. At the same time, many applications deployed over the Internet, e.g., remote surgery, would greatly benefit from latency guarantees. Whereas the networks considered in this thesis can be managed through an out-of-band mechanism, wide-area networks require in-band management. In the context of predictability, this introduces various challenges. For example, control and data flows must be properly isolated while still guaranteeing the appropriate QoS for both channels. The investigation of this kind of network and how the solutions proposed in this thesis can be applied and adapted is an interesting research direction. In particular, the (re-)configuration of forwarding devices is a challenging problem, as the solution based on end-host networking proposed in this thesis cannot be deployed on networks of such scale and physical size. Such networks rather require the solution to be implemented in the network and it is an interesting research direction to investigate how data plane programmability technologies such as P4 [Bos+14] can help to implement an in-network solution for providing strict latency guarantees, e.g., by performing tagging operations and/or taking local decisions based on observed delay variations.

Bibliography

Publications by the Author

Journal Publications

[Guc+17] J. W. Guck, A. Van Bemten, M. Reisslein, and W. Kellerer. "Unicast QoS Routing Algorithms for SDN: A Comprehensive Survey and Performance Evaluation." In: *IEEE Communications Surveys & Tutorials* 20.1 (2017), pp. 388–415.

[GVK17] J. W. Guck, A. Van Bemten, and W. Kellerer. "DetServ: Network Models for Real-Time QoS Provisioning in SDN-based Industrial Environments." In: *IEEE Transactions on Network and Service Management (TNSM)* 14.4 (2017), pp. 1003–1017.

[Viz+19] P. Vizarreta, A. Van Bemten, E. Sakic, K. Abbasi, N. E. Petroulakis, W. Kellerer, and C. Mas Machuca. "Incentives for a Softwarization of Wind Park Communication Networks." In: *IEEE Communications Magazine* 57.5 (2019), pp. 138–144.

[Zop+18] S. Zoppi, A. Van Bemten, H. M. Gürsu, M. Vilgelm, J. Guck, and W. Kellerer. "Achieving Hybrid Wired/Wireless Industrial Networks with WDetServ: Reliability-based Scheduling for Delay Guarantees." In: *IEEE Transactions on Industrial Informatics* 14.5 (2018), pp. 2307–2319.

Conference Publications

[Kat+17a] S. Katsikeas, K. Fysarakis, A. Miaoudakis, A. Van Bemten, I. Askoxylakis, I. Papaefstathiou, and A. Plemenos. "Lightweight & Secure Industrial IoT Communications via the MQ Telemetry Transport Protocol." In: *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*. IEEE. 2017, pp. 1193–1200.

[Sak+20] E. Sakic, A. Van Bemten, M. Avdic, and W. Kellerer. "Automated Bootstrapping of A Fault-Resilient In-Band Control Plane." In: *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*. ACM. 2020, pp. 1–13.

[Van+18a] A. Van Bemten, J. W. Guck, C. Mas Machuca, and W. Kellerer. "Routing Metrics Depending on Previous Edges: The M*n* Taxonomy and its Corresponding Solutions." In: *Proceedings of the IEEE International Conference on Communications (ICC)*. IEEE. 2018, pp. 1–7.

[Van+18b] A. Van Bemten, J. W. Guck, P. Vizarreta, C. Mas Machuca, and W. Kellerer. “LARAC-SN and Mole in the Hole: Enabling Routing through Service Function Chains.” In: *Proceedings of the IEEE Conference on Network Softwarization and Workshops (NetSoft).* IEEE. 2018, pp. 298–302.

[Van+19a] A. Van Bemten, N. Đerić, A. Varasteh, A. Blenk, S. Schmid, and W. Kellerer. “Empirical Predictability Study of SDN Switches.” In: *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS).* ACM/IEEE. 2019, pp. 1–13.

[Van+19b] A. Van Bemten, N. Đerić, J. Zerwas, A. Blenk, S. Schmid, and W. Kellerer. “Loko: Predictable Latency in Small Networks.” In: *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT).* ACM. 2019, pp. 355–369.

[Van+20] A. Van Bemten, N. Đerić, A. Varasteh, S. Schmid, C. Mas Machuca, A. Blenk, and W. Kellerer. “Chameleon: Predictable Latency and High Utilization with Queue-Aware and Adaptive Source Routing.” In: *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT).* 2020, pp. 451–465.

Technical Reports

[Van+19c] A. Van Bemten, J. W. Guck, C. Mas Machuca, and W. Kellerer. “Bounded Dijkstra (BD): Search Space Reduction for Expediting Shortest Path Subroutines.” In: *arXiv preprint arXiv:1903.00436* (2019).

[VK16] A. Van Bemten and W. Kellerer. “Network Calculus: A Comprehensive Guide.” In: *Technical University of Munich, Chair of Communication Networks, Technical Report No. 201603* (Oct. 2016).

Web

[Van19a] A. Van Bemten. *Source code, configuration files, and data sets associated to* Loko. 2019. URL: https://loko.lkn.ei.tum.de.

[Van19b] A. Van Bemten. *Source code, configuration files, and data sets associated to the empirical predictability study of SDN switches.* 2019. URL: https://sdn-predictability.lkn.ei.tum.de.

[Van19c] A. Van Bemten. *LORA: The League of Routing Algorithms.* 2017–2019. URL: https://lora.lkn.ei.tum.de.

General Publications

[05] “Communication Delivery Time Performance Requirements for Electric Power Substation Automation.” In: *IEEE Std 1646-2004* (2005), pp. 1–24.

[Abr+12] I. Abraham, D. Delling, A. Goldberg, and R. Werneck. “Hierarchical hub labelings for shortest paths.” In: *Springer European Symposium on Algorithms* (2012), pp. 24–35.

[Ada+15] D. Adami, L. Donatini, S. Giordano, and M. Pagano. “A network control application enabling Software-Defined Quality of Service.” In: *Proceedings of the IEEE International Conference on Communications (ICC)*. 2015, pp. 6074–6079.

[Add+15] B. Addis, D. Belabed, M. Bouet, and S. Secci. “Virtual network functions placement and routing optimization.” In: *Proceedings of the IEEE International Conference on Cloud Networking (CloudNet)*. Oct. 2015, pp. 171–177.

[AE10] R. A. Adams and C. Essex. *Calculus. A Complete Course*. 7th ed. Pearson Education Canada, 2010.

[ÅGB11] J. Åkerberg, M. Gidlund, and M. Björkman. “Future research challenges in wireless sensor and actuator networks targeting industrial automation.” In: *Proceedings of the International Conference on Industrial Informatics*. IEEE. 2011, pp. 410–415.

[Agy+14] P. K. Agyapong, M. Iwamura, D. Staehle, W. Kiess, and A. Benjebbour. “Design considerations for a 5G network architecture.” In: *IEEE Communications Magazine* 52.11 (2014), pp. 65–75.

[AIN] A. Adam, A. Ilan, and T. Nadeau. *Introduction to virtio-networking and vhost-net (Red Hat Blog)*. https://www.redhat.com/en/blog/introduction-virtio-networking-and-vhost-net. Accessed: 2020-02-02.

[AIY13] T. Akiba, Y. Iwata, and Y. Yoshida. “Fast exact shortest-path distance queries on large networks by pruned landmark labeling.” In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM. 2013, pp. 349–360.

[Ali+11] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. “Data center TCP (DCTCP).” In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. ACM. 2011, pp. 63–74.

[Ali+12] M. Alizadeh, A. Kabbani, T. Edsall, B. Prabhakar, A. Vahdat, and M. Yasuda. “Less is more: trading a little bandwidth for ultra-low latency in the data center.” In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2012, pp. 253–266.

[Ali+13] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. “pFabric: Minimal near-optimal datacenter transport.” In: *ACM SIGCOMM Computer Communication Review*. Vol. 43. 4. ACM. 2013, pp. 435–446.

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[An+16] N. An, T. Ha, K.-J. Park, and H. Lim. “Dynamic priority-adjustment for real-time flows in software-defined networks.” In: *Proceedings of the IEEE International Telecommunications Network Strategy and Planning Symposium*. IEEE. 2016, pp. 144–149.

[AN78] Y. P. Aneja and K. P. Nair. “The constrained shortest path problem.” In: *Wiley Naval Research Logistics (NRL)* 25.3 (1978), pp. 549–555.

[And16] R. C. de Andrade. "New formulations for the elementary shortest-path problem visiting a given set of nodes." In: *Elsevier European Journal of Operational Research* 254.3 (2016), pp. 755–768.

