PADERBORN UNIVERSITY
The University for the Information Society

Stefan Schneider

Network and Service Coordination: Conventional and Machine Learning Approaches

Dissertation

in partial fulfillment of the requirements for the degree of

Doctor rerum naturalium (Dr. rer. nat.)

submitted to

Paderborn University, Germany

Faculty of Electrical Engineering, Computer Science, and Mathematics

Supervisor

Prof. Dr. Holger Karl

Paderborn, November 2021

Referees:

Prof. Dr. Holger Karl, Hasso Plattner Institute, University of Potsdam, Germany
Prof. Dr. Martina Zitterbart, Karlsruhe Institute of Technology, Germany
Jun.-Prof. Dr. Sebastian Peitz, Paderborn University, Germany

Submission: November 2021
Examination: January 2022

Abstract

Modern services typically consist of interconnected components, e.g., microservices in a service mesh, Virtual Network Functions (VNFs) in a network service, or machine learning functions in a pipeline. Examples for service components are video optimizers or web servers for video streaming or management functions for telecommunication. As these service components are implemented in software, they can be started, stopped, and scaled on demand on available compute nodes in the network. This increased flexibility allows reducing costs while improving service quality. At the same time, it comes with new challenges, requiring dynamic service scaling and placement as well as flow scheduling and routing. Such dynamic network and service coordination is a key challenge in Network Function Virtualization (NFV), edge and cloud computing, and 5G and beyond.

This PhD thesis motivates and precisely states the problem of network and service coordination and discusses relevant real-world use cases. Such coordination is challenging due to the many influencing factors, the frequent changes (e.g., in demand), the interdependencies between different coordination decisions, and the trade-offs between multiple optimization objectives. The main contributions of this PhD thesis are therefore several coordination approaches, which solve the problem by coordinating networks and services autonomously. Specifically, Part I presents three conventional approaches for network and service coordination. Such conventional approaches are ideal for well-understood scenarios as they build on expert knowledge and follow hand-crafted algorithms or solve predefined Mixed-Integer Linear Programs (MILPs). Part II proposes four approaches based on machine learning, either supporting existing conventional coordination approaches or replacing them entirely with self-learning using Deep Reinforcement Learning (DRL). These machine learning approaches rely less on a priori or expert knowledge and instead leverage available data to *learn* network and service coordination, self-adapting to unknown and changing scenarios. The proposed approaches in Parts I and II optimize multiple, even opposing objectives and use varying optimization techniques. They range from centralized to distributed architectures and focus on different aspects of network and service coordination, complementing each other. Overall, the proposed approaches enable automated and highly optimized network and service coordination for higher service quality and efficiency, leading to better user experience, lower costs, and reduced resource consumption.

Zusammenfassung

Moderne Dienste bestehen in der Regel aus miteinander verbundenen Komponenten, z.B. aus untereinander verknüpften Microservices oder Virtual Network Functions (VNFs). Beispiele für solche Komponenten sind Video-Optimierer oder Webserver für Videostreaming, Managementfunktionen für Telekommunikation oder Funktionen für maschinelles Lernen. Da diese Komponenten üblicherweise in Software implementiert sind, können sie je nach Bedarf auf verfügbaren Rechenknoten im Netzwerk gestartet, gestoppt und skaliert werden. Durch diese erhöhte Flexibilität können die Kosten gesenkt und trotzdem die Dienstqualität verbessert werden. Gleichzeitig birgt diese Flexibilität neue Herausforderungen und erfordert eine dynamische Skalierung und Platzierung von Diensten im Netzwerk sowie eine Zuweisung von Anfragen in Echtzeit. Eine solche dynamische Netz- und Dienstkoordination ist eine zentrale Herausforderung in Network Function Virtualization (NFV), Edge- und Cloud-Computing, 5G und auch in zukünftigen Anwendungen (z.B. 6G).

In dieser Dissertation wird das Problem der Netz- und Dienstkoordination motiviert und präzise beschrieben sowie relevante Anwendungsfälle aus der Praxis diskutiert. Das Koordinationsproblem ist komplex und herausfordernd aufgrund der vielen Einflussfaktoren, der häufigen Änderungen (z.B. der Nachfrage), der Abhängigkeiten zwischen verschiedenen Koordinierungsentscheidungen und der Abwägung zwischen mehreren Optimierungszielen. Als Hauptbeitrag präsentiert diese Dissertation daher verschiedene Koordinationsansätze, die das geschilderte Problem durch die autonome Koordination von Netzen und Diensten lösen. Im Einzelnen stellt Teil I drei konventionelle Ansätze für die Netz- und Dienstkoordination vor. Solche konventionellen Ansätze sind ideal für bekannte und genau analysierte Szenarien, da sie auf Expertenwissen aufbauen und manuell definierten Algorithmen folgen oder vordefinierte Mixed-Integer Linear Programs (MILPs) lösen. Teil II präsentiert vier weitere Koordinationsansätze, die auf maschinellem Lernen basieren. Diese Ansätze unterstützen entweder bestehende konventionelle Koordinationsansätze oder ersetzen diese vollständig durch selbstständiges Lernen auf Basis von Deep Reinforcement Learning (DRL). Dank maschinellen Lernens stützen sich die Ansätze weniger auf detailliertes Expertenwissen, sondern nutzen stattdessen verfügbare Daten, um Netz- und Dienstkoordination zu *lernen* und sich selbst an unbekannte und ständig wechselnde Szenarien anzupassen. Die in den Teilen I und II vorgeschlagenen Ansätze optimieren mehrere, sogar gegensätzliche Ziele und verwenden unterschiedliche Optimierungstechniken. Sie reichen von zentralisierten bis zu verteilten Architekturen, konzentrieren sich auf verschiedene Aspekte der Netz- und Dienstkoordination und ergänzen sich gegenseitig. Insgesamt ermöglichen die vorgeschlagenen Ansätze eine automatisierte und hochgradig optimierte Netz- und Dienstkoordination. Dies führt zu höherer Dienstqualität und Effizienz und damit zu besseren Nutzererfahrungen, niedrigeren Kosten und geringerem Ressourcenverbrauch.

Contents

1. Introduction

Modern services often consist of multiple chained components, each with its own functionality. Such services are relevant in several practical contexts. For example, network services in Network Function Virtualization (NFV) consist of chained Virtual Network Functions (VNFs), e.g., firewalls or Intrusion Detection Systems (IDSs) [103, 102]. In cloud or edge computing, multiple microservices (e.g., web servers or video optimizers) may form a service mesh [150, 114]. Components for data cleaning, augmentation, processing, or training could be parts of a machine learning pipeline [120]. Nowadays, such components are typically implemented in software and can run on standard industry servers, which are available at various locations in the network.

Such softwarization has numerous benefits compared to deployment of physical hardware components. For example, in NFV, software-based VNFs replace traditional hardware middleboxes, which allows easier and faster development, deployment, and maintenance as well as reduced time-to-market and personnel costs [103, 178, 268]. In contrast to static and expensive middleboxes running on specialized hardware, softwarized components can be deployed flexibly on demand, running on standard servers. This is not only cheaper, reducing Capital Expenditure (CapEx) and Operational Expenditure (OpEx), but also enables completely new services. For example, delay-sensitive services (e.g., for vehicular networking [245], cloud gaming [232, 278], smart manufacturing [128, 151], or 5G and beyond [6, 245]) can be realized by flexibly deploying service components at the edge, i.e., close to user locations, minimizing path delay and ensuring good Quality of Service (QoS) [192, 51]. To support delay-sensitive services with hardware middleboxes, they would need to be dimensioned to handle possible peak loads and installed at all possible edge locations. Since user locations and demand change over time, this would be a significant waste of resources and far too expensive. Hence, supporting such highly responsive, delay-sensitive services is only realistically possible with modern services consisting of softwarized components.

These potential benefits and opportunities of modern, softwarized services in NFV but also in other use cases come with new challenges regarding network and service coordination [103, 178, 275, 159]. To provide such services to users (or machines), all components of a service need to be instantiated in the network at compute nodes with sufficient resources. Compute nodes are distributed geographically and topologically in the network and may represent, for example, small edge servers at a cell tower or large cloud data centers in the network core. Generally, compute resources are limited at these nodes and the available data rate on links between nodes is also restricted. Moreover, propagation delay on these links is non-negligible. These capacities and delays have to be considered when steering traffic from users through deployed instances of all service components. In doing so, load needs to be balanced between instances on different compute nodes and routing paths. As user demand changes over time, instances may need to be started or stopped and load balancing needs to be adjusted. Deploying sufficient instances and allocating enough resources is important to satisfy user demand and ensure good service quality. In turn, running too many instances or allocating too many resources leads to wasted resources and energy and unnecessary costs. Additional factors or requirements, e.g., short delays for QoS or licensing costs for deployed instances, may lead to complex trade-offs and make coordination even more challenging.

Network and service coordination concerns all online management decisions to ensure services are available to all users with good quality and low costs. This is crucial for use cases in NFV, cloud and edge computing, distributed machine learning, and 5G and beyond [110, 255, 120, 8, 148]. In this thesis, I particularly consider four main aspects of network and service coordination: 1. Service scaling, i.e., how many instances to deploy per component (horizontal scaling) and how many resources to allocate to them (vertical scaling)? 2. Service placement, i.e., at which compute nodes to deploy (or place) these instances? 3. Flow scheduling, i.e., at which of the placed instances to process incoming flows? 4. Routing, i.e., once a flow is scheduled for processing at a destination instance, over which links to route the flow for reaching the instance? Due to interdependencies and trade-offs between these four aspects, it is important to optimize them jointly together [203, 61]. Solving them sequentially and separately would make each coordination aspect simpler due to the reduced decision space but possibly also lead to much worse solutions [131, 61]. Furthermore, such coordination decisions depend on a variety of influencing factors, which can change frequently over time. Because of this complexity and fast changes, manual coordination by humans leads to suboptimal results at best or may even be completely infeasible and break service availability (e.g., if resources are over-subscribed or delay requirements are exceeded).

To this end, I present several approaches for automated and optimized network and service coordination in this thesis. The proposed approaches automatically ensure services are scaled and placed properly and traffic is balanced among instances, compute nodes, and paths adequately. By automating network and service coordination, service scaling and placement as well as flow scheduling and routing are constantly adjusted to changing demands, avoiding manual errors and continuously optimizing desired coordination objectives. Most of my approaches support optimization of multiple, even opposing optimization objectives such as high throughput vs. short delays. While I have worked on technical aspects of network and service coordination (see my papers in Section 1.1.1), my approaches presented in this thesis focus on the conceptual and algorithmic challenges of optimizing network and service coordination.

I group and present my approaches in two parts for conventional and machine learning approaches. Conventional approaches include Mixed-Integer Linear Programs (MILPs) or heuristic algorithms, which are based on detailed models of the network and involved services, designed by human experts. In contrast, machine learning approaches automatically derive useful coordination policies from previous data or own coordination experience when interacting with the network environment. Conventional coordination approaches are better understood and constitute the large majority of current state-of-the-art approaches. Machine learning approaches are often more flexible and, with the raise of deep learning, have recently gained more recognition and often outperform conventional approaches [162, 273]. Generally, the presented approaches have different strengths and weaknesses, but all allow automated and highly optimized network and service coordination.

1.1. Contributions

I designed and developed the approaches presented in this thesis as well as other related work in the time between May 2017 and November 2021. In this time, I authored or co-authored over 25 publications in international conferences and journals (Section 1.1.1). I received two awards for these publications and was invited to speak about one of them at the 110-th meeting of the Internet Engineering Task Force (IETF)[1], indicating recognition of my work both inside and outside of academia. I have been involved in the Internet Research Task Force (IRTF) NMRG, discussing and documenting challenges of applying artificial intelligence for network management (intended as Internet draft).[2] Furthermore, I was involved in three projects and was responsible for securing a grant and leading one of them (Section 1.1.2).

The following chapters and approaches in Part I and II are directly based on some of these publications and include verbatim copies. To ease the flow of reading, these verbatim copies of my own previous work are not quoted explicitly but are clearly mentioned at the beginning of each chapter. With exception of the approach in Chapter 4, which is included mostly as background and related work, I am the main author of all presented approaches. For ease of reading, I consistently write in the singular form of first person ("I" rather than "we"). I further clarify my contributions in Section 1.2 and at the beginning of each chapter in Part I and II.

1.1.1. Publications

I list my (co-)authored publications grouped into conventional and machine learning coordination approaches, which are the focus of this thesis. Additional publications, which are not in the direct scope of this thesis, are grouped into tools and concepts supporting coordination and more practical aspects on softwarized services for a smart manufacturing use case. Within each group, the papers are listed in chronological order. The list also contains papers submitted for publication that are still under review, which is indicated accordingly for these papers. All source code belonging to these papers is published as open source and referenced inside the papers and corresponding chapters. To better distinguish my work from work by others, references to papers or code that I authored or co-authored are prefixed with "S" and listed separately in the bibliography. Other references do not have any prefix.

Conventional Coordination Approaches

[S2] Sevil Dräxler, Stefan Schneider, and Holger Karl. "Scaling and Placing Bidirectional Services with Stateful Virtual and Physical Network Functions." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2018, pp. 123–131

[S24] Stefan Schneider, Lars Dietrich Klenner, and Holger Karl. "Every Node for Itself: Fully Distributed Service Coordination." In: *IFIP/IEEE Conference on Network and Service Management (CNSM)*. IFIP/IEEE. 2020

[S18] Stefan Schneider, Mirko Jürgens, and Holger Karl. "Divide and Conquer: Hierarchical Network and Service Coordination." In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IFIP/IEEE. 2021

[1]Agenda of the Network Management Research Group (NMRG) session at the IETF 110 meeting: `https://datatracker.ietf.org/doc/agenda-110-nmrg/` (accessed June 30, 2021)

[2]Relevant NMRG meeting minutes: `https://datatracker.ietf.org/doc/minutes-interim-2021-nmrg-03-202106221400/` (accessed June 30, 2021)

Machine Learning Coordination Approaches

[S35] Stefan Schneider et al. "Machine Learning for Dynamic Resource Allocation in Network Function Virtualization." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2020, pp. 122–130

[S21] Stefan Schneider et al. "Self-Driving Network and Service Coordination Using Deep Reinforcement Learning." In: *IFIP/IEEE Conference on Network and Service Management (CNSM)*. **Best Student Paper Award** **Invited Talk at the IETF 110 Meeting**. IFIP/IEEE. 2020

[S34] Stefan Schneider, Haydar Qarawlus, and Holger Karl. "Distributed Online Service Coordination Using Deep Reinforcement Learning." In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2021

[S19] Stefan Schneider et al. "DeepCoMP: Coordinated Multipoint Using Multi-Agent Deep Reinforcement Learning." In: *IEEE Transactions on Network and Service Management (TNSM)* (2022). **Under Review** **Invited Talk at Ray Summit 2021**

[S22] Stefan Schneider et al. "Self-Learning Multi-Objective Service Coordination Using Deep Reinforcement Learning." In: *IEEE Transactions on Network and Service Management (TNSM)* 18.3 (2021), pp. 3829–3842

[S38] Stefan Werner, Stefan Schneider, and Holger Karl. "Use What You Know: Network and Service Coordination Beyond Certainty." In: *IEEE/IFIP Network Operations and Management Symposium (NOMS)*. **Under Review**. IEEE/IFIP. 2022

Tools and Concepts Supporting Coordination

[S40] Mengxuan Zhao et al. "Verification and Validation Framework for 5G Network Services and Apps." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2017, pp. 321–326

[S16] Stefan Schneider, Sevil Dräxler, and Holger Karl. "Trade-Offs in Dynamic Resource Allocation in Network Function Virtualization." In: *IEEE Global Communications Conference (GLOBECOM) Workshops*. IEEE. 2018, pp. 1–3

[S8] Manuel Peuster et al. "A Prototyping Platform to Validate and Verify Network Service Header-based Service Chains." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2018, pp. 1–5

[S33] Stefan Schneider et al. "A Fully Integrated Multi-Platform NFV SDK." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2018, pp. 1–2

[S32] Stefan Schneider, Manuel Peuster, and Holger Karl. "A Generic Emulation Framework for Reusing and Evaluating VNF Placement Algorithms." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2018, pp. 1–6

[S37] Peter Twamley et al. "5GTANGO: An Approach for Testing NFV Deployments." In: *European Conference on Networks and Communications (EuCNC)*. 2018. DOI: `10.1109/EuCNC.2018.8442844`

[S10] Manuel Peuster et al. "Introducing Automated Verification and Validation for Virtualized Network Functions and Services." In: *IEEE Communications Magazine* 57.5 (2019), pp. 96–102

[S36] Stefan Schneider et al. "Specifying and Analyzing Virtual Network Services Using Queuing Petri Nets." In: *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IFIP/IEEE. 2019, pp. 116–124

[S9] Manuel Peuster, Stefan Schneider, and Holger Karl. "The Softwarised Network Data Zoo." In: *IFIP/IEEE International Conference on Network and Service Management (CNSM)*. 🏆 **Best Poster Award**. IFIP/IEEE. 2019, pp. 1–5

[S6] Askhat Nuriddinov et al. "Reproducible Functional Tests for Multi-scale Network Services." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2019, pp. 1–6

Softwarized Services for Smart Manufacturing

[S7] Manuel Peuster et al. "Prototyping and Demonstrating 5G verticals: The Smart Manufacturing Case." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2019, pp. 236–238

[S30] Stefan Schneider et al. "Putting 5G into Production: Realizing a Smart Manufacturing Vertical Scenario." In: *European Conference on Networks and Communications (EuCNC)*. IEEE. 2019, pp. 305–309

[S3] Marcel Müller et al. "5G as Key Technology for Networked Factories: Application of Vertical-specific Network Services for Enabling Flexible Smart Manufacturing." In: *IEEE International Conference on Industrial Informatics (INDIN)*. IEEE. 2019, pp. 1495–1500

[S1] Daniel Behnke et al. "NFV-Driven Intrusion Detection for Smart Manufacturing." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2019, pp. 1–6

[S5] Marcel Müller et al. "Putting NFV into Reality: Physical Smart Manufacturing Testbed." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2019, pp. 1–6

[S31] Stefan Schneider et al. ""Producing Cloud-Native": Smart Manufacturing Use Cases on Kubernetes." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2019, pp. 1–2

[S4] Marcel Müller et al. "Cloud-Native Threat Detection and Containment for Smart Manufacturing." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2020, pp. 347–349

[S39] Anastasios Zafeiropoulos et al. "Benchmarking and Profiling 5G Verticals' Applications: An Industrial IoT Use Case." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2020, pp. 310–318

1.1.2. Projects

My work was produced and used in the scope of three projects in collaboration with academic and industry partners. From 2017 to 2020, I was involved in the European 5GPPP H2020 project 5GTANGO [3], where I worked on conventional coordination approaches (Part I) as well as tools and concepts supporting coordination. I also developed cloud-native softwarized services and designed architectures for smart manufacturing in the context of 5G and NFV.

From 2018 to 2021, I led my own research project, RealVNF [S20]. For the project, I secured funding from the German Ministry for Education and Research through the Software Campus program [237], recruited student assistants, and collaborated with researchers from Huawei Germany. The project initiated my research on machine learning coordination approaches (Part II) but also drove further work on conventional coordination approaches.

From 2020 to 2021, I was involved in a follow-up research project with Huawei Germany, focusing on self-learning coordination for 5G and beyond.

All code written for these three projects is available publicly as open source. I have been actively participating in the open-source community, particularly in the context the popular Deep Reinforcement Learning (DRL) framework Ray RLlib [153, 48], where I serve as maintainer of the open-source library and spoke at the Ray Summit 2021 with over 10 000 registered attendees[3].

1.2. Thesis Structure and Overview

Following this introduction, I present general background about NFV, cloud and edge computing, and mobile networks in Chapter 2. All these practical scenarios have structural similarities that I point out in the chapter and explain how my approaches could be applied for automated and optimized coordination in these use cases. A more detailed discussion of work related to my coordination approaches is introduced in each corresponding chapter.

Chapter 3 defines the network and service coordination problem in more detail. It also introduces a common notation for problem parameters and decision variables that is used throughout the thesis. To address the network and service coordination problem, Part I presents conventional coordination approaches and Part II machine learning coordination approaches. Both parts start with typical centralized approaches, which assume global knowledge and control of network and services, and end with highly scalable distributed approaches. In line with related work [30], I consider an intuitive scalability definition where an approach is scalable if it performs gracefully with increasing input size, e.g., in terms of load or network size. I intentionally keep the introduction and problem sections within these chapters brief and specific to each chapter rather than including and frequently repeating the general introduction, background, and problem statement, which are detailed in Chapters 1–3. Finally, Chapter 11 discusses potential future work and concludes this thesis.

The following outline provides a brief overview of the approaches presented in Parts I and II. It also indicates the corresponding publications and the contributions by others, e.g., by students who I advised for their bachelor's or master's theses. The contributions for each approach are detailed again at the beginning of the corresponding chapters.

Part I: Conventional Coordination Approaches

Chapter 4: Centralized Coordination

Chapter 4 presents the Bidirectional Scaling and Placement Coordination Approach (BSP). BSP is an example for a typical centralized, conventional approach, which is still a very common (if not the most common) kind of coordination approaches in the current state of the art. Leveraging central knowledge and control of the network and services, BSP finds highly optimized coordination solutions.

[3]My talk at Ray Summit 2021: `https://www.anyscale.com/ray-summit-2021/agenda/deepcomp-multi-agent-reinforcement-learning-for-multi-cell-selection-in-5g` (accessed June 30, 2021)

Unlike the following chapters, which are new contributions produced during my PhD, this chapter is mostly previous, related work based on my master's thesis [S15]. During my PhD, I extended the BSP approach and evaluation and published it together with my colleague Sevil Dräxler [S2]. Still, the main idea, model, and approach are based on my master's thesis and are thus *not* a contribution of this PhD thesis. I nevertheless present the BSP heuristic algorithm at a high level in Chapter 4 because it was the starting point of all following research during my PhD and represents an example of typical state-of-the-art coordination approaches. As I use BSP as a comparison case for several of the following approaches, its description in a separate chapter seemed appropriate in addition to the usual discussion of related work within each of the following chapters.

Chapter 5: Hierarchical Coordination

The centralized BSP approach finds good coordination solutions but requires central knowledge and control of network and services. In practical, large networks, detailed, up-to-date information may not be available centrally. Furthermore, central coordination may be too inefficient to make frequent coordination decisions at a large network scale. Instead, this chapter proposes a novel hierarchical coordination approach that splits the network into separate, hierarchical domains and works in two phases. It aggregates and hides details from lower hierarchical levels such that higher-level coordinators can efficiently make coarse-grained decisions that are then refined by lower-level coordinators on each domain (by solving an MILP). This approach reaches solution quality similar to an equivalent centralized approach but makes faster coordination decisions based on aggregated information.

This chapter is based on my paper [S18]. The paper relies on concepts, code, and evaluation results created by Mirko Jürgens for his master's thesis [124]. I proposed the initial topic and idea and advised his thesis, providing feedback in weekly discussions to structure and develop ideas for approaching the problem. Based on the convincing outcome of the thesis, I also compiled the results and wrote the published paper.

Chapter 6: Fully Distributed Coordination

Compared to Chapter 5, Chapter 6 goes even further and proposes two fully distributed coordination approaches. The presented heuristic algorithms are fast and simple and are executed on each network node in a distributed fashion. Each node only has knowledge of its neighbors and only controls coordination locally. This architecture for fully distributed, parallelized coordination makes the approaches highly scalable and suitable for large networks. Evaluation results show that the approaches even reach comparable coordination quality as a centralized approach but are much faster.

This chapter is based on my paper [S24]. This paper relies on concepts, code, and evaluation results created by Lars Klenner for his bachelor's thesis [141]. Again, I proposed the initial topic and idea and advised his thesis through weekly discussions and written feedback to help structure and develop possible ideas. I also compiled the results, performed additional evaluation experiments, and wrote the published paper.