[Ara+18] J. T. Araújo, L. Saino, L. Buytenhek, and R. Landa. "Balancing on the edge: Transport affinity without network state." In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2018, pp. 111–124.

[AX14] A. V. Akella and K. Xiong. "Quality of service (QoS)-guaranteed network resource allocation via software defined networking (SDN)." In: *Proceedings of the IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC)*. IEEE. 2014, pp. 7–13.

[Bac+92] F. Baccelli, G. Cohen, G. J. Olsder, and J.-P. Quadrat. *Synchronization and Linearity, An Algebra for Discrete Event Systems*. John Wiley and Sons, 1992.

[Bai+16] W. Bai, L. Chen, K. Chen, and H. Wu. "Enabling {ECN} in Multi-Service Multi-Queue Data Centers." In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2016, pp. 537–549.

[Bal+11] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. "Towards predictable datacenter networks." In: *ACM SIGCOMM Computer Communication Review*. Vol. 41. 4. ACM. 2011, pp. 242–253.

[Bal+13] H. Ballani, K. Jang, T. Karagiannis, C. Kim, D. Gunawardena, and G. O'Shea. "Chatty tenants and the cloud network sharing problem." In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2013, pp. 171–184.

[Bar+13] M. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, et al. "PolicyCop: An Autonomic QoS Policy Enforcement Framework for Software Defined Networks." In: *Proceedings of the IEEE SDN for Future Networks and Services (SDN4FNS) Conference*. IEEE. 2013, pp. 1–7.

[Bar+15] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba. "On orchestrating virtual network functions." In: *Proceedings of the IEEE/IFIP International Conference on Network and Service Management (CNSM)*. IEEE. 2015, pp. 50–56.

[Bas+14] A. Basta, W. Kellerer, M. Hoffmann, H. J. Morper, and K. Hoffmann. "Applying NFV and SDN to LTE mobile core gateways, the functions placement problem." In: *Proceedings of the ACM SIGCOMM Workshop on All Things Cellular*. ACM. 2014, pp. 33–38.

[Bas+16] H. Bast, D. Delling, A. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck. "Route planning in transportation networks." In: *Algorithm Engineering*. Springer, 2016, pp. 19–80.

[Bau+07] R. Baumann, S. Heimlicher, M. Strasser, and A. Weibel. "A survey on routing metrics." In: *ETH Zürich, Computer Engineering and Networks Laboratory, TIK Report* 262 (2007).

[Bau+18] S. Bauer, D. Raumer, P. Emmerich, and G. Carle. "Behind the scenes: what device benchmarks can tell us." In: *Proceedings of the ACM/IRTF Applied Networking Research Workshop (ANRW)*. ACM/IRTF. Montreal, Canada, 2018, pp. 58–65.

[BCS94] R. Braden, D. Clark, and S. Shenker. *Integrated Services in the Internet Architecture: an Overview.* RFC 1633. RFC Editor, June 1994. URL: http://www.rfc-editor.org/rfc/rfc1633.txt.

[BE12] M. J. Bannister and D. Eppstein. "Randomized speedup of the Bellman-Ford algorithm." In: *Proceedings of the SIAM Meeting on Analytic Algorithmics and Combinatorics.* 2012, pp. 41–47.

[Bel58] R. Bellman. "On a routing problem." In: *Quarterly of Applied Mathematics* 16.1 (Apr. 1958), pp. 87–90.

[BG96] D. Blokh and G. Gutin. "An approximate algorithm for combinatorial optimization problems with two parameters." In: *Australasian Journal of Combinatorics* 14 (1996), pp. 157–164.

[Bia+10] A. Bianco, R. Birke, L. Giraudo, and M. Palacin. "OpenFlow switching: Data plane performance." In: *Proceedings of the IEEE International Conference on Communications (ICC).* IEEE. 2010, pp. 1–5.

[BJT09] A. Bouillard, L. Jouhet, and E. Thierry. "Service Curves in Network Calculus: dos and don'ts." In: *Research Report 7094, INRIA* (2009).

[BM12] M. Bourdellès and N. Menegale. "Routing optimization for network coding." In: *Proceedings of the IFIP Wireless Days.* IEEE. 2012, pp. 1–6.

[BM99] S. Bradner and J. McQuaid. *Benchmarking Methodology for Network Interconnect Devices.* RFC 2544. RFC Editor, Mar. 1999. URL: http://www.rfc-editor.org/rfc/rfc2544.txt.

[BO62] A. M. Brucker and E. Ostrow. "Some Function Classes Related to the Class of Convex Functions." In: *Pacific Journal of Mathematics* 12.4 (Apr. 1962), pp. 1203–1215.

[Bos+08] P. Bose, P. Carmi, M. Farshi, A. Maheshwari, and M. Smid. "Computing the greedy spanner in near-quadratic time." In: *Scandinavian Workshop on Algorithm Theory.* Springer. 2008, pp. 390–401.

[Bos+14] P. Bosshart et al. "P4: Programming protocol-independent packet processors." In: *ACM SIGCOMM Computer Communication Review* 44.3 (2014), pp. 87–95.

[BR13] Z. Bozakov and A. Rizk. "Taming SDN controllers in heterogeneous hardware environments." In: *Proceedings of the IEEE European Workshop on Software Defined Networks (EWSDN).* IEEE. 2013, pp. 50–55.

[BS18] P. Biondi and the Scapy community. *Scapy.* https://scapy.net. Accessed: 2018-10-18. 2018.

[BV04] S. Boyd and L. Vandenberghe. *Convex optimization.* Cambridge University Press, 2004.

[CA03] G. Cheng and N. Ansari. "A new heuristics for finding the delay constrained least cost path." In: *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM).* Vol. 7. 2003, pp. 3711–3715.

[CBL05] F. Ciucu, A. Burchard, and J. Liebeherr. "A Network Service Curve Approach For the Stochastic Analysis of Networks." In: *ACM SIGMETRICS Performance Evaluation Review.* Vol. 33. 1. ACM. 2005, pp. 279–290.

[CGR96] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. "Shortest paths algorithms: Theory and experimental evaluation." In: *Springer Mathematical Programming* 73.2 (May 1996), pp. 129–174.

[CGS99] B. V. Cherkassky, A. V. Goldberg, and C. Silverstein. "Buckets, heaps, lists, and monotone priority queues." In: *SIAM Journal on Computing* 28.4 (1999), pp. 1326–1346.

[Cha00] C.-S. Chang. *Performance Guarantees in Communication Networks.* Springer Verlag, 2000.

[Che+16] G. Chen et al. "Fast and Cautious: Leveraging Multi-path Diversity for Transport Loss Recovery in Data Centers." In: *Proceedings of the USENIX Annual Technical Conference.* 2016, pp. 29–42.

[Cho+16] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. "{HUG}: Multi-Resource Fairness for Correlated and Elastic Demands." In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI).* 2016, pp. 407–424.

[Cho95] E. I. Chong. "On finding single source single destination k shortest paths." In: *Proceedings of the International Conference on Computing and Information.* 1995, pp. 40–47.

[Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* 3rd ed. MIT Press, 2009.

[Cor+13] J. C. Corbett et al. "Spanner: Google's globally distributed database." In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), pp. 1–22.

[Cru91] R. L. Cruz. "A Calculus for Network Delay, Part I: Network Elements in Isolation." In: *IEEE Transactions on Information Theory* 37.1 (Jan. 1991), pp. 114–131.

[DBK15] R. Durner, A. Blenk, and W. Kellerer. "Performance study of dynamic QoS management for OpenFlow-enabled SDN switches." In: *Proceedings of the IEEE International Symposium on Quality of Service (IWQoS).* IEEE. 2015, pp. 177–182.

[DDC18] A. Drescher, J. DeHart, and P. Crowley. "Bayesian factor analysis and performance measurement of the Linux forwarding architecture." In: *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS).* ACM/IEEE. 2018, pp. 28–40.

[Deb01] K. Deb. *Multi-objective Optimization Using Evolutionary Algorithms.* Vol. 16. John Wiley & Sons, Chichester, UK, 2001.

[DeC+07] G. DeCandia et al. "Dynamo: amazon's highly available key-value store." In: *ACM SIGOPS Operating Systems Review.* Vol. 41. 6. ACM. 2007, pp. 205–220.

[Dec05] J.-D. Decotignie. "Ethernet-based real-time and industrial communications." In: *Proceedings of the IEEE* 93.6 (2005), pp. 1102–1117.

[Del] Dell. *Dell EMC Networking S4048-ON Switch.* https://i.dell.com/sites/doccontent/shared-content/data-sheets/en/Documents/Dell-EMC-Networking-S4048-ON-Spec-Sheet.pdf. Accessed: 2019-01-31.