Part II: Machine Learning Coordination Approaches

Chapter 7: Machine Learning for Dynamic Resource Allocation

Most conventional coordination approaches such as the ones in Part I assume a given model for the resource requirements of service components. These resource requirements are typically modeled as a (often linear) function of the load a component instance has to handle. In practice, the exact function is likely unknown and may even be non-linear. To this end, this chapter proposes to train machine learning models on available real-world benchmarking data to learn the true (often non-linear) relationship between required resources and load. The trained machine learning models can then be integrated into existing conventional coordination approaches for more precise dynamic resource allocation, leading to good service quality with fewer wasted resources.

This chapter is based on my paper [S35], for which I developed the ideas and approaches, wrote code, and produced all experiment results. Some of the work done by Narayanan Puthenpurayil Satheeschandran for his master's thesis [222] has contributed to the paper as well. I designed the topic and work plan of his thesis and advised it closely, providing the main ideas and strongly guiding the development of solution approaches as well as their evaluation.

Chapter 8: Self-Learning Coordination

This chapter proposes a self-learning network and service coordination approach using DRL. Unlike the conventional approaches in previous chapters, which follow predefined rules or algorithms developed by human experts, this model-free DRL approach learns coordination directly from its own experience through feedback when interacting with the network environment. It only has very few built-in assumptions and uses realistically available partial and delayed information about the network state. It self-adapts to various scenarios with different traffic patterns, network topologies, or optimization objectives without human intervention. The proposed approach uses a centralized DRL agent controlling and periodically updating distributed coordination rules that are deployed and applied locally at each network node.

This chapter is based on my papers [S21, S22], which rely on outcomes of the RealVNF project [S20]. In the project, student assistants helped with developing prototypes and collecting evaluation results. I led the RealVNF project over two years, proposed the initial ideas and strongly guided and contributed to their development, advised the student assistants, discussed my ideas with the project partners, and wrote the papers.

Chapter 9: Self-Learning Distributed Coordination

This chapter combines ideas from Chapters 6 and 8, proposing DRL for self-learning fully distributed network and service coordination. After centralized training, separate DRL agents can be deployed at each node in the network with only local observations and control. The fully distributed architecture allows fast, local coordination decisions and more fine-grained control than the centralized DRL approach in Chapter 8. Still, the self-learning coordination using DRL allows flexible adaptation to varying scenarios without human intervention, unlike the heuristic algorithms in Chapter 6.

This chapter is based on my paper [S34]. The paper relies on concepts, code, and evaluation results created by Haydar Qarawlus for his master's thesis [207] and later extended in the RealVNF project [S20]. I proposed the initial topic and idea of his thesis, advised it through weekly discussions and feedback to develop and improve the ideas, led the RealVNF project, and wrote the paper.

Chapter 10: Self-Learning Coordination for Mobile Networks

While Chapters 4–9 focus on network and service coordination in wired networks, this chapter focuses on coordination in wireless mobile networks. I propose three different DRL variants, controlling which users are connected to which cells, allowing multiple connections for Coordinated Multipoint (CoMP) in 5G and beyond. Again, a challenge is to ensure good connection quality with limited (radio) resources and adapting to different scenarios and system configurations with only partial observations. The three proposed DRL variants differ in how training and inference is performed: The first variant uses centralized training and inference. The second trains multiple DRL agents centrally but performs inference in a distributed fashion. The third variant uses multi-agent DRL for both distributed training and inference.

This chapter is based on my paper [S19]. I developed the idea and concepts, discussed them with project partners, implemented and evaluated prototypes, and wrote the paper.

2. Background

My proposed coordination approaches in Parts I and II are useful for a variety of practical use cases and scenarios. This chapter introduces relevant background on use cases in network softwarization (Section 2.1), cloud and edge computing (Section 2.2), and mobile networks (Section 2.3). In each section, I provide a general, high-level overview of relevant concepts and how my coordination approaches can be applied in the corresponding use case. Finally, I highlight structural similarities and common challenges of these and potential future use cases. On the basis of these similarities, I define the network and service coordination problem in Chapter 3 and then present my approaches for solving it in Parts I and II. Related work specific to each coordination approach is detailed in the corresponding chapters.

2.1. Network Softwarization

Network softwarization mainly builds on two independent but highly complementary concepts: Network Function Virtualization (NFV) and Software-Defined Networking (SDN). As the name suggests, the general idea of NFV is to virtualize network functions that were traditionally deployed as hardware middleboxes in the network (e.g., firewalls, Intrusion Detection Systems (IDSs), or Deep Packet Inspections (DPIs)) [70, 103]. The resulting Virtual Network Functions (VNFs) are software components that can be executed on commodity servers, typically running on virtualized resources as illustrated in Figure 2.1. Since VNFs are software, they can be scaled and deployed flexibly on compute nodes across the network, i.e., starting one or multiple instances of a VNF according to current demand. Multiple VNFs can be chained together to form network services (also called Service Function Chainings (SFCs) [102]), e.g., for telecommunication or video streaming. Overall, this virtualization enables cheaper, faster, easier, and more flexible development, deployment, and maintenance of network services [70, 103]. The concept of NFV was originally introduced in a white paper by European Telecommunications Standards Institute (ETSI) [70] but has also been adopted and extended by Internet Engineering Task Force (IETF) [26]. In recent years, NFV has attracted significant amounts of research [103, 178, 152, 110, 268].

Complementary to NFV, SDN facilitates network softwarization and programmability by splitting the control and data plan of network switches into a (logically) centralized, programmable control plane and a simple, distributed data plane [143]. A core idea of SDN is the open, vendor-independent interface of the centralized SDN controller. It enables applications on the management plane to program the SDN controller via its north-bound Application Programming Interface (API) (Figure 2.2). Compared to traditional, individually controlled switches, the global view and control of the SDN control plane allows easier, more flexible and powerful network control, e.g., for improved traffic engineering. The SDN management and control plane can be implemented in software and run on commodity servers. At the same time, SDN switches are simplified to plain data forwarding elements, instructed by the SDN controller via its south-bound API (typically using the OpenFlow protocol [170]). The abstraction and flexibility provided by SDN has considerably affected research on traffic engineering and network coordination [143, 260, 136].

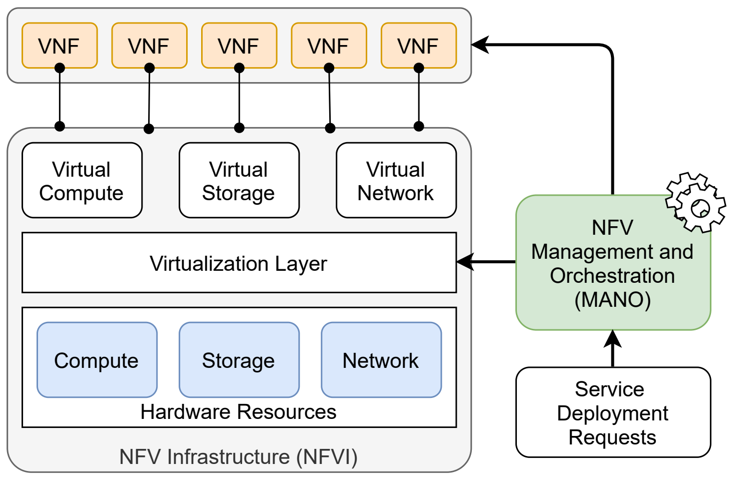


Figure 2.1.: Virtualization layers in NFV (figure adapted from [152]; © 2015 IEEE).

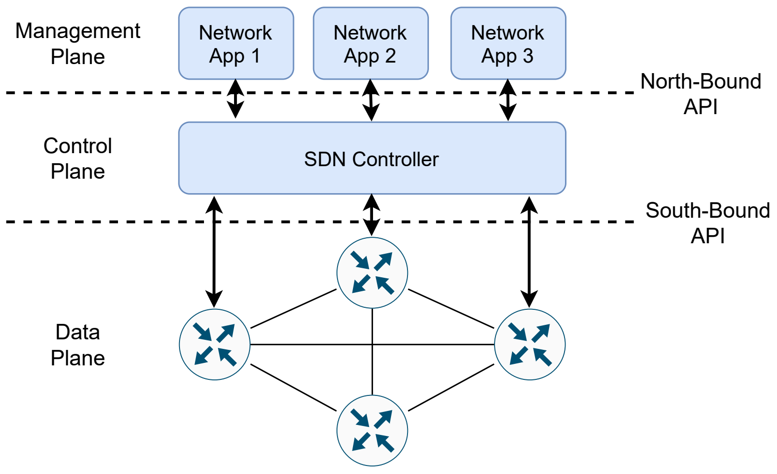


Figure 2.2.: Management, control, and data plane in SDN (figure adapted from [231]; © 2019 Wiley).

In combination, NFV and SDN enable network softwarization with more dynamic and programmable traffic processing (NFV) and steering (SDN) within the network (Figure 2.3). This allows powerful network optimization and flexible adaptation to fluctuating demand but also comes with additional challenges. Available compute resources should be utilized but not over-subscribed, link capacities and delays need to be considered, distributed and changing user demand as well as additional requirements (e.g., for Quality of Service (QoS)) must be satisfied. To ensure good service quality without wasting resources, network and services need to be coordinated properly.

Regarding coordination in NFV, VNFs are typically instantiated, scaled, monitored, and terminated on demand by Management and Orchestration (MANO) systems like Open Source MANO (OSM) [74], Open Network Automation Platform (ONAP) [156], or 5GTANGO [3]. Service

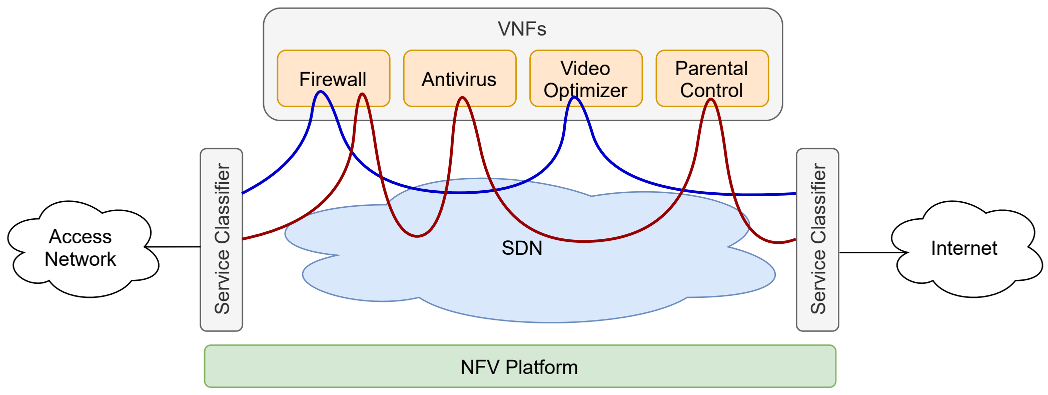


Figure 2.3.: Network softwarization with VNFs for traffic processing and SDN for traffic steering (figure adapted from [165]).

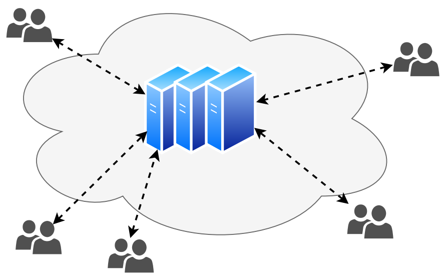
providers pass descriptors of their network services and involved VNFs to the MANO system when onboarding new services [86] and then request deployment through the MANO system (Figure 2.1). If not done manually, coordination approaches inside the MANO system may decide how to scale each involved VNF and choose suitable compute nodes for deployment (called placement). Here, my proposed coordination approaches (Parts I and II) come into play and can be leveraged inside NFV MANO systems for improved and automated scaling and placement of network services. Similarly, control of flow scheduling and routing paths is programmable inside the SDN controller (e.g., OpenDaylight [157] or Ryu [218]). Again, my proposed coordination approaches could be executed on the SDN controller via the SDN management plane for optimized scheduling and routing of flows between users and instantiated VNFs. NFV MANO systems often provide interfaces to control not only VNF scaling and placement but also traffic steering in SDN networks (e.g., through OSM's Wide Area Network (WAN) Infrastructure Managers [73]). Executing my coordination approaches on such MANO systems would allow them to jointly control and optimize both service scaling and placement as well as flow scheduling and routing.