[Dij59] E. W. Dijkstra. "A note on two problems in connexion with graphs." In: *Springer Numerische mathematik* 1.1 (1959), pp. 269–271.

[Don02] S. F. Donnelly. “High precision timing in passive measurements of data networks.” PhD thesis. University of Waikato, 2002.

[DPD18] DPDK Project. *Home - DPDK*. https://www.dpdk.org. Accessed: 2018-10-18. 2018.

[Dre69] S. E. Dreyfus. “An appraisal of some shortest-path algorithms.” In: *INFORMS Operations Research* 17.3 (1969), pp. 395–412.

[Dua14] Q. Duan. “Network-as-a-service in Software-Defined Networks for end-to-end QoS provisioning.” In: *Proceedings of the IEEE Wireless and Optical Communication Conference (WOCC)*. IEEE. 2014, pp. 1–5.

[Duf+17] M. Dufour, S. Paris, J. Leguay, and M. Draief. “Online Bandwidth Calendaring: On-the-fly admission, scheduling, and path computation.” In: *Proceedings of the IEEE International Conference on Communications (ICC)*. IEEE. 2017, pp. 1–6.

[Egi+12] 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 IEEE Asia-Pacific Signal & Information Processing Association Annual Summit and Conference (APSIPA ASC)*. IEEE. 2012, pp. 1–8.

[Emm+14] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. “Performance characteristics of virtual switching.” In: *Proceedings of the IEEE International Conference on Cloud Networking (CloudNet)*. IEEE. 2014, pp. 120–125.

[Emm+15] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. “Moongen: A scriptable high-speed packet generator.” In: *Proceedings of the ACM Internet Measurement Conference (IMC)*. ACM. 2015, pp. 275–287.

[End16] Endace Technology Limited. *Endace DAG 7.5G4 Datasheet"*. https://www.endace.com/dag-7.5g4-datasheet.pdf. Accessed: 2018-10-26. 2016.

[Fen+02a] G. Feng, C. Douligeris, K. Makki, and N. Pissinou. “Performance evaluation of delay-constrained least-cost QoS routing algorithms based on linear and nonlinear lagrange relaxation.” In: *Proceedings of the IEEE International Conference on Communications (ICC)*. Vol. 4. 2002, pp. 2273–2278.

[Fen+02b] G. Feng, K. Makki, N. Pissinou, and C. Douligeris. “Heuristic and exact algorithms for QoS routing with multiple constraints.” In: *IEICE Transactions on Communications* 85.12 (2002), pp. 2838–2850.

[Fid06] M. Fidler. “WlC15-2: A Network Calculus Approach to Probabilistic Quality of Service Analysis of Fading Channels.” In: *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*. IEEE. 2006, pp. 1–6.

[Fir+18] D. Firestone et al. “Azure accelerated networking: SmartNICs in the public cloud.” In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2018, pp. 51–66.

[Fir17] D. Firestone. “VFP: A Virtual Switch Platform for Host SDN in the Public Cloud.” In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 2017, pp. 315–328.

[For56] L. R. Ford Jr. *Network flow theory.* Tech. rep. DTIC Document, 1956.

[FR17] N. Feamster and J. Rexford. “Why (and how) networks should run themselves.” In: *arXiv preprint arXiv:1710.11583* (2017).

[FSR06] L. Fu, D. Sun, and L. R. Rilett. “Heuristic shortest path algorithms for transportation applications: State of the art.” In: *Elsevier Computers & Operations Research* 33.11 (Nov. 2006), pp. 3324–3343.

[FSV18] K.-T. Foerster, S. Schmid, and S. Vissicchio. “Survey of consistent software-defined network updates.” In: *IEEE Communications Surveys & Tutorials* 21.2 (2018), pp. 1435–1461.

[FT87] M. L. Fredman and R. E. Tarjan. “Fibonacci heaps and their uses in improved network optimization algorithms.” In: *Journal of the ACM (JACM)* 34.3 (1987), pp. 596–615.

[Gal+15] S. Gallenmüller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. “Comparison of frameworks for high-performance packet IO.” In: *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS).* ACM/IEEE. 2015, pp. 29–38.

[Gen+16] Y. Geng, V. Jeyakumar, A. Kabbani, and M. Alizadeh. “J uggler: a practical reordering resilient network stack for datacenters.” In: *Proceedings of the ACM European Conference on Computer Systems.* ACM. 2016, p. 20.

[GGT10] R. G. Garroppo, S. Giordano, and L. Tavanti. “A survey on multi-constrained optimal path computation: Exact and approximate algorithms.” In: *Elsevier Computer Networks* 54.17 (2010), pp. 3081–3107.

[GH05] A. V. Goldberg and C. Harrelson. “Computing the shortest path: A search meets graph theory.” In: *Proceedings of the ACM SIAM Symposium on Discrete Algorithms.* Society for Industrial and Applied Mathematics. 2005, pp. 156–165.

[GH13] B. Galloway and G. P. Hancke. “Introduction to Industrial Control Networks.” In: *IEEE Communications Surveys & Tutorials* 15.2 (Second Qu. 2013), pp. 860–880.

[Gha+16] M. Ghaznavi, N. Shahriar, R. Ahmed, and R. Boutaba. “Service Function Chaining Simplified.” In: *arXiv preprint arXiv:1601.00751* (2016).

[GHG14] S. Gorlatch, T. Humernbrum, and F. Glinka. “Improving QoS in real-time internet applications: from best-effort to Software-Defined Networks.” In: *International Conference on Computing, Networking and Communications (ICNC).* IEEE. 2014, pp. 189–193.

[Gho+17] S. Ghorbani, Z. Yang, P. Godfrey, Y. Ganjali, and A. Firoozshahian. “DRILL: Micro load balancing for low-latency data center networks.” In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication.* ACM. 2017, pp. 225–238.

[GJF13a] P. Gaj, J. Jasperneite, and M. Felser. “Computer Communication Within Industrial Distributed Environment—a Survey.” In: *IEEE Transactions on Industrial Informatics* 9.1 (Feb. 2013), pp. 182–189.

[GJF13b] P. Gaj, J. Jasperneite, and M. Felser. “Computer communication within industrial distributed environment - A survey.” In: *IEEE Transactions on Industrial Informatics.* Vol. 9. 1. IEEE, 2013, pp. 182–189.

[GM03] L. Guo and I. Matta. “Search space reduction in QoS routing.” In: *Elsevier Computer Networks* 41.1 (2003), pp. 73–88.

[Gom+15] T. Gomes, S. Marques, L. Martins, M. Pascoal, and D. Tipper. “Protected shortest path visiting specified nodes.” In: *Proceedings of the IEEE International Workshop on Reliable Networks Design and Modeling (RNDM).* IEEE. 2015, pp. 120–127.

[Gom+17] T. Gomes, L. Martins, S. Ferreira, M. Pascoal, and D. Tipper. “Algorithms for determining a node-disjoint path pair visiting specified nodes.” In: *Elsevier Optical Switching and Networking* 23 (2017), pp. 189–204.

[GRK15] J. W. Guck, M. Reisslein, and W. Kellerer. “Model-based control plane for fast routing in industrial QoS network.” In: *Proceedings of the IEEE International Symposium on Quality of Service (IWQoS).* IEEE. 2015, pp. 65–66.

[GRK16] J. W. Guck, M. Reisslein, and W. Kellerer. “Function Split Between Delay-Constrained Routing and Resource Allocation for Centrally Managed QoS in Industrial Networks.” In: *IEEE Transactions on Industrial Informatics* 12.6 (Dec. 2016), pp. 2050–2061.

[Gro+15] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. “Queues Don’t Matter When You Can JUMP Them!” In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI).* USENIX. 2015, pp. 1–14.

[Guc18] J. W. Guck. “Centralized Online Routing for Deterministic Quality of Service in Packet Switched Networks.” PhD thesis. Technische Universität München, 2018.

[Gun+11] V. C. Gungor, D. Sahin, T. Kocak, S. Ergut, C. Buccella, C. Cecati, and G. P. Hancke. “Smart grid technologies: communication technologies and standards.” In: *IEEE Transactions on Industrial Informatics* 7.4 (2011), pp. 529–539.

[Guo+10] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. “Secondnet: a data center network virtualization architecture with bandwidth guarantees.” In: *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT).* ACM. 2010, p. 15.

[GY05] J. L. Gross and J. Yellen. *Graph theory and its applications.* CRC Press, 2005.

[GYG13] A. Gelberger, N. Yemini, and R. Giladi. “Performance analysis of software-defined networking (SDN).” In: *Proceedings of the IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS).* IEEE. 2013, pp. 389–393.

[Han+17] M. Handley, C. Raiciu, A. Agache, A. Voinescu, A. W. Moore, G. Antichi, and M. Wójcik. “Re-architecting datacenter networks and stacks for low latency and high performance.” In: *Proceedings of ACM SIGCOMM.* ACM. 2017, pp. 29–42.

[HCG12] C.-Y. Hong, M. Caesar, and P. B. Godfrey. “Finishing flows quickly with preemptive scheduling.” In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 127–138.

[He+15a] K. He, J. Khalid, S. Das, A. Gember-Jacobson, C. Prakash, A. Akella, L. E. Li, and M. Thottan. "Latency in software defined networks: Measurements and mitigation techniques." In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 43. 1. ACM. 2015, pp. 435–436.

[He+15b] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan. "Measuring control plane latency in sdn-enabled switches." In: *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*. ACM. 2015, p. 25.

[He+15c] K. He, E. Rozner, K. Agarwal, W. Felter, J. Carter, and A. Akella. "Presto: Edge-based load balancing for fast datacenter networks." In: *ACM SIGCOMM Computer Communication Review*. Vol. 45. 4. ACM. 2015, pp. 465–478.

[HNR68] P. E. Hart, N. J. Nilsson, and B. Raphael. "A formal basis for the heuristic determination of minimum cost paths." In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107.

[Hos+07] M. Hosseini, D. T. Ahmed, S. Shirmohammadi, and N. D. Georganas. "A survey of application-layer multicast protocols." In: *IEEE Communications Surveys & Tutorials* 9.3 (Third Qu. 2007), pp. 58–74.

[HS09] M. Hilger and H. Schilling. "Fast point-to-point shortest path computations with arc-flags." In: *The Shortest Path Problem: Ninth DIMACS Implementation Challenge* 74 (2009), pp. 41–72.

[HSW09] M. Holzer, F. Schulz, and D. Wagner. "Engineering multilevel overlay graphs for shortest-path queries." In: *ACM Journal of Experimental Algorithmics (JEA)* 13 (2009), p. 5.

[Hu+16] S. Hu, W. Bai, K. Chen, C. Tian, Y. Zhang, and H. Wu. "Providing bandwidth guarantees, work conservation and low latency simultaneously in the cloud." In: *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. 2016, pp. 1–9.

[HWJ16] D. Henneke, L. Wisniewski, and J. Jasperneite. "Analysis of realizing a future industrial network by means of Software-Defined Networking (SDN)." In: *Proceedings of the IEEE World Conference on Factory Communication Systems (WFCS)*. IEEE. 2016, pp. 1–4.

[HYS13] D. Y. Huang, K. Yocum, and A. C. Snoeren. "High-fidelity switch models for software-defined network emulation." In: *Proceedings of the ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. ACM. 2013, pp. 43–48.

[HZ80] G. Y. Handler and I. Zang. "A dual algorithm for the constrained shortest path problem." In: *Wiley Networks* 10.4 (1980), pp. 293–309.

[IAK98] K. Ishida, K. Amano, and N. Kannari. "A delay-constrained least-cost path routing protocol and the synthesis method." In: *Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications*. 1998, pp. 58–65.

[Iba73] T. Ibaraki. "Algorithms for obtaining shortest paths visiting specified nodes." In: *SIAM Review* 15.2 (1973), pp. 309–317.

[IEC18] IEC/IEEE 60802 Joint Project. *Use Cases IEC/IEEE 60802 v1.3.* http://www.ieee802.org/1/files/public/docs2018/60802-industrial-use-cases-0918-v13.pdf. Accessed: 2020-01-30. 2018.

[IEE] IEEE. *Time-Sensitive Networking (TSN) Task Group.* URL: https://1.ieee802.org/tsn/.

[Ike+94] T. Ikeda, M.-Y. Hsu, H. Imai, S. Nishimura, H. Shimoura, T. Hashimoto, K. Tenmoku, and K. Mitoh. "A fast algorithm for finding better routes by AI search techniques." In: *Proceedings of the IEEE Vehicle Navigation and Information Systems Conference.* IEEE. 1994, pp. 291–296.

[ITU15] ITU-R: Radiocommunication Sector of ITU. "IMT Vision–Framework and overall objectives of the future development of IMT for 2020 and beyond." In: *Rec. ITU-R M.2083-0* (2015).

[Jal+13] V. Jalaparti, P. Bodik, S. Kandula, I. Menache, M. Rybalkin, and C. Yan. "Speeding up distributed request-response workflows." In: *ACM SIGCOMM Computer Communication Review.* Vol. 43. 4. ACM. 2013, pp. 219–230.

[Jan+15] K. Jang, J. Sherry, H. Ballani, and T. Moncaster. "Silo: Predictable message latency in the cloud." In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 435–448.

[Jar+11] M. Jarschel, S. Oechsner, D. Schlosser, R. Pries, S. Goll, and P. Tran-Gia. "Modeling and performance evaluation of an OpenFlow architecture." In: *Proceedings of the International Teletraffic Congress.* International Teletraffic Congress. 2011, pp. 1–7.

[Jey+13] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. "EyeQ: Practical network performance isolation at the edge." In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI).* USENIX, 2013, pp. 297–311.

[Jia06] Y. Jiang. "A Basic Stochastic Network Calculus." In: *ACM SIGCOMM Computer Communication Review* 36.4 (2006), pp. 123–134.

[JL08] Y. Jiang and Y. Liu. *Stochastic Network Calculus.* Springer, 2008.

[JN04] J. Jasperneite and P. Neumann. "How to guarantee realtime Behavior using Ethernet." In: *Proceedings of the IFAC Symposium on Information Control Problems in Manufacturing (INCOM).* Gulf Professional Publishing. 2004.

[Jos+18] R. Joshi, T. Qu, M. C. Chan, B. Leong, and B. T. Loo. "BurstRadar: Practical real-time microburst monitoring for datacenter networks." In: *Proceedings of the ACM Asia-Pacific Workshop on Systems.* ACM. 2018, pp. 1–8.

[JSQ02] G. R. Jagadeesh, T. Srikanthan, and K. Quek. "Heuristic techniques for accelerating hierarchical routing on road networks." In: *IEEE Transactions on Intelligent Transportation Systems* 3.4 (2002), pp. 301–309.

[Jüt+01] A. Jüttner, B. Szviatovski, I. Mécs, and Z. Rajkó. "Lagrange relaxation based method for the QoS routing problem." In: *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM).* Vol. 2. 2001, pp. 859–868.

[JV01] Z. Jia and P. Varaiya. “Heuristic Methods for Delay-Constrained Least-Cost Routing Problem Using k-Shortest-Path Algorithms.” In: *IEEE International Conference on Computer Communications (INFOCOM).* 2001, pp. 1–9.

[Kas+12] A. Kassler, L. Skorin-Kapov, O. Dobrijevic, M. Matijasevic, and P. Dely. “Towards QoE-driven multimedia service negotiation and path optimization with software defined networking.” In: *Proceedings of the IEEE International Conference on Software, Telecommunications and Computer Networks (SoftCOM).* IEEE. Sept. 2012, pp. 1–5.

[Kat+17b] N. Katta, A. Ghag, M. Hira, I. Keslassy, A. Bergman, C. Kim, and J. Rexford. “Clove: Congestion-aware load balancing at the virtual edge.” In: *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT).* ACM. 2017, pp. 323–335.

[KCL14] A. L. King, S. Chen, and I. Lee. “The middleware assurance substrate: Enabling strong real-time guarantees in open systems with OpenFlow.” In: *Proceedings of the IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC).* IEEE. 2014, pp. 133–140.

[Kel+19] W. Kellerer, P. Kalmbach, A. Blenk, A. Basta, M. Reisslein, and S. Schmid. “Adaptable and data-driven softwarized networks: Review, opportunities, and challenges.” In: *Proceedings of the IEEE* 107.4 (2019), pp. 711–731.

[Kim+10] W. Kim, P. Sharma, J. Lee, S. Banerjee, J. Tourrilhes, S.-J. Lee, and P. Yalagandula. “Automated and Scalable QoS Control for Network Convergence.” In: *Proceedings of the USENIX Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN).* Vol. 10. 1. 2010, pp. 1–1.

[KK01] T. Korkmaz and M. Krunz. “Multi-constrained optimal path selection.” In: *IEEE International Conference on Computer Communications (INFOCOM).* Vol. 2. 2001, pp. 834–843.