2.2. Cloud and Edge Computing

The rise of cloud computing has preceded and facilitated the advance of network softwarization [233]. The main idea of cloud computing is to run compute jobs on pooled and virtualized compute resources in the network rather than on local and physical on-premise devices [172, 158]. The traditional approach of on-premise computing requires considerable Capital Expenditure (CapEx) to purchase compute hardware and ongoing overhead and Operational Expenditure (OpEx) for configuration, maintenance, and energy. Furthermore, dimensioning on-premise hardware is difficult. Since the demand in compute resources typically fluctuates over time, there are likely either too many or too few compute resources available at a time. If the purchased compute hardware is underutilized, it signifies waste of unused resources and thus unnecessary costs. If the on-premise compute hardware is overutilized, compute jobs take longer to complete and may even fail completely if there are too many jobs or they are particularly compute-intensive.

In contrast to on-premise computing, cloud data centers typically have vast amounts of compute resources that are virtualized and shared by multiple users, e.g., using virtual machines, Docker [174], or Linux containers [34]. In common public clouds, e.g., Amazon Web Services (AWS) [13], Google Cloud Platform (GCP) [96], and Microsoft Azure [176], these centralized and virtualized cloud resources are available publicly to any customer as opposed to private clouds with access limited to individual organizations. These large cloud data centers profit from the economy of scale and are often located at remote locations with cheap cooling and energy (e.g., on Iceland) [117]. By multiplexing compute demands from multiple users and offering cheap preemptive compute units (in addition to more expensive non-preemptive units), cloud providers can efficiently utilize their compute resources. For customers, cloud computing is often simpler, cheaper, more flexible, and offers better availability than on-premise computing [18]. With the typical "pay-as-you-go" pricing model, there is no CapEx, and OpEx directly corresponds to the actual compute demand. Thus, there are almost no costs during low demand. For compute-intensive jobs, cloud computing allows flexible scaling to many resources to accelerate job completion. In addition to compute resources, cloud data centers also offer all kinds of other resources for processing, short-term memory, or long-term storage [13, 96, 176].

While large, cost-efficient cloud compute centers at the network core or at remote locations have their benefits, there are also disadvantages. Besides potential privacy issues with data being stored outside an organization's premise [262], transmitting all data between users and remote cloud locations may lead to prohibitive end-to-end delays. Even with high-capacity fiber connections at the data center, mere propagation delay along long distances and multiple hops easily adds up to over 100 ms end-to-end delay, which is too much for delay-sensitive applications like online gaming, Augmented Reality (AR)/Virtual Reality (VR), smart manufacturing, or vehicular networks [264]. To this end, edge computing proposes the use of small compute nodes at the network edge, i.e., close to users [234] (Figure 2.4). For example, edge compute nodes could be servers that are co-located with mobile base stations to allow ultra low delays of under 1 ms (cf. Multi-Access Edge Computing (MEC)) [5]. Following the cloud paradigm, edge resources may also be virtualized and shared by multiple users. In contrast to remote cloud data centers, edge compute nodes enable shorter delays but only have very limited compute resources and are often more expensive since they benefit less from the economy of scale.


(a) Cloud computing

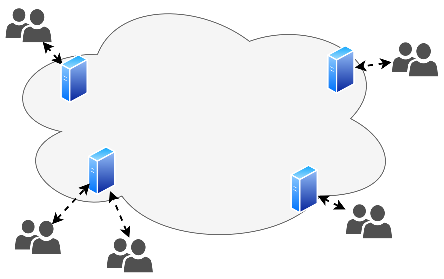

(b) Edge computing

Figure 2.4.: Illustration of cloud vs. edge computing.

In cloud and edge computing, compute jobs or services are often built of interconnected microservices forming a service mesh [150]. Orchestrators like Kubernetes [45], KubeEdge [43], or Rancher [213] automate resource allocation and coordination of such microservices within and across cloud and edge compute nodes. Again, my coordination approaches could interface or be applied inside such orchestrators (e.g., as custom Kubernetes scheduler [44]) to improve coordination of cloud/edge services and optimize the balance between short delays and low cost. My approaches do not strictly distinguish between tiers like cloud and edge but model network and compute resources in a generic way such that they can be applied to any scenario.

2.3. Mobile Networks

Network softwarization and cloud and edge computing are also important for mobile networks, particularly for 5G and beyond. 5G builds on flexible softwarized networks and virtualized compute resources to scale and deploy required network services for telecommunication and other vertical industries (e.g., health or manufacturing) according to fluctuating demand [271, 10]. In addition, modern mobile networks need to coordinate radio resources and cell selection across multiple users and cells. As users move around and connect to different cells, they compete with other connected users for limited radio resources, e.g., in the form of Physical Resource Blocks (PRBs) in Long-Term Evolution (LTE) networks. Depending on the required data rate and the channel quality, users may need considerable radio resources to ensure good Quality of Experience (QoE). This problem is particularly exacerbated for users at the edge between multiple cells, where received signals are strongly attenuated, reducing the received data rate.

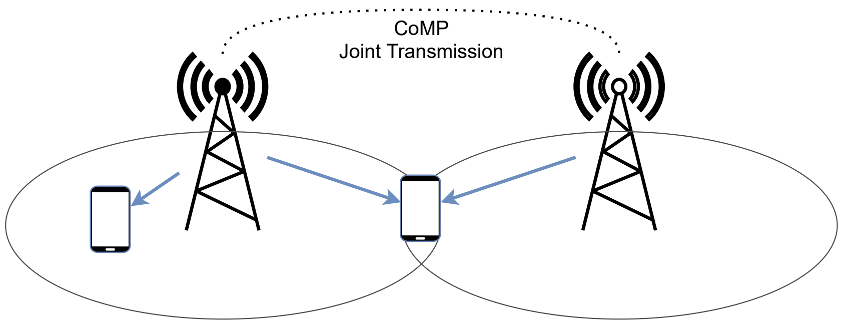


Figure 2.5.: Joint transmission from multiple serving cells using CoMP (figure adapted from [212]).

Coordinated Multipoint (CoMP) is one approach to handle increasing demand in mobile data rate, particularly at the cell edge. CoMP leverages macrodiversity by enabling joint transmissions from multiple serving cells to individual users, increasing their received data rate [119] (Figure 2.5). While CoMP was introduced in LTE Release 11 for 4G [1], it will play an increasingly important role in 5G and beyond where many small, densely deployed cells partially overlap [50, 69]. A challenge in CoMP is to dynamically select how many and which cells should serve which users in order to balance competition for limited radio resources and achieve high data rates for all users. Typically, automated coordination approaches such as the ones presented in this thesis are used to continuously optimize multi-cell selection in mobile networks. These approaches could run in the network or even on user terminals for either network-initiated or user-initiated cell selection.

The presented practical use cases in Sections 2.1–2.3 differ in many aspects but also share structural similarities. In all cases, users request services in the form of either chained VNFs or microservices, compute jobs consisting of consecutive tasks, or for mobile telecommunication. Providing these services requires resources (compute, radio, or other) that are limited and distributed to different locations in the network. User demand for these services varies over time and location. Therefore, all use cases require continuous coordination and optimization of how resources are allocated to these services. Coordination approaches need to respect limited resources, adapt to changing demands, and navigate trade-offs between different objectives, e.g., good service quality vs. low costs.

My proposed approaches solve the underlying generic problem of network and service coordination, which is relevant in all discussed real-world use cases due to their structural similarities. Future paradigms will likely also present similar challenges and may therefore profit from my proposed coordination approaches. For practical application, my approaches clearly need to focus on details of the corresponding use case. While I have worked on use case-specific technical aspects [S33, S30, S31], this thesis focuses on the underlying conceptual and algorithmic challenges of optimizing network and service coordination. Most of my approaches (Chapters 4–9) focus on coordination in wired networks as described in Sections 2.1 and 2.2. Therefore, the problem description in Chapter 3 also focuses on these use cases. Nevertheless, in Chapter 10, I also present a problem definition and approaches for coordination in wireless mobile networks as described in Section 2.3.

3. Problem Statement

Following the motivation and background in Chapters 1 and 2, in this chapter, I define the network and service coordination problem in more detail. In Parts I and II following this chapter, I present my approaches for solving the problem, each focusing on varying aspects and with different strengths and weaknesses. This chapter introduces relevant assumptions regarding problem parameters (Section 3.1), decision variables (Section 3.2), and possible optimization objectives (Section 3.3). It also establishes a common notation that I use throughout the following chapters. My intention here is to precisely define the problem; it is not the goal to provide a full mathematical formalization, e.g., as Mixed-Integer Linear Program (MILP) for feeding into a solver. I present a possible formalization as MILP as part of my coordination approach in Chapter 5.

While there are structural similarities between many relevant use cases (Chapter 2), the precise problem statement varies for coordination in wired vs. wireless networks (e.g., due to differences in user mobility). Hence, I consider coordination in wired and wireless networks separately, generally focusing more on coordination in wired networks. Specifically, the problem statement in this chapter and the following approaches in Chapters 4–9 focus on network and service coordination in wired networks, e.g., in the context of network softwarization or cloud and edge computing (Sections 2.1 and 2.2). In Chapter 10, I consider coordination in wireless mobile networks as motivated in Section 2.3. The following problem statement is based on my papers [S18, S33, S21, S22, S34] and contains verbatim copies (© 2021 IEEE).

3.1. Problem Parameters

The network and service coordination problem has three main parameters: The underlying network of nodes and links (Section 3.1.1), the services to be deployed in the network (Section 3.1.2), and the incoming traffic from users or machines requesting these services (Section 3.1.3). Table 3.1 provides an overview of all problem parameters and their notation, which I describe in more detail next.

3.1.1. Network

I model the network as undirected graph $G = (V, L)$ with topologically and geographically distributed nodes V and links L. For example, Figure 3.1 shows the real-world Abilene network topology that connects nodes across 11 cities in the United States [142]. Nodes may represent anything from large data centers to small edge nodes or even nodes without any capacity. I denote the total capacity of a node $v \in V$ as $\text{cap}_v \in \mathbb{R}_{\geq 0}$ in terms of a generic (compute) resource, e.g., Central Processing Units (CPUs). This could easily be extended to a vector $\text{cap}_v \in \mathbb{R}^n_{\geq 0}$ for modeling n different types of node resources, e.g., memory, storage, Graphics Processing

Table 3.1.: Problem parameters

Symbol	Definition
$G = (V, L)$	Network with nodes V and links L
$V^{\text{in}}, V^{\text{eg}} \subseteq V$	Ingress and egress nodes
V_v, L_v	Neighboring nodes and adjoining links of node v
cap_v	Compute capacity of node v
cap_l	Maximum data rate of link l
d_l	Propagation delay along link l
S, C	Set of all services and of all components
$V_s^{\text{in}} \subseteq V^{\text{in}}$	Subset of ingress nodes for service s
$V_s^{\text{eg}} \subseteq V^{\text{eg}}$	Possibly empty subset of egress nodes for service s
$r_c(\lambda)$	Required resources of an instance of c receiving data rate λ
$\lambda_c^{\text{out}}(\lambda)$	Outgoing data rate of an instance of c receiving data rate λ
d_c	Processing delay of instances of c
Θ_s	QoS requirements of service s
F	Set of all flows
s_f	Service requested by flow f
$v_f^{\text{in}}, v_f^{\text{eg}}$	Ingress and egress node of f
λ_f	Data rate of f
$t_f^{\text{in}}, \delta_f$	Arrival time and duration of f

Units (GPUs), or Tensor Processing Units (TPUs). A link $l \in L$ connects two nodes v and v' bidirectionally, i.e., (v, v') and (v', v) refer to the same undirected link l. The maximum data rate $\text{cap}_l \in \mathbb{R}_{>0}$ is shared in both directions over link l. Traversing link l incurs a propagation delay $d_l \in \mathbb{R}_{>0}$, which typically depends on the distance between v and v'. I assume communication within a node to be unrestricted in terms of data rate and to have negligible delay.