[KK13] R. H. Khan and J. Y. Khan. “A comprehensive review of the application characteristics and traffic requirements of a smart grid communications network.” In: *Elsevier Computer Networks* 57.3 (2013), pp. 825–845.

[KK77] L. Kleinrock and F. Kamoun. “Hierarchical routing for large networks performance evaluation and optimization.” In: *Elsevier Computer Networks* 1.3 (1977), pp. 155158–174.

[Kni+11] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. “The Internet Topology Zoo.” In: *IEEE Journal on Selected Areas in Communications* 29.9 (Oct. 2011), pp. 1765–1775.

[KOK14] A. A. Kumar S., K. Ovsthus, and L. M. Kristensen. “An Industrial Perspective on Wireless Sensor Networks—A Survey of Requirements, Protocols, and Challenges.” In: *IEEE Communications Surveys & Tutorials* 16.3 (Third Qu. 2014), pp. 1391–1412.

[KPK14] M. Kuźniar, P. Perešíni, and D. Kostić. *What you need to know about SDN control and data planes.* Tech. rep. EPFL-REPORT-199497. École Polytechnique Fédérale de Lausanne (EPFL), 2014.

[KPK15] M. Kuźniar, P. Perešíni, and D. Kostić. "What you need to know about SDN flow tables." In: *Proceedings of the International Conference on Passive and Active Network Measurement.* Springer. 2015, pp. 347–359.

[KR13] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-Down Approach.* 6th ed. Pearson Education, 2013.

[KRR16] A. A. Khan, M. H. Rehmani, and M. Reisslein. "Cognitive radio for smart grids: Survey of architectures, spectrum sensing mechanisms, and networking protocols." In: *IEEE Communications Surveys & Tutorials* 18.1 (Jan. 2016), pp. 860–898.

[KS98] S. Keshav and R. Sharma. "Issues and trends in router design." In: *IEEE Communications Magazine.* Vol. 36. 5. IEEE, 1998, pp. 144–151.

[Kum+15] A. Kumar et al. "BwE: Flexible, hierarchical bandwidth allocation for WAN distributed computing." In: *ACM SIGCOMM Computer Communication Review* 45.4 (2015), pp. 1–14.

[Kuo+16] T.-W. Kuo, B.-H. Liou, K. C.-J. Lin, and M.-J. Tsai. "Deploying chains of virtual network functions: On the relation between link and server usage." In: *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM).* Apr. 2016, pp. 1–9.

[Kuź+18] M. Kuźniar, P. Perešíni, D. Kostić, and M. Canini. "Methodology, measurement and analysis of flow table update characteristics in hardware openflow switches." In: *Elsevier Computer Networks* 136 (2018), pp. 22–36.

[Kwa89] J. B. Kwa. "BS*: An admissible bidirectional staged heuristic search algorithm." In: *Elsevier Artificial Intelligence* 38.1 (1989), pp. 95–109.

[Lam+12] V. T. Lam, S. Radhakrishnan, R. Pan, A. Vahdat, and G. Varghese. "Netshare and stochastic netshare: predictable bandwidth allocation for data centers." In: *ACM SIGCOMM Computer Communication Review* 42.3 (2012), pp. 5–11.

[Laz+14] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu. "Tango: Simplifying SDN control with automatic switch property inference, abstraction, and optimization." In: *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT).* ACM. 2014, pp. 199–212.

[Lee+14] J. Lee, Y. Turner, M. Lee, L. Popa, S. Banerjee, J.-M. Kang, and P. Sharma. "Application-driven bandwidth guarantees in datacenters." In: *ACM SIGCOMM Computer Communication Review.* Vol. 44. 4. ACM. 2014, pp. 467–478.

[LHH95] W. C. Lee, M. G. Hluchyi, and P. A. Humblet. "Routing subject to quality of service constraints in integrated communication networks." In: *IEEE Network* 9.4 (July 1995), pp. 46–55.

[Li+17] J. Q. Li, F. R. Yu, G. Deng, C. Luo, Z. Ming, and Q. Yan. "Industrial Internet: A Survey on the Enabling Technologies, Applications, and Challenges." In: *IEEE Communications Surveys & Tutorials* 19.3 (Third Qu. 2017), pp. 1504–1526.

[Li+19] Y. Li et al. "HPCC: high precision congestion control." In: *Proceedings of ACM SIGCOMM.* 2019, pp. 44–58.

[Lin] Linux man pages. *tc(8): show/change traffic control settings - Linux man page*. https://linux.die.net/man/8/tc. Accessed: 2018-10-18.

[Lin+17] J. Lin, W. Yu, N. Zhang, X. Yang, H. Zhang, and W. Zhao. “A survey on internet of things: Architecture, enabling technologies, security and privacy, and applications.” In: *IEEE Internet of Things Journal* (2017).

[Lin+18] Y.-D. Lin, Y.-K. Lai, C.-Y. Wang, and Y.-C. Lai. “OFBench: Performance test suite on OpenFlow switches.” In: *IEEE Systems Journal* 12.3 (2018), pp. 2949–2959.

[Liu+16] F. Liu, J. Guo, X. Huang, and J. C. Lui. “eBA: Efficient bandwidth guarantee under traffic variability in datacenters.” In: *IEEE/ACM Transactions on Networking* 25.1 (2016), pp. 506–519.

[Liu+18] Z. Liu, K. Chen, H. Wu, S. Hu, Y.-C. Hut, Y. Wang, and G. Zhang. “Enabling Work-Conserving Bandwidth Guarantees for Multi-Tenant Datacenters via Dynamic Tenant-Queue Binding.” In: *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. 2018, pp. 1–9.

[LL09] A. Lingas and E.-M. Lundell. “Efficient approximation algorithms for shortest cycles in undirected graphs.” In: *Elsevier Information Processing Letters* 109.10 (2009), pp. 493–498.

[LLF05] W. Liu, W. Lou, and Y. Fang. “An efficient quality of service routing algorithm for delay-sensitive applications.” In: *Elsevier Computer Networks* 47.1 (Jan. 2005), pp. 87–104.

[LR01] G. Liu and K. Ramakrishnan. “A*Prune: An algorithm for finding K shortest paths subject to multiple constraints.” In: *IEEE International Conference on Computer Communications (INFOCOM)*. Vol. 2. 2001, pp. 743–749.

[LT12] J.-Y. Le Boudec and P. Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer, Apr. 2012.

[MA04] R. T. Marler and J. S. Arora. “Survey of multi-objective optimization methods for engineering.” In: *Springer Structural and Multidisciplinary Optimization* 26.6 (Apr. 2004), pp. 369–395.

[Mah+16] T. Mahmoodi et al. “VirtuWind: virtual and programmable industrial network prototype deployed in operational wind park.” In: *Wiley Transactions on Emerging Telecommunications Technologies* 27.9 (2016), pp. 1281–1288.

[McK+08] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. “OpenFlow: enabling innovation in campus networks.” In: *SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[Mel+19] W. M. Mellette, R. Das, Y. Guo, R. McGuinness, A. C. Snoeren, and G. Porter. “Expanding across time to deliver bandwidth efficiency and low latency.” In: *arXiv preprint arXiv:1903.12307* (2019).

[MGT16] L. Martins, T. Gomes, and D. Tipper. “An efficient heuristic for calculating a protected path with specified nodes.” In: *Proceedings of the IEEE International Workshop on Reliable Networks Design and Modeling (RNDM)*. IEEE. 2016, pp. 150–157.

[Mic17] Microchip Technology Inc. *Microchip KSZ8795CLX*. http://ww1.microchip.com/downloads/en/DeviceDoc/00002112B.pdf. Accessed: 2018-10-26. 2017.

[Mij+15] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba. "Network function virtualization: state-of-the-art and research challenges." In: *IEEE Communications Surveys & Tutorials* 18.1 (2015), pp. 236–262.

[MMB17] D. S. Marcon, F. M. Mazzola, and M. P. Barcellos. "Achieving minimum bandwidth guarantees and work-conservation in large-scale, SDN-based datacenter networks." In: *Elsevier Computer Networks* 127 (2017), pp. 109–125.

[MMP15] W. Ma, C. Medina, and D. Pan. "Traffic-Aware Placement of NFV Middleboxes." In: *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM)*. IEEE. 2015, pp. 1–6.

[MOA11] A. W. Marshall, I. Olkin, and B. C. Arnold. *Inequalities: Theory of Majorization and Its Applications*. 2nd ed. Springer Series in Statistics, 2011.

[Mon+18] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. "Homa: A receiver-driven low-latency transport protocol using network priorities." In: *Proceedings of ACM SIGCOMM*. ACM. 2018, pp. 221–235.