I define $V^{\text{in}} \subseteq V$ as set of all ingress nodes where traffic arrives in the network. Similarly, $V^{\text{eg}} \subseteq V$ denotes the set of egress nodes where traffic departs. As I elaborate further in Section 3.1.2, different services may use different ingress and egress nodes. Depending on the service, egress nodes may be different from or identical to the ingress nodes or there may be no egress nodes at all. In general, ingress and egress nodes may be located anywhere in the network, not just at its borders. Furthermore, set $V_v \subseteq V$ contains all direct neighbors of node v, which can be reached via a hop over a single link. The corresponding set of links connected to v is denoted as $L_v = \{l = (v, v') \in L | \forall v' \in V\} \subseteq L$.

3.1.2. Services

I define services abstractly as acyclic, linear chain of components that each provide some functionality. This definition is in line with the definition by Internet Engineering Task Force (IETF) for Service Function Chaining (SFC) [102] and therefore seems most relevant in practice.

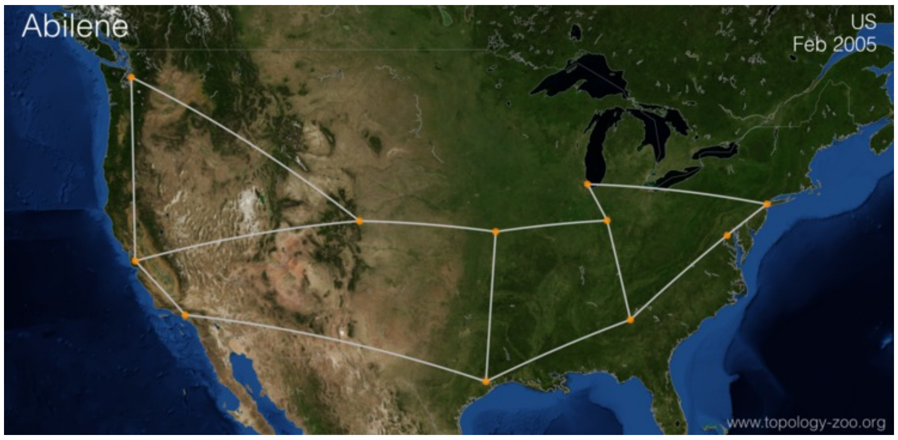


Figure 3.1.: The Abilene network topology (figure from Topology Zoo [142]).

Other, more general graph-based service structures can raise considerable questions and issues (e.g., ambiguous semantics), which I discuss in my paper [S36]. Hence, I define the components of a service $s \in S$ as ordered chain $C_s = \langle c_1, c_2, ... \rangle$ of arbitrary but fixed length. Set S contains all services and $C = \bigcup_{s \in S} C_s$ denotes the set of all components over all services.

A component c provides certain functionality and may constitute, for example, Virtual Network Functions (VNFs) in Network Function Virtualization (NFV), microservices in a service mesh, or machine learning functions in a pipeline [103, 150, 120]. Each component can be instantiated zero, one, or multiple times and deployed across different nodes in the network, depending on the current demand. All instances of c are functionally equivalent but independent of each other. In line with the current serverless trend [21], I do not explicitly consider intra-node coordination. Instead, my approaches focus on inter-node network and service coordination, e.g., deciding at which nodes to instantiate a component c. If my approaches decide to instantiate c at a node v, internally *within* v, there may be one or multiple instances of component c, depending on the demand and the resources at the node (e.g., large data center vs. single server). I assume these intra-node coordination decisions to be handled independently and transparently by the node's operating system or systems like Kubernetes [45].

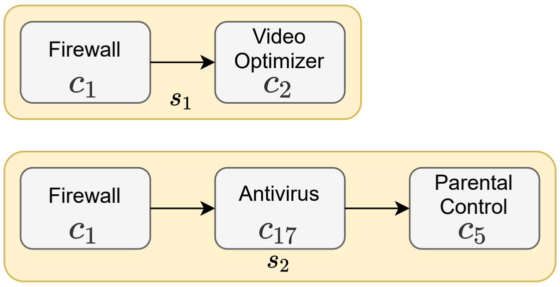


Figure 3.2.: Example services s_1 and s_2 share the same component c_1, here a firewall.

Components may be reused and shared across multiple services, e.g., $C_{s_1} = \langle c_1, c_2 \rangle$ and $C_{s_2} = \langle c_1, c_{17}, c_5 \rangle$. In the example shown in Figure 3.2, component c_1 refers to a firewall that is used in both services s_1 and s_2. Instances of c_1 may handle traffic belonging to both s_1 and s_2. If this is not desired, then using a different identifier, e.g., c_1 in s_1 and c_1' in s_2, enforces separate instances and avoids reuse and sharing across services. The same holds for components that are used multiple times within a single service. For example, in $C_{s_3} = \langle c_1, c_4, c_1, c_6 \rangle$, a single instance of component c_1 could be deployed and used twice in s_3. Using distinct identifiers c_1 and c_1' to distinguish the component at the two different positions within the service chain, i.e., $C_{s_3} = \langle c_1, c_4, c_1', c_6 \rangle$, enforces separate coordination and instances for c_1 and c_1'.

When traffic traverses an instance of component c, it incurs a processing delay $d_c \in \mathbb{R}_{\geq 0}$ during which the instance requires resources as defined by function $r_c(\lambda) \in \mathbb{R}_{\geq 0}$, e.g., to meet a predefined Service-Level Agreement (SLA). Function $r_c(\lambda)$ is monotonically increasing with the current data rate λ of incoming traffic. I discuss options for modeling $r_c(\lambda)$ based on real-world data in Chapter 7. For simplicity, I assume that processing delay d_c is a fixed or randomly distributed value independent of the current load or allocated resources. Furthermore, instances may modify traversing traffic, affecting its data rate, e.g., through compression or data augmentation [164]. I therefore define the outgoing data rate as function $\lambda_c^{\text{out}}(\lambda) \in \mathbb{R}_{>0}$ for an instance of component c with currently incoming data rate λ. Finally, services may have additional requirements for QoS, which are specified in Θ_s. For example, $\Theta_s = (d_s)$ may optionally specify a maximally acceptable end-to-end delay $d_s \in \mathbb{R}_{>0} \cup \{\varnothing\}$ for traffic traversing service s. If multiple flows traverse s, the delay bound d_s applies to all flows. Similarly, when multiple flows traverse instances of a component c in parallel, they each incur a processing delay d_c, require resources according to function $r_c(\lambda)$ for processing, and leave c with a modified data rate according to function $\lambda_c^{\text{out}}(\lambda)$. I provide further details and examples in Section 3.1.3.

Finally, each service s defines its own set of ingress nodes V_s^{in} and egress nodes V_s^{eg}, where $V^{\text{in}} = \bigcup_{s \in S} V_s^{\text{in}}$ and $V^{\text{eg}} = \bigcup_{s \in S} V_s^{\text{eg}}$. As mentioned in Section 3.1.1, both ingress and egress nodes may be arbitrary nodes in the network. This allows modeling different kinds of services and scenarios easily. For example, in telecommunications, ingress and egress may correspond to different nodes based on the location of the caller and callee, respectively, i.e., $V_s^{\text{in}} \neq V_s^{\text{eg}}$. In other scenarios, e.g., online gaming, traffic returns to its source after traversing the requested service, i.e., $V_s^{\text{in}} = V_s^{\text{eg}}$. Finally, it is also possible to model traffic that is just processed by service s but does not have a specific egress. In this case, $V_s^{\text{eg}} = \varnothing$ such that traffic traverses the requested service and ends at the last service component, e.g., for processing and collecting measurement data in Internet of Things (IoT) scenarios.

3.1.3. Traffic

Traffic arrives in the form of flows at the network's ingress nodes V^{in} over time steps $t = 1, 2, ..., T$. Flows are unsplittable and represent traffic, for example, from users, machines, or sensors using some service. A flow $f = (s_f, v_f^{\text{in}}, v_f^{\text{eg}}, \lambda_f, t_f^{\text{in}}, \delta_f) \in F$ is characterized by its requested service $s_f \in S$, its ingress node $v_f^{\text{in}} \in V_{s_f}^{\text{in}}$ and egress node $v_f^{\text{eg}} \in V_{s_f}^{\text{eg}} \cup \{\varnothing\}$, its data rate λ_f, its time of arrival t_f^{in}, and its duration δ_f. It holds that $V^{\text{in}} = \bigcup_{f \in F} v_f^{\text{in}}$ and $V^{\text{eg}} = \bigcup_{f \in F} v_f^{\text{eg}} \setminus \{\varnothing\}$. Figure 3.3 illustrates these flow attributes, which I further discuss in the following.

I model traffic on the abstraction level of flows as a continuous stream and do not consider individual packets or bits (cf. fluid approximation [38]). When flows reach an instance of a component c, I assume processing to start directly for each incoming flow, incurring processing

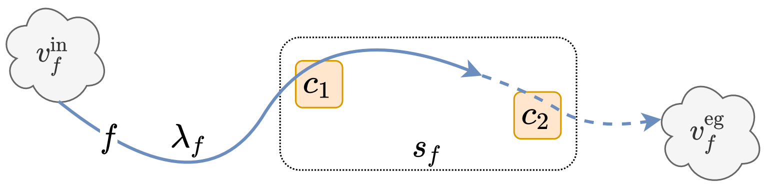


Figure 3.3.: Visualization of flow f with data rate λ_f and duration δ_f arriving at ingress v_f^{in} (at time t_f^{in}), traversing its requested service s_f, and departing at egress v_f^{eg}.

delay d_c, rather than waiting for the entire flow to arrive completely before processing. The end-to-end delay d_f of a flow f is the time from its arrival t_f^{in} to time t_f^{out} when it starts its departure, i.e., when reaching v_f^{eg} or an instance of the last component in C_{s_f} if $v_f^{\text{eg}} = \varnothing$. If the requested service s_f has QoS requirements Θ_{s_f} in form of a maximally acceptable end-to-end delay d_{s_f} (Section 3.1.2), then f must reach its egress within d_{s_f}, i.e., $d_f = t_f^{\text{out}} - t_f^{\text{in}} \leq d_{s_f}$, or it expires.

I assume a flow f to have a required (average) data rate λ_f (e.g., in byte/s), which stays fixed for the entire flow duration δ_f (e.g., in seconds) but may be modified by traversed components according to $\lambda_c^{\text{out}}(\lambda_f)$ as described in Section 3.1.2. This model requires a given data rate and duration per flow and does not allow to specify traffic only in terms of total volume, e.g., to transmit data as quickly as possible. Still, flow data rate λ_f and duration δ_f (and all other flow attributes) can be chosen arbitrarily to model a wide variety of traffic scenarios. As this model is very generic and flexible, it is useful to consider two relevant edge cases of possible traffic scenarios more closely: Few, long flows (Case I) vs. many, short flows (Case II). In the following, I discuss these two cases and their implications for network and service coordination in more detail.

Case I: Few, Long Flows

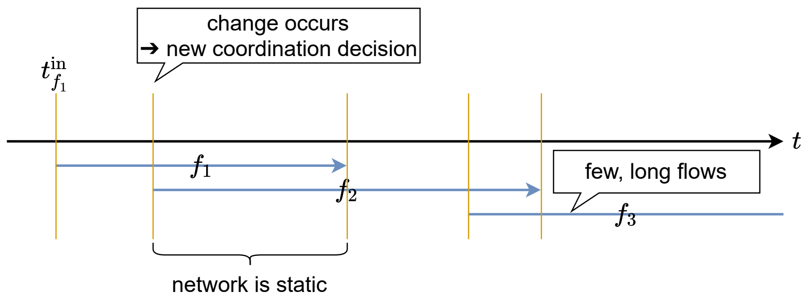


Figure 3.4.: Case I: Few, long flows with significant time intervals in between flow arrivals or departures, during which the network is static.