[Moo59] E. F. Moore. *The Shortest Path Through a Maze*. Bell Telephone System, 1959.

[Mor17] A. Morton. *Updates for the Back-to-back Frame Benchmark in RFC 2544*. Internet-Draft draft-morton-bmwg-b2b-frame-00. IETF Secretariat, Oct. 2017. URL: http://www.ietf.org/internet-drafts/draft-morton-bmwg-b2b-frame-00.txt.

[MP12] J. C. Mogul and L. Popa. "What we talk about when we talk about cloud network performance." In: *ACM SIGCOMM Computer Communication Review* 42.5 (2012), pp. 44–48.

[MP18] R. McGuinness and G. Porter. "Evaluating the performance of software NICs for 100-Gb/s datacenter traffic control." In: *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM/IEEE. 2018, pp. 74–88.

[Nao+08] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. "Implementing an OpenFlow switch on the NetFPGA platform." In: *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM/IEEE. Nov. 2008, pp. 1–9.

[Nor18] Northbound Networks. *GitHub - NorthboundNetworks/ZodiacFX: Firmware for the Northbound Networks Zodiac FX OpenFlow Switch*. https://github.com/NorthboundNetworks/ZodiacFX. Accessed: 2018-08-03. 2018.

[Nor19] Northbound Networks. *Zodiac FX Switch Hardware*. https://northboundnetworks.com/products/zodiac-fx. Accessed: 2019-03-18. 2019.

[OD+13] I. Owens, A. Durresi, et al. "Video over Software-Defined Networking (VSDN)." In: *Proceedings of the IEEE International Conference on Network-Based Information Systems (NBiS)*. IEEE. 2013, pp. 44–51.

[Ope09] Open Networking Foundation (ONF). *OpenFlow Switch Specification Version 1.0.0 (ONF TS-001).* https://www.opennetworking.org/wp-content/uploads/2013/04/openflow-spec-v1.0.0.pdf. 2009.

[PBS17] J. Perry, H. Balakrishnan, and D. Shah. “Flowtune: Flowlet control for datacenter networks.” In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI).* 2017, pp. 421–435.

[Per+14] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. “Fastpass: A centralized zero-queue datacenter network.” In: *ACM SIGCOMM Computer Communication Review.* Vol. 44. 4. 2014, pp. 307–318.

[Pfa+15] B. Pfaff et al. “The Design and Implementation of Open vSwitch.” In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI).* Vol. 15. 2015, pp. 117–130.

[PMK13] G. Pongrácz, L. Molnár, and Z. L. Kis. “Removing roadblocks from SDN: OpenFlow software switch performance on Intel DPDK.” In: *Proceedings of the IEEE European Workshop on Software Defined Networks (EWSDN).* IEEE. 2013, pp. 62–67.

[Pop+12] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. “FairCloud: sharing the network in cloud computing.” In: *ACM SIGCOMM Computer Communication Review.* Vol. 42. 4. 2012, pp. 187–198.

[Pop+13] L. Popa, P. Yalagandula, S. Banerjee, J. C. Mogul, Y. Turner, and J. R. Santos. “Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing.” In: *ACM SIGCOMM Computer Communication Review.* Vol. 43. 4. ACM. 2013, pp. 351–362.

[Pop19] D. A. Popescu. “Latency-driven performance in data centres.” PhD thesis. University of Cambridge, 2019.

[PR02] P. Paul and S. V. Raghavan. “Survey of QoS Routing.” In: *Proceedings of the ACM International Conference on Computer Communication (ICCC).* ACM. Mumbai, Maharashtra, India, 2002, pp. 50–75.

[Rah+16] R. Rahimi, M. Veeraraghavan, Y. Nakajima, H. Takahashi, S. Okamoto, and N. Yamanaka. “A high-performance OpenFlow software switch.” In: *Proceedings of the IEEE International Conference on High Performance Switching and Routing (HPSR).* IEEE. 2016, pp. 93–99.

[Ram00] M. Ramalho. “Intra- and inter-domain multicast routing protocols: A survey and taxonomy.” In: *IEEE Communications Surveys & Tutorials* 3.1 (First Qu. 2000), pp. 2–25.

[Rau+16] D. Raumer, S. Gallenmüller, F. Wohlfart, P. Emmerich, P. Werneck, and G. Carle. “Revisiting benchmarking methodology for interconnect devices.” In: *Proceedings of the ACM/IRTF Applied Networking Research Workshop (ANRW).* ACM/IRTF. Berlin, Germany, 2016, pp. 55–61.

[Rod+11] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. O. Guedes. “Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks.” In: vol. 1. 3. 2011, pp. 784–789.

[Ros50] R. Rosenbaum. “Sub-additive Functions.” In: *Duke Mathematical Journal* 17.3 (1950), pp. 227–247.

[Rot+12] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. “OFLOPS: An open framework for OpenFlow switch evaluation.” In: *Proceedings of the International Conference on Passive and Active Network Measurement.* Springer. 2012, pp. 85–95.

[RS00] D. S. Reeves and H. F. Salama. “A distributed algorithm for delay-constrained unicast routing.” In: *IEEE/ACM Transactions on Networking* 8.2 (Apr. 2000), pp. 239–250.

[RS16] M. Rost and S. Schmid. “Service chain and virtual network embeddings: Approximations using randomized rounding.” In: *arXiv preprint arXiv:1604.02180* (2016).

[RT12] M. N. Rice and V. J. Tsotras. “Bidirectional A* Search with Additive Approximation Bounds.” In: *Proceedings of the AAAI Symposium on Combinatorial Search (SOCS).* 2012.

[RT13] M. N. Rice and V. J. Tsotras. “Parameterized algorithms for generalized traveling salesman problems in road networks.” In: *Proceedings of the ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM. 2013, pp. 114–123.

[Ryu17] Ryu SDN Framework Community. *Ryu SDN Framework.* https://osrg.github.io/ryu/. Accessed: 2018-10-26. 2017.

[Sau10] T. Sauter. “The three generations of field-level networks - evolution and compatibility issues.” In: *IEEE Transactions on Industrial Electronics.* Vol. 57. 11. IEEE, 2010, pp. 3585–3595.

[SCC07] L. Santos, J. Coutinho-Rodrigues, and J. R. Current. “An improved solution algorithm for the constrained shortest path problem.” In: *Elsevier Transportation Research Part B: Methodological* 41.7 (Aug. 2007), pp. 756–771.

[Sch+03] J. Schmitt, P. Hurley, M. Hollick, and R. Steinmetz. “Per-flow guarantees under class-based priority queueing.” In: *Proceedings of the IEEE Global Telecommunications Conference (GLOBECOM).* Vol. 7. IEEE. 2003, pp. 4169–4174.

[Sch+16] E. Schweissguth, P. Danielis, C. Niemann, and D. Timmermann. “Application-aware industrial ethernet based on an SDN-supported TDMA approach.” In: *Proceedings of the IEEE World Conference on Factory Communication Systems (WFCS).* IEEE. 2016, pp. 1–8.

[SDJ15] S. K. Singh, T. Das, and A. Jukan. “A Survey on Internet Multipath Routing and Provisioning.” In: *IEEE Communications Surveys & Tutorials* 17.4 (Fourth Qu. 2015), pp. 2157–2175.

[Sha+13] P. Sharma, S. Banerjee, S. Tandel, R. Aguiar, R. Amorim, and D. Pinheiro. “Enhancing network management frameworks with SDN-like control.” In: *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management (IM).* IFIP/IEEE. 2013, pp. 688–691.

[Sha+14] S. Sharma et al. “Implementing quality of service for the software defined networking enabled future Internet.” In: *Proceedings of the IEEE European Workshop on Software Defined Networks.* IEEE. 2014, pp. 49–54.

[Sha+17] M. Shafi et al. "5G: A tutorial overview of standards, trials, challenges, deployment, and practice." In: *IEEE Journal on Selected Areas in Communications* 35.6 (2017), pp. 1201–1221.

[She+16] M. Shen, L. Zhu, M. Wei, Q. Zhang, M. Wang, and F. Li. "Joint Optimization of Flow Latency in Routing and Scheduling for Software Defined Networks." In: *Proceedings of the IEEE International Conference on Computer Communication and Networks (ICCCN)*. IEEE. 2016, pp. 1–8.

[Shi+11] A. Shieh, S. Kandula, A. G. Greenberg, C. Kim, and B. Saha. "Sharing the Data Center Network." In: *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Vol. 11. 2011, pp. 23–23.