In Case I, I consider scenarios where traffic arrives in the form of few but long flows, e.g., representing long video streams, large backups, or aggregated traffic from multiple smaller but semantically related connections. As illustrated in Figure 3.4, the time between flow arrivals and departures is relatively long, leading to considerable time intervals in which the load is basically static, i.e., no flow arrives or departs.

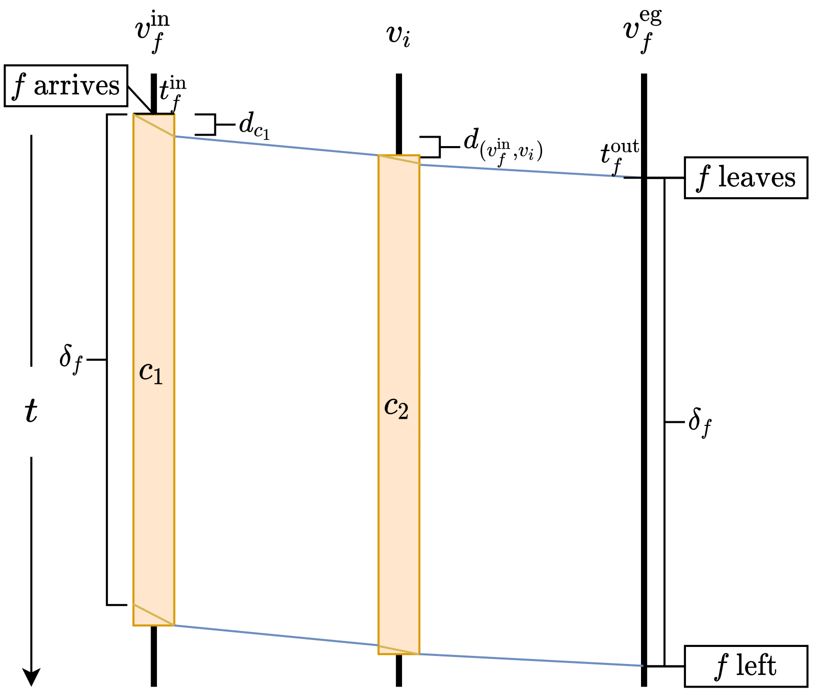
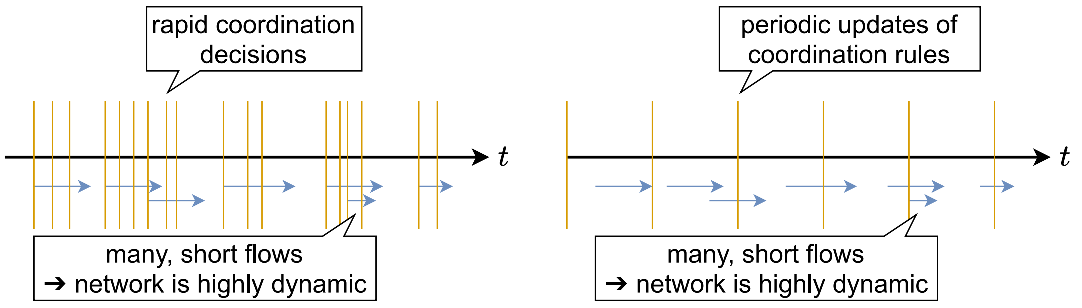


Figure 3.5.: In Case I, flow f is long and continuously utilizes instances and links along its entire path. Processing and propagation delays are virtually negligible compared to long flow duration δ_f.

In these static time intervals, traffic continuously flows from ingress to egress and utilizes resources on all links on the path as well as at instances of all requested service components. This is visualized in Figure 3.5, showing processing and propagation times of a single long flow f arriving at its ingress at time t_f^{in} and continuously traversing example service s_1 with $C_{s_1} = \langle c_1, c_2 \rangle$ from Figure 3.2. Figure 3.5 shows the total processing times of f at instances of c_1 and c_2 as yellow boxes. These total processing times depend on flow length δ_f and the components' processing delays d_{c_1} and d_{c_2}, respectively. In Case I, the short processing delays d_{c_1} and d_{c_2} are virtually negligible compared to the long flow duration δ_f. Similarly, link delay $d_{(v_f^{\text{in}},v_i)}$ for propagation from ingress v_f^{in} to c_2 at node v_i and link delay $d_{(v_i,v_f^{\text{eg}})}$ from v_i to the egress are rather insignificant compared to long flow duration δ_f in Case I. During processing (shown as yellow boxes in Figure 3.5), instances require compute resources of $r_{c_1}(\lambda_f)$ and $r_{c_2}(\lambda_f)$ as described in Section 3.1.2. Here, c_1 may change f's data rate λ_f according to function $\lambda_{c_1}^{\text{out}}(\lambda_f)$ before reaching c_2, where it may change again according to $\lambda_{c_2}^{\text{out}}(\lambda_f)$.

Due the few and long flows and the resulting long time intervals without relevant changes, coordination decisions are only required infrequently in Case I, i.e., whenever a flow arrives or departs (shown as yellow lines in Figure 3.4). Hence, in such situations, compute-intensive and time-intensive coordination approaches are viable, e.g., centralized algorithms or approaches based on MILPs.

Case II: Many, Short Flows


rapid coordination decisions
many, short flows
➔ network is highly dynamic
t
periodic updates of coordination rules
many, short flows
➔ network is highly dynamic
t

(a) Frequent coordination decisions (b) Periodic coordination decisions

Figure 3.6.: Case II: Many, short flows arrive, traverse, and depart the network rapidly, making the scenario highly dynamic. There are different options for coordination in such fast-paced, dynamic scenarios, e.g., a) rapid decisions at flow arrival and during traversal or b) coordination rules that are updated periodically.

In contrast to Case I, Case II concerns traffic scenarios with many, small flows that arrive rapidly in the network. Thus, the two cases represent opposite sides in the spectrum of possible traffic scenarios. The many, small flows in Case II may, for example, indicate a more fine-grained view of individual, small Transmission Control Protocol (TCP) connections or represent short-lived traffic, e.g., for web services. Figure 3.6 illustrates that, in this case, flows arrive and depart frequently, leading to a highly dynamic scenario. Different coordination options are conceivable in such a scenario: For example, a coordination approach could make rapid coordination decisions whenever a flow arrives and again as it traverses the network, e.g., at every hop (shown as yellow lines in Figure 3.6a). This requires fine-grained, up-to-date information about the individual flows and fast coordination decisions. Alternatively, a coordination approach could define coordination rules that are applied to all flows at runtime and are updated in periodic coordination decisions, independent of flow arrival or departure (Figure 3.6b). In addition to these two examples, other options for coordination are also conceivable. How and when coordination decisions are taken depends on the specific coordination approach.

Unlike the long flows in Case I, flow duration in Case II is short compared to propagation and processing delays, such that a single flow typically does not utilize the entire routing path and instances of all components at once. Instead, short flows traverse the network and only occupy some links and instances on their path at a time. This is visualized in Figure 3.7, showing processing and propagation times of a single short flow f traversing example service s_1 (Figure 3.2). Here, flow duration δ_f is rather short compared to processing delays d_{c_1} and d_{c_2} as well as link propagation delays $d_{(v_f^{\text{in}},v_i)}$ and $d_{(v_i,v_f^{\text{eg}})}$. Consequently, processing f at c_1 barely overlaps with processing at c_2. Similarly, f only occupies the links from v_f^{in} to v_i and from v_i to v_f^{eg} for short, only briefly overlapping time intervals. Hence, node and link resource utilization is highly volatile in Case II, making the scenario very dynamic. Consequently, making per-flow decisions in Case II (as shown in Figure 3.6a) requires frequent and fine-grained coordination with highly efficient coordination approaches, e.g., fast heuristics or distributed algorithms.

My approaches in Parts I and II are not strictly limited to either Case I or Case II but support both. Still, the approaches in Chapters 4 and 5 are comparably compute-intensive and time-

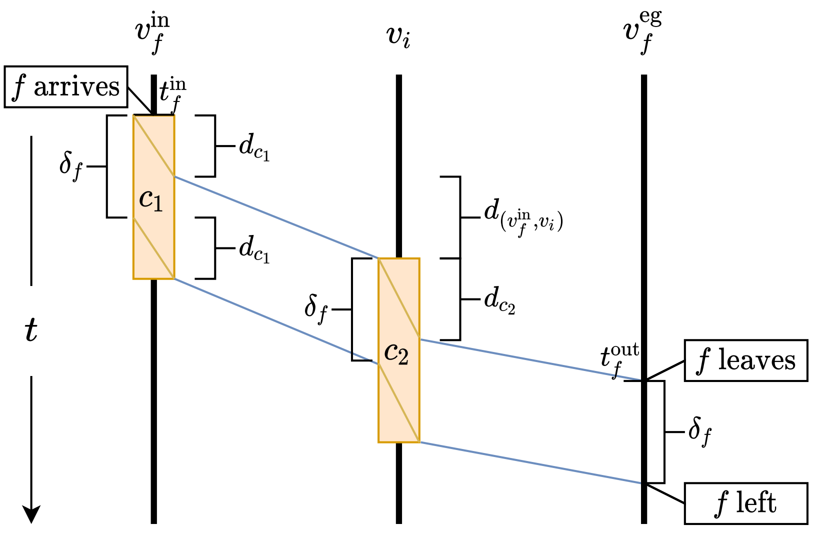


Figure 3.7.: Illustration of times and durations for arrival, processing, propagation, and departure of a short flow f in Case II. (Figure adapted from [S34]; © 2021 IEEE.)

intensive and thus more suitable when considering few and long flows (Case I). The approaches in Chapters 6–9 focus more on efficient coordination of many short flows (Case II).

3.2. Decision Variables

The purpose of network and service coordination is to ensure that services are deployed and provided to their users in good quality and without wasting resources or costs. To this end, I focus on four main aspects of network and service coordination, which I describe in the following sections: 1. Service scaling, 2. service placement (both in Section 3.2.1), 3. flow scheduling (Section 3.2.2), and 4. routing (Section 3.2.3). My coordination approaches optimize multiple (often all) of these four aspects jointly together. Such joint optimization is important due to interdependencies and trade-offs between the coordination aspects [203, 61]. Solving them sequentially and separately would make each coordination aspect simpler due to the reduced decision space but possibly also lead to much worse solutions [131, 61]. I use separate decision variables to clearly distinguish service scaling and placement, flow scheduling, and routing. Still, due to the interdependencies between these aspects, the setting of the corresponding decision variables can often be derived from each other. I explain the four coordination aspects, their interdependencies, and the corresponding decision variables in the following, using Figure 3.8 as running example. Table 3.2 provides an overview of the main decision variables and their notation.

3.2.1. Service Scaling and Placement

Service scaling determines how often to instantiate each service component and service placement controls where to deploy (or place) these instances in the network. Since I focus on

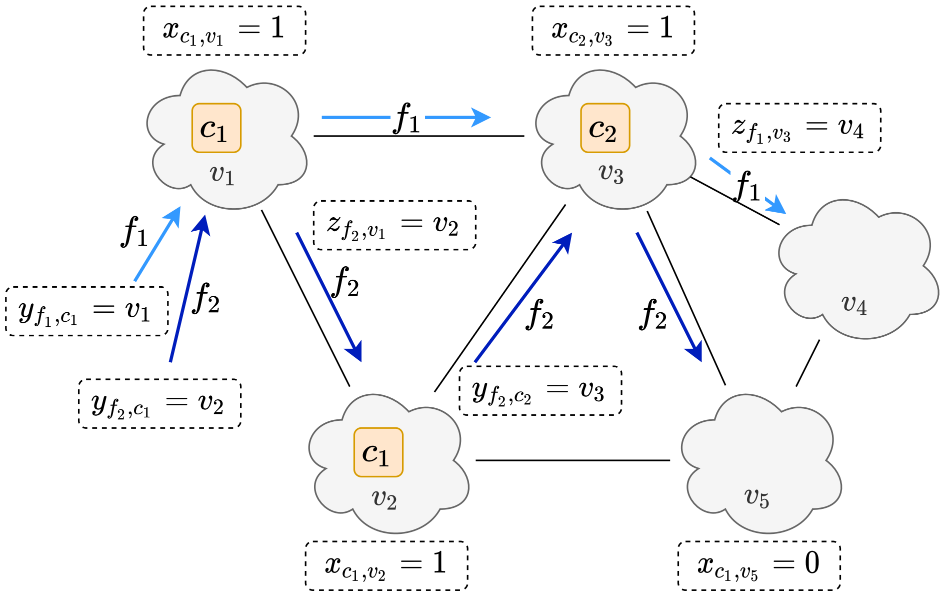


Figure 3.8.: Illustration of decision variables for service scaling and placement ($x_{c,v}(t)$), flow scheduling ($y_{f,c}(t)$), and routing ($z_{f,v}(t)$). For simplicity, the figure only shows some but not all variable settings and omits parametrization with time t. (Figure adapted from [S34]; © 2021 IEEE.)

inter-node not intra-node coordination (Section 3.1.2), service scaling and placement only concerns whether or not a component c is instantiated and placed at a node v. This scaling and placement decision can be jointly controlled via binary decision variable $x_{c,v}(t) \in \{0,1\}$, which is 1 if c is placed at v at time t and 0 otherwise. Internally within v, I assume the node's operating system or systems like Kubernetes [45] to handle the deployment of c independently and transparently if $x_{c,v}(t) = 1$. In this case, there may be one or multiple instances of c deployed within v, depending on the load as well as the type and resources of the node (e.g., large data center vs. small single-core server). In the example of Figure 3.8, flows f_1 and f_2 traverse example service s_1 (Figure 3.2) with $C_{s_1} = \langle c_1, c_2 \rangle$. At time t shown in the figure, component c_1 is scaled and placed at two nodes v_1 and v_2, and c_2 is only placed at v_3. Accordingly, $x_{c_1,v_1}(t) = x_{c_1,v_2}(t) = x_{c_2,v_3}(t) = 1$. For all other components c and nodes v, $x_{c,v}(t) = 0$.

Coordinating service scaling and placement is important since deploying all components on

Table 3.2.: Decision variables

Symbol	Domain	Definition
$x_{c,v}(t)$	$\{0,1\}$	Scaling and placement: Whether or not an instance of component c is placed at node v at time t
$y_{f,c}(t)$	$V \cup \{\varnothing\}$	Flow scheduling: At which node (if any) to process flow f requesting an instance of c at time t
$z_{f,v}(t)$	$V_v \cup \{\varnothing\}$	Routing: To which neighbor (if any) to route flow f, whose head is currently at node v at time t

all nodes permanently is typically not feasible or desirable. Unused idle instances may still consume resources, i.e., $r_c(0) > 0$. Even if idle instances only require negligible compute resources (e.g., lightweight containers, unikernels, or simple processes), they often have non-negligible memory requirements [S9]. Deployment also requires an up-to-date image of the component (e.g., a Docker or virtual machine image) to be available at the node. With many different components and versions in the network, their combined images likely require considerable storage space, which is limited for small edge nodes. Even if feasible, deploying all components at all nodes generally tends to waste resources and may also incur extensive licensing or other costs. Therefore, it is typically desirable to actively and dynamically control service scaling and placement according to the current demand. Indeed, I show that flexibly adjusting scaling and placement to the demand is crucial for good service quality and sensible resource utilization [S16].

3.2.2. Flow Scheduling

Flow scheduling concerns where to process incoming flows. Particularly, the question is at which instance of a component c to process a flow f, where $c \in C_{s_f}$ is a component of the requested service s_f. As I focus on inter-node scaling and placement of instances, I only consider instances of c that are distributed across different nodes. Hence, the scheduling decision selects the node of a suitable instance for processing. Specifically, decision variable $y_{f,c}(t) = v \in V \cup \{\varnothing\}$ indicates that f should be processed at an instance of c at node v and time t. Note that node v may be any node in the network, not just neighbors of f's previously traversed node. If f is not scheduled for processing at c at time t, $y_{f,c}(t) = \varnothing$. In the example of Figure 3.8, flow f_1 and f_2 both arrive at ingress v_1 and request service s_1 with $C_{s_1} = \langle c_1, c_2 \rangle$. Here, f_1 is scheduled for processing c_1 at v_1, and f_2 is scheduled to node v_2, i.e., $y_{f_1,c_1}(t) = v_1$ and $y_{f_2,c_1}(t) = v_2$.

Clearly, there is an interdependency between service scaling and placement and flow scheduling. Processing f at an instance of c at v is only possible if a corresponding instance is available, i.e., $y_{f,c}(t) = v$ requires $x_{c,v}(t) = 1$. Depending on $x_{c,v}(t)$, component c may not be available at node v when a flow arrives to be processed there. At this point, fetching, installing, and starting an instance of c on demand could take considerable time, in which the flow may already expire and get dropped. Hence, jointly coordinating service scaling and placement as well as flow scheduling is important to avoid expired and dropped flows.

Proper flow scheduling allows to balance load across different instances and nodes according to the current demand and available compute resources. As described in Section 3.1.2, instances of c require resources $r_c(\lambda)$ relative to their handled data rate λ. The total resource requirements $r_v(t)$ at a node v at time t add up over all deployed instances and must be within the node's capacity:

$$r_v(t) = \sum_{c \in C} x_{c,v}(t) r_c\Big(\sum_{f \in F} \mathbb{1}_{\{y_{f,c}(t)=v\}} \lambda_f\Big) \leq \mathrm{cap}_v \tag{3.1}$$

where $\mathbb{1}_{\{y_{f,c}(t)=v\}}$ is an indicator variable that is 1 if $y_{f,c}(t) = v$ and 0 otherwise. Hence, optimized flow scheduling ensures that load is distributed across nodes to avoid exceeding node capacities and resulting dropped flows. At the same time, scheduling flows to instances close by can reduce the length of routing paths and resulting end-to-end delay, thus improving QoS.

3.2.3. Routing

Finally, I consider routing as the fourth main aspect of network and service coordination. Routing a flow f requires deciding its path from its ingress v_f^{in}, through the deployed instances belonging to service s_f, and to its egress v_f^{eg} (if any). While I consider flow scheduling and routing as distinct coordination aspects, there is clearly an interdependency between the two: Flow scheduling determines at which instances to process flow f, and routing decides the path to and in between these instances.

The routing path is defined as a series of hops starting at v_f^{in}. Each hop goes from a node v to a direct neighbor $v' \in V_v$ via a single link $(v, v') \in L_v$. The corresponding decision variable $z_{f,v}(t) = v' \in V_v \cup \{\varnothing\}$ indicates that flow f should be routed from node v to neighbor v' at time t. Again, $z_{f,v}(t) = \varnothing$ indicates that f is not routed to any neighbor at time t. To ensure that routing decisions are well defined, they always refer to the head to the flow. The remainder of the flow, which may span across multiple nodes and links (particularly long flows in Case I), follows the routing path taken by the flow head. In the example of Figure 3.8, flow f_1 is processed locally at ingress v_1 at component c_1, but f_2 is scheduled to v_2 (i.e., $y_{f_2,c_1}(t) = v_2$). Consequently, f_2 is routed along the shortest path via the direct link (v_1, v_2) to neighbor v_2, i.e., $z_{f_2,v_1}(t) = v_2$. In principle, f_2 could also be routed along another path, e.g., from v_1 via v_3 to v_2. This may be helpful if link (v_1, v_2) is already fully utilized.

Routing f from v to v' occupies data rate λ_f on the corresponding link $l = (v, v') \in L_v$. Consequently, total allocated data rate $r_l(t)$ is summed up over all flows routed via l and must be within the link's capacity:

$$r_l(t) = \sum_{f \in F} \mathbb{1}_{\{z_{f,v}(t) = v'\}} \lambda_f \leq \text{cap}_l \tag{3.2}$$

where $\mathbb{1}_{\{z_{f,v}(t)=v'\}}$ is an indicator variable that is 1 if $z_{f,v}(t) = v'$ and 0 otherwise. I assume flows cannot traverse a link if they were to exceed the link's capacity. Instead, these flows would need to be routed via a different link or are dropped. Therefore, similar to how flow scheduling balances load across compute nodes, optimized routing is important to balance load across different paths to avoid exceeding link capacities and resulting dropped flows. Still, routing via shorter paths or links with short delay d_l helps reduce end-to-end delay and improve QoS.

3.3. Optimization Objectives

There are many reasonable objectives for network and service coordination, depending on the use case and the individual goals of an operator or service provider. A core objective is providing the requested services with acceptable quality to as many users (or machines) as possible. All of my coordination approaches therefore aim at successfully processing as many flows as possible without exceeding the network's limited node and link capacities.

In addition, there are often other desirable objectives, for example,

- minimization of end-to-end delay, which is important for QoS,
- minimization of the number of placed instances in order to reduce costs (e.g., for licensing),

- minimization of used compute nodes, where completely unused nodes may be turned off to save energy and costs (e.g., at the network edge).

A challenge with multiple objectives is that these objectives are often competing. For example, distributing load across nodes and links in the entire network may allow processing more flows without exceeding capacities, but it also leads to higher end-to-end delays and worse QoS. Vice versa, processing all flows close to their ingress may reduce end-to-end delay but increases load at and around the ingress, thus potentially exceeding capacities and dropping more flows.

As optimizing one objective can easily deteriorate another, my coordination approaches allow to actively optimize multiple desired objectives and to navigate the trade-off between these objectives. I provide the exact formulation of optimization objectives for each approach in the corresponding chapters.

Part I.

Conventional Coordination Approaches

4. Centralized Coordination

As discussed in Chapters 1–3, network and service coordination is crucial for many real-world use cases and requires joint scaling, placement, scheduling, and routing. The large majority of existing work proposes conventional centralized coordination approaches [110]. Examples are Mixed-Integer Linear Programs (MILPs) or heuristic algorithms designed by experts that require global knowledge and control of the network and involved services. In this chapter, I present such a conventional centralized coordination approach that I refer to as Bidirectional Scaling and Placement Coordination Approach (BSP).

BSP is mostly previous work based on my master's thesis [S15], where I developed the main idea, model, and approach of BSP. I further built on this work during my PhD together with my colleague Sevil Dräxler to publish it as conference paper [S2]. Unlike the master's thesis, the paper includes the notion of individual flows as defined in Section 3.1.3, an adjusted MILP formulation to properly distinguish these flows, a proof of NP-hardness, and an updated and extended evaluation. The MILP approach and NP proof are mostly Sevil Dräxler's contribution, whereas the heuristic algorithm is mostly my contribution. Since the main idea and approach were already developed during my master's thesis [S15] and are also presented in Sevil Dräxler's PhD thesis [59], *this chapter is mostly intended as background and related work, not as a contribution as part of my PhD*. I still provide a high-level overview of the BSP heuristic in this chapter because I repeatedly compare against it in following chapters. I omit other parts of this work, e.g., the NP-hardness proof and the MILP formulation, which are not relevant for the remainder of this thesis but can be found in the corresponding paper [S2]. This chapter contains verbatim copies of the following paper [S2]: Sevil Dräxler, Stefan Schneider, and Holger Karl. "Scaling and Placing Bidirectional Services with Stateful Virtual and Physical Network Functions." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2018, pp. 123–131, © 2018 IEEE. In this chapter and throughout my dissertation, I consistently write in the first-person singular form ("I" rather than "we") for ease of reading. The source code corresponding to this chapter is publicly available on GitHub [S12].

4.1. Introduction

In this chapter, I present BSP, which is a conventional centralized approach for solving the network and service coordination problem described in Chapter 3. More specifically, I propose two approaches for joint scaling, placement, scheduling, and routing: First, an MILP that can be solved to find optimal solutions but is, in practice, limited to small scenarios. Second, a heuristic that finds close-to-optimal solutions much faster than the MILP approach and is thus useful for larger, practical scenarios. Both approaches optimize multiple objectives and support dynamic reuse of shared instances within a single service or across different services as discussed in Section 3.1.2. They also support coordination of bidirectional services by distinguishing upstream and downstream traversal of each component to ensure correct use of stateful components in such bidirectional services. Since this thesis focuses on more common unidirectional services, I omit details about coordination of bidirectional services in this chapter and refer interested readers to the paper [S2]. Here, I focus on the BSP heuristic (Section 4.2) and its evaluation, comparing it against the MILP (Section 4.3).