[Shi54] A. Shimbel. "Structure in communication nets." In: *Proceedings of the Symposium on Information Networks*. 1954, pp. 199–203.

[Sin18a] Sinovoip. *Banana Pi BPI-R1 Open-source Router*. http://www.banana-pi.org/r1.html. Accessed: 2018-10-24. 2016-2018.

[Sin18b] Sinovoip. *Banana Pi BPI-R2 Open-source Router*. http://www.banana-pi.org/r2.html. Accessed: 2018-10-24. 2016-2018.

[SJH06] T. Skeie, S. Johannessen, and O. Holmeide. "Timeliness of real-time IP communication in switched industrial Ethernet networks." In: *IEEE Transactions on Industrial Informatics* 2.1 (Feb. 2006), pp. 25–39.

[SK18] E. Sakic and W. Kellerer. "Impact of adaptive consistency on distributed sdn applications: An empirical study." In: *IEEE Journal on Selected Areas in Communications* 36.12 (2018), pp. 2702–2715.

[SK66] J. Saksena and S. Kumar. "The Routing Problem with "K" Specified Nodes." In: *INFORMS Operations Research* 14.5 (1966), pp. 909–913.

[SL98] Q. Sun and H. Langendörfer. "A new distributed routing algorithm for supporting delay-sensitive applications." In: *Elsevier Computer Communications* 21.6 (May 1998), pp. 572–578.

[SMM98] R. Sriram, G. Manimaran, and C. S. R. Murthy. "Preferred link based delay-constrained least-cost routing in wide area networks." In: *Elsevier Computer Communications* 21.18 (Dec. 1998), pp. 1655–1669.

[Som+10] J. Sommer, S. Gunreben, F. Feller, M. Kohn, A. Mifdaoui, D. Saß, and J. Scharf. "Ethernet–a survey on its fields of application." In: *IEEE Communications Surveys & Tutorials*. Vol. 12. 2. IEEE, 2010, pp. 263–284.

[SPD14] P. Simari, G. Picciau, and L. De Floriani. "Fast and scalable mesh superfacets." In: *Computer Graphics Forum*. Vol. 33. 7. Wiley Online Library. 2014, pp. 181–190.

[SRV97] H. F. Salama, D. S. Reeves, and Y. Viniotis. "A distributed algorithm for delay-constrained unicast routing." In: *IEEE International Conference on Computer Communications (INFOCOM)*. Vol. 1. 1997, pp. 84–91.

[ST83] I. Stewart and D. Tall. *Complex Analysis (The Hitchhiker's Guide to the Plane).* Cambridge University Press, 1983.

[Sta15] W. Stallings. *Foundations of modern networking: SDN, NFV, QoE, IoT, and Cloud.* Addison-Wesley Professional, 2015.

[STV12] L. Seno, F. Tramarin, and S. Vitturi. "Performance of industrial communication systems: Real application contexts." In: *IEEE Industrial Electronics Magazine* 6.2 (June 2012), pp. 27–37.

[SV08] T. Schuster and D. Verma. "Networking concepts comparison for avionics architecture." In: *Proceedings of the IEEE/AIAA Digital Avionics Systems Conference.* Oct. 2008, pp. 1.D.1-1-1.D.1–11.

[Tho03] M. Thorup. "Integer priority queues with decrease key in constant time and the single source shortest paths problem." In: *Proceedings of the ACM Symposium on Theory of Computing.* ACM. 2003, pp. 149–158.

[TPR14] S. Tomovic, N. Prasad, and I. Radusinovic. "SDN control framework for QoS provisioning." In: *Proceedings of the IEEE Telecommunications Forum (TELFOR).* IEEE. 2014, pp. 111–114.

[Tri11] TriaGnoSys GmbH. *Onair and TriaGnoSys launch most lightweight inflight connectivity solution for business jets.* http://triagnosys.com/assets/PressReleases/OnAirTGSbizjet.pdf. Accessed: 2019-01-31. May 2011.

[TW11] A. S. Tanenbaum and D. J. Wetherall. *Computer Networks.* 5th ed. Prentice Hall, 2011.

[US] US Naval Research Laboratory. *Multi-Generator (MGEN) | Networks and Communication Systems Branch.* https://www.nrl.navy.mil/itd/ncs/products/mgen. Accessed: 2018-10-18.

[Var14] S. Varone. *On a many-to-one shortest paths for a taxi service.* Tech. rep. HES-SO/HEG-GE/C–14/1/1-CH. Haute école de gestion de Genève, 2014.

[Vat+12] B. C. Vattikonda, G. Porter, A. Vahdat, and A. C. Snoeren. "Practical TDMA for datacenter ethernet." In: *Proceedings of the ACM European Conference on Computer Systems.* ACM. 2012, pp. 225–238.

[VHV12] B. Vamanan, J. Hasan, and T. N. Vijaykumar. "Deadline-aware datacenter TCP (D2TCP)." In: *ACM SIGCOMM Computer Communication Review.* Vol. 42. 4. ACM, 2012, pp. 115–126.

[Vir17] VirtuWind EU Project. *VirtuWind - Deliverable D3.2: Detailed Intra-Domain SDN & NFV Architecture.* http://www.virtuwind.eu/. Accessed: 2020-01-30. 2017.

[Viz+17] P. Vizarreta, M. Condoluci, C. Mas Machuca, T. Mahmoodi, and W. Kellerer. "QoS-driven Function Placement Reducing Expenditures in NFV Deployments." In: *Proceedings of the IEEE International Conference on Communications (ICC).* 2017.

[VJ14] S. Varone and V. Janilionis. "Insertion heuristic for a dynamic dial-a-ride problem using geographical maps." In: *Proceedings of the Conférence Francophone de Modélisation, Optimisation et Simulation (MOSIM).* 2014.

[Wax88] B. M. Waxman. "Routing of multipoint connections." In: *IEEE Journal on Selected Areas in Communications* 6.9 (Dec. 1988), pp. 1617–1622.

[WH00] B. Wang and J. C. Hou. "Multicast routing and its QoS extension: Problems, algorithms, and protocols." In: *IEEE Network* 14.1 (Jan. 2000), pp. 22–36.

[Wid94] R. Widyono. *The design and evaluation of routing algorithms for real-time channels.* Tech. rep. TR-94-024. International Computer Science Institute Berkeley, 1994.

[Wil+11] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. "Better never than late: Meeting deadlines in datacenter networks." In: *ACM SIGCOMM Computer Communication Review.* Vol. 41. 4. ACM. 2011, pp. 50–61.

[WMZ19] J. Woodruff, A. W. Moore, and N. Zilberman. "Measuring Burstiness in Data Center Applications." In: *Proceedings of the ACM Workshop on Buffer Sizing.* 2019.

[Xie+12] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. "The only constant is change: Incorporating time-varying network reservations in data centers." In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 199–210.

[Yan+12] Y. Yan, Y. Qian, H. Sharif, and D. Tipper. "A survey on smart grid communication infrastructures: Motivations, requirements and challenges." In: *IEEE Communications Surveys & Tutorials* 15.1 (2012), pp. 5–20.

[Yaq+17] I. Yaqoob, E. Ahmed, I. A. T. Hashem, A. I. A. Ahmed, A. Gani, M. Imran, and M. Guizani. "Internet of things architecture: Recent advances, taxonomy, requirements, and open challenges." In: *IEEE Wireless Communications* 24.3 (June 2017), pp. 10–16.

[Yen70] J. Y. Yen. "An algorithm for finding shortest routes from all source nodes to a given destination in general networks." In: *AMS Quarterly of Applied Mathematics* 27.4 (Jan. 1970), pp. 526–530.

[Yen71] J. Y. Yen. "Finding the *k* shortest loopless paths in a network." In: *INFORMS Management Science* 17.11 (1971), pp. 712–716.

[Zah+19] E. Zahavi, A. Shpiner, O. Rottenstreich, A. Kolodny, and I. Keslassy. "Links as a Service (LaaS): Guaranteed tenant isolation in the shared cloud." In: *IEEE Journal on Selected Areas in Communications* 37.5 (2019), pp. 1072–1084.

[Zat+12] D. Zats, T. Das, P. Mohan, D. Borthakur, and R. Katz. "DeTail: reducing the flow completion time tail in datacenter networks." In: *Proceedings of ACM SIGCOMM.* ACM. 2012, pp. 139–150.

[ZBC19] J. Zhang, W. Bai, and K. Chen. "Enabling ECN for datacenter networks with RTT variations." In: *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT).* 2019, pp. 233–245.

[ZPG95] Q. Zhu, M. Parsa, and J. Garcia-Luna-Aceves. "A source-based algorithm for delay-constrained minimum-cost multicasting." In: *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM).* Vol. 1. 1995, pp. 377–385.

Acronyms and Abbreviations

***k*CSP** *k* constrained shortest paths 66

***k*MCP** *k* multi-constrained paths 66

***k*MCSP** *k* multi-constrained shortest paths 66

***k*SP** *k* shortest paths 66, 68, 84, 85, 87

***k*SPT** *k* shortest paths tree 84–86

ACL access control list 128

API application program interface 145

AS autonomous system 188

ASIC application-specific integrated circuit 120, 123, 130

AUT algorithm under test 73

BD bounded Dijkstra 6, 64, 80–94

BF Bellman-Ford 64, 67–70, 81, 82, 84–87, 106

BFS breadth-first search 67, 104, 106

BW bandwidth 12

CAT cache allocation technology 148

CBR constant bit rate 34

CLI command line interface 161, 164

CMST constrained minimum Steiner tree 80, 82

CP control plane 2, 4, 9, 10, 14, 43, 60, 160, 164–167, 173, 176–178, 180

CPS cyber-physical system 1, 185, 187

CPU central processing unit 3, 57, 78, 87, 101, 112, 117, 120, 124, 125, 147–149, 151, 159–161, 163, 164, 167, 170–173, 180, 181, 186

CSP constrained shortest path 63, 64, 66–71, 74, 76, 78–80, 82, 85–88, 90, 91, 93–99, 101, 102, 106, 111–114

D^{2}TCP deadline-aware datacenter TCP 12, 13

D^3 deadline-driven delivery 12, 13

DAG data acquisition and generation 121, 153, 154, 161, 164, 165, 167

DCLC delay-constrained least-cost 4, 57, 63, 69, 82, 143, 146, 147, 152

DCTCP data center TCP 12, 13

DetServ deterministic services 4, 6, 7, 9, 10, 12, 15, 43, 44, 54, 56, 58–61, 64, 68–70, 78, 79, 93, 95, 96, 103–107, 115, 134, 158, 159, 163, 173, 174, 181

DNC deterministic network calculus 7, 9, 10, 15–17, 19, 23, 24, 26, 30, 32, 37, 41, 42, 44, 53, 106, 107, 118, 124–126, 132, 136, 139, 141, 142, 148, 154, 158, 163, 172–175, 181, 187

DP data plane 2, 4, 9, 10, 14, 41, 160, 161, 164, 166–169, 171, 176, 180

DPDK data plane development kit 6, 7, 145, 147–149, 152, 153, 171, 180, 181, 186

DPI deep packet inspection 94

E2E end-to-end 1–4, 7, 9, 10, 21, 37, 38, 40–43, 45, 54, 59, 60, 65, 69, 117, 120, 129, 139, 154, 158, 173, 176, 181, 185–187

EBD edge-based Dijkstra 6, 64, 103, 109, 113–115

ECDF empirical cumulative distribution function 58, 112–114, 152

ECN early congestion notification 12–14

EDF earliest deadline first 13

EEPROM electrically erasable programmable read-only memory 160

FCT flow completion time 12, 13

FIFO first-in first-out 31, 36, 38

FPP function placement problem 94

GPS generalized processor sharing 37

GR grid random 72–74, 76

GTA graph transformation algorithm 6, 64, 103, 109–115

HPCC high precision congestion control 12, 14

HTTP hypertext transfer protocol 145, 160, 161, 164

HUG high utilization with guarantees 12, 13

HULL high-bandwidth ultra-low latency 12, 14

i*k*SP iterative *k* shortest paths 68, 70, 73–75, 77, 79

I/O input/output 57, 72

IEEE institute of electrical and electronics engineers 15

IFG interframe gap 170

ILP integer linear program 96

ILS input link shaping 52–56, 58, 104–107

IntServ integrated services 35

IoT internet of things 1, 185

IP Internet protocol 11, 134, 148, 161

IS integrated switch 159–161, 164, 169–172

ISP Internet service provider 188

L2 layer-2 121, 124, 128, 159–161, 170, 171

L3 layer-3 121, 124, 128

L4 layer-4 13, 14, 121, 124, 161

LaaS links as a service 12, 14

LAN local area network 11, 154

LARAC-SN Lagrange relaxation based aggregated cost for specified nodes 6, 64, 95, 96, 98–103, 107, 108

LC least-cost 63, 67–70, 73–75, 82, 85, 87, 88, 90, 92, 101, 103

LD least-delay 57, 69, 70, 73–76, 78–80, 82, 85, 87, 90, 92, 101, 104

LLC last level cache 148

LLDP link-layer discovery protocol 145

M2M machine-to-machine 1

MAC medium access control 11

MCP multi-constrained path 66, 80, 82

MCSP multi-constrained shortest path 66, 67, 70, 71, 80, 82, 103, 109, 114, 115

MHM multi-hop model 44, 47, 58

MITH mole in the hole 6, 64, 95, 96, 99–103, 107, 108

MPLS multi-protocol label switching 12, 147

MTBF mean time between failures 181

NBI northbound interface 2, 9, 41, 145

NFV network function virtualization 94, 95, 102

NIC network interface card 7, 147, 148, 152, 181

NTP network time protocol 150

OF OpenFlow 2, 45, 61, 118, 121, 124, 128–130, 145, 147, 160, 161, 164, 165, 167, 180

OG optimality gap 67, 69, 73–79, 95, 101–103

ORB one ring bottleneck 72, 76

OS operating system 12, 14, 15, 73, 120, 124, 158, 180

OSNE ordered specified nodes extension 94–99, 101

OSP optimal substructure property 6, 64, 103, 105, 106, 108–111, 113, 115

OVS open vswitch 130

PDQ preemptive distributive quick 12, 13

PLC programmable logic controller 57, 72

PQ priority queuing 12, 38, 45, 48, 49, 118, 120, 124–126, 133, 134, 136

PTP precision time protocol 150, 154, 155

QoS quality of service 1, 2, 4, 7, 10–15, 40–42, 55, 59, 61, 103, 105, 115, 118, 158, 185, 187, 188

RAM random access memory 147, 151, 160

REST representational state transfer 145

RFC request for comments 172

RSVP resource reservation protocol 13

s*k*SP static k shortest paths 68, 70, 71, 77

s*k*SPT static k shortest paths tree 70, 77

SBI southbound interface 145, 146

SCADA supervisory control and data acquisition 150

SDN software-defined networking 2, 3, 6, 7, 9–11, 41, 45, 61, 95, 117, 118, 120, 136, 137, 180, 185

SFC service function chain 6, 7, 64, 79, 94, 102, 107, 108, 115

SFD start frame delimiter 121, 164, 165, 170

SJF shortest job first 13

SLA service level agreement 9, 105, 187

SNC stochastic network calculus 9, 15

SNE specified nodes extension 94, 96

SoA state of the art 6, 7, 12, 40, 54, 61, 63, 64, 69, 95, 96, 99–106, 108–111, 113–115, 117–121, 126, 129, 132, 133, 136, 138, 140, 149–151, 155, 157–159, 161, 173, 175, 181, 185–187

SP shortest path 64, 66–71, 74–77, 79–87, 89–93, 96–98, 103, 105, 106, 109, 111, 112, 114, 115, 146, 152

SP-SN shortest path with specified nodes 97–99

SPT shortest path tree 67, 68, 70, 71, 75–77, 79–82, 84–87, 90–93

SSH secure shell 145

TBM threshold-based model 44, 47–50, 52–56, 58, 61, 64, 104, 106, 107, 115, 142, 157, 158

TCAM ternary content-addressable memory 9, 124, 128, 137

TCP transmission control protocol 132, 161, 181

TDMA time-division multiple access 12, 14, 59, 60

TIVC temporally interleaved virtual cluster 12, 13

ToS type of service 121

TRB two rings bottleneck 72, 76

TRR two rings random 72, 76

TSC timestamp counter 149

TSN time-sensitive networking 15

USART universal synchronous/asynchronous receiver/transmitter 160

USB universal serial bus 160, 161

VBR variable bit rate 34, 35

VLAN virtual local area network 12, 15, 45, 55, 136, 137, 147, 148, 155

VM virtual machine 11, 13, 138–140, 142, 144, 145, 147–149, 152, 153, 155, 185

VMDQ virtual machine device queues 148

VNF virtual network function 7, 64, 79, 94, 95, 108

vNIC virtual network interface card 13

WC work-conserving 12

WDetServ wireless deterministic services 4, 10, 59–61

WFQ weighted fair queuing 12, 13

WG wireless gateway 59, 60

WSN wireless sensor network 59