4.2. BSP Heuristic Approach

BSP requires detailed global information of the network (including capacities and delays), services (including components, resource requirements, etc.), and current traffic (including information of all active flows and their attributes) as defined in Section 3.1. Given this information, BSP calculates embeddings that define how to scale and place involved service components and how to schedule and route incoming flows in the entire network. As it requires global knowledge of all flows, BSP is most suitable for scenarios with few and long flows as described in Section 3.1.3, Case I, where it is called to adjust network and service coordination whenever a flow arrives or departs. It supports dynamic online coordination, taking previous embeddings into account and adjusting them to the current situation. In general, BSP tries to accept and serve all incoming flows while optimizing four objectives with decreasing priority: 1. Avoid (or minimize) over-subscription of node and link capacities. 2. Minimize changes when adjusting an existing embedding in terms of added or removed instances to minimize overhead and licensing costs. 3. Minimize the total resource consumption of all deployed services. 4. Minimize the total delay of flows traversing the deployed services. By default, BSP optimizes a lexicographical combination of these four objectives with the given priorities.

Algorithm 1 provides an overview of the BSP heuristic algorithm, which I simply refer to as BSP. The algorithm consists of three parts: After an initialization phase (Section 4.2.1), the heuristic's embedding procedure (Section 4.2.2) is executed repeatedly, following a destroy-and-repair approach using tabu search [94] to iteratively improve the solution (Section 4.2.3). In the following, I describe the algorithm and the three parts in more detail.

4.2.1. Initialization

During initialization, the heuristic computes the shortest paths between all pairs of nodes in the substrate network based on the Floyd-Warshall algorithm [82] (line 1 in Algorithm 1). These precomputed paths are stored and can then be accessed in constant time throughout the heuristic algorithm. To favor paths with high capacity and low delay, each link l is given a weight $w_l = \frac{1}{\text{cap}_l + \frac{1}{d_l}}$ that is used for path computation, where I assume cap_l and d_l to be centrally known, static, and positive. If these link capacities or delays were to change, the paths would need to be recomputed and updated accordingly. While the assigned weights favor paths with high capacity, routing via precomputed paths is generally prone to exceeding link capacities if these capacities are very tight. To this end, the iterative improvement phase repeatedly tests alternative placements and paths to overcome and avoid potential over-subscription (Section 4.2.3).

In the initialization phase, BSP removes embeddings from any old services that are no longer used, i.e., removing any placed instances or allocated resources for these services (line 2). For the services that are in use, it computes an ordered set S_{sorted} containing all active services in decreasing order of estimated resource requirements (line 3). BSP estimates the total required

Algorithm 1 BSP Heuristic Algorithm

```
 1: Precompute and store all-pairs shortest paths [82]                    ▷ Initialization
 2: Remove embeddings of old, expired services $s \notin S$
 3: $S_{\text{sorted}} \leftarrow$ sort $S$ with decreasing resource requirements
 4: tabu $\leftarrow \varnothing$
 5: Init best, incumb, and curr solutions
 6: while consecutive non-improving iterations < max. iterations do
 7:     for all service $s \in S_{\text{sorted}}$ do                            ▷ Embedding Procedure
 8:         curr $\leftarrow$ place dummy instances of $c_{\text{in}}$ at $V_s^{\text{in}}$ and of $c_{\text{eg}}$ at $V_s^{\text{eg}}$
 9:         for all instance $i$ in component order $\langle c_{\text{in}}, c \in C_s \rangle$ do
10:             if $i$ is tabu or not used then
11:                 curr $\leftarrow$ remove $i$ and continue with the next instance
12:             curr $\leftarrow$ map flows to paths; add required instances; avoid tabu
13:     tabu $\leftarrow$ random instance from curr                          ▷ Iterative Improvement
14:     if curr is better than before then
15:         Update incumb and best solution
16:     else if curr is just slightly worse than before then
17:         Update incumb with certain probability
18:     curr $\leftarrow$ incumb
19: return best
```

compute resources for deploying each service by taking information about incoming flows and all service components into account. This sorting helps in the embedding procedure by embedding resource-heavy services first when there are still enough resources, i.e., before embedding more light-weight services. Finally, BSP initializes `tabu` and three initially empty solutions `curr`, `incumb`, and `best` (line 4 and 5), which are used during iterative improvement (Section 4.2.3).

4.2.2. Embedding Procedure

After initialization, BSP creates an initial solution `curr`. Particularly, BSP embeds all active services in the predefined order S_{sorted}, starting with the services that have the highest estimated resource requirements (line 7). BSP introduces auxiliary ingress and egress components, c_{in} and c_{eg}, and places dummy instances of these auxiliary components at all ingress and egress nodes, V_s^{in} and V_s^{eg}, for each service s (line 8). These auxiliary dummy instances do not consume any compute resources but enable a more consistent embedding procedure later on. The heuristic then starts embedding the current service s by iterating through any already existing instances, starting at the instances of c_{in} and following with instances according to the order of components in C_s (line 9). As traffic changes over time, some of the previously existing instances may have become unused, i.e., have no more incoming flows, and are therefore removed by BSP (lines 10 and 11). Similarly, BSP removes the current instance i if its placement is declared tabu (i.e., i is `tabu`), which happens later in the iterative improvement phase.

For active instances with incoming flows, BSP computes the flows' outgoing data rates based on $\lambda_c^{\text{out}}(\lambda)$, which needs to be known for each component c. It uses these calculated data rates to adjust the scheduling of outgoing flows to instances of the next component c' (according to C_s) and for mapping them to suitable paths (routing). It deallocates resources for any departed flows that no longer exist and allocates resources for any new flows (line 12). In the special case of $c' = c_{\text{eg}}$, BSP schedules and routes each flow f to its egress node v_f^{eg}. Otherwise, it tries to schedule new flows to already existing instances of c', routing them via the precomputed paths. For each new flow f, it considers all available instances of c' that have enough free resources to process f, taking flow data rate λ_f and resource requirements $r_{c'}(\lambda)$ into account. Out of these candidate instances, BSP selects the closest instance in terms of path delay and schedules f to be processed there. If there are no existing instances of c' or they do not have enough resources to process f, BSP places a new instance of c' at another node. Again, it selects this node based on its remaining resources and path delay. In doing so, it avoids selecting nodes that are declared tabu. Once all flows leaving current instance i are scheduled and routed to instances of the next component c', BSP repeats the same procedure for the next instance (line 9).

4.2.3. Iterative Improvement

While the embedding procedure ensures correct embeddings of all services $s \in S$ and tries to optimize the embedding in terms of the aforementioned objectives, its sequential and somewhat greedy approach can still lead to suboptimal results. Therefore, BSP modifies the current solution (`curr`) iteratively based on tabu search [94] by picking a random instance in `curr`, excluding dummy instances of c_{in} and c_{eg}, and declaring it as *tabu* (line 13). The `curr` solution is then reset and recreated using the embedding procedure again but disallowing to place the tabu instance at the same location as before (cf. destroy-and-repair principle of large neighborhood search [202]). This leads to a different distribution of instances and scheduling of flows, possibly decreasing or avoiding over-subscription and improving the desired objectives.

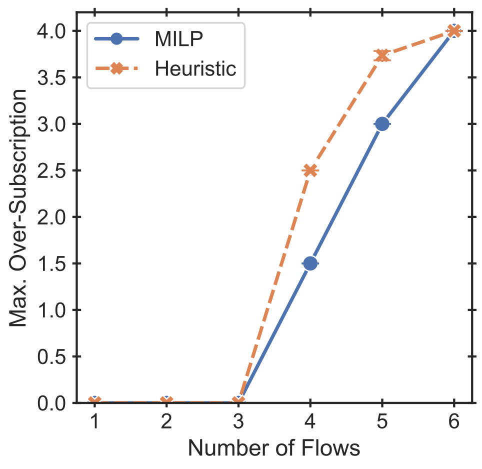
The heuristic modifies the embeddings of all services in multiple iterations while maintaining three different solutions, which are updated in lines 14–18: The current solution `curr`, which is modified during the embedding procedure, the incumbent solution `incumb`, which is used to reset `curr` after each iteration (line 18), and `best`, which is the overall best solution found so far. After each iteration, BSP discards unsuccessful modifications but also enables exploration of slightly worse solutions without losing the best found solution (lines 16 and 17). Hence, the tabu-based improvement scheme allows to overcome local optima while reusing the efficient embedding procedure iteratively. The iterative improvement phase ends when the incumbent solution (`incumb`) could not be improved for a predefined number of consecutive iterations (line 6). At this point, BSP terminates and returns the best solution for scaling and placing all services and for scheduling and routing all active flows (line 19).

4.3. Evaluation

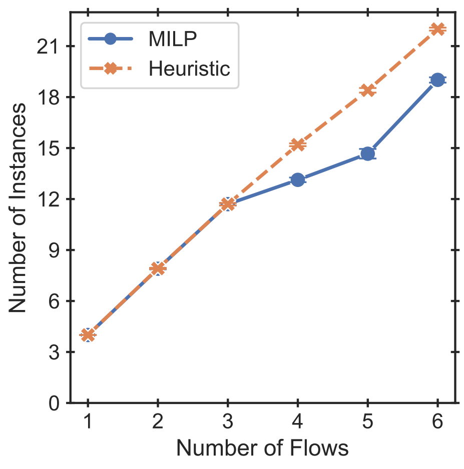
In this section, I evaluate the previously described BSP heuristic in terms of the four optimization objectives and compare it against the MILP approach, which is described in the paper [S2]. Generally, the MILP approach finds optimal solutions but requires excessive runtimes, whereas the heuristic is much faster and produces suboptimal yet good solutions. Section 4.3.1 describes the evaluation setup, Section 4.3.2 compares the BSP heuristic against the optimal MILP approach in a small scenario, and Section 4.3.3 evaluates its scalability and online coordination capabilities in a much larger scenario.

4.3.1. Evaluation Setup

I implemented BSP in Python 3.5 and performed all simulations on machines with an Intel Xeon E5-2695 v3 CPU running at 2.30 GHz, using GNU Parallel [244] for parallelization across CPU cores. For reproducibility and reuse, all code is publicly available on GitHub [S12]. For the evaluation, I consider a typical video streaming service consisting of three components, particularly, a cache, a server, and a video optimizer [72]. Furthermore, I use two real-world network topologies from the SNDlib library [194] with link propagation delays d_l calculated based on the distances between the geographical locations of the nodes. For comparison against the MILP approach in Section 4.3.2, I use a very small network with few flows due to the high complexity and long runtime of the MILP approach. Specifically, I use the western half of the Abilene network with 6 nodes and 14 directed links and simulate traffic with 1–6 flows arriving at random ingress nodes. In contrast, I evaluate the scalability and online coordination of the heuristic in Section 4.3.3 using up to 30 flows arriving randomly at 10 ingress nodes on the largest network topology from SNDlib, the “Brain” network, with 161 nodes and 664 directed links. In both networks, flows arrive with unit data rate ($\lambda_f = 1$) and node and link capacities are set uniformly to $\text{cap}_v = 4$, $\text{cap}_l = 4$ in the Abilene network and $\text{cap}_v = 10$, $\text{cap}_l = 10$ in the Brain network. Based on preliminary experiments, I set the hyperparameters of the BSP heuristic as follows: 1. Maximum number of consecutive non-improving iterations before terminating the algorithm (line 6 in Algorithm 1): 30. 2. Threshold for considering a solution as just “slightly worse” and as candidate for the new `incumb` solution in line 16: 110 % compared to objective score of the current `incumb` solution. 3. Probability for accepting such slightly worse solutions as new `incumb` solution (line 17): 50 %.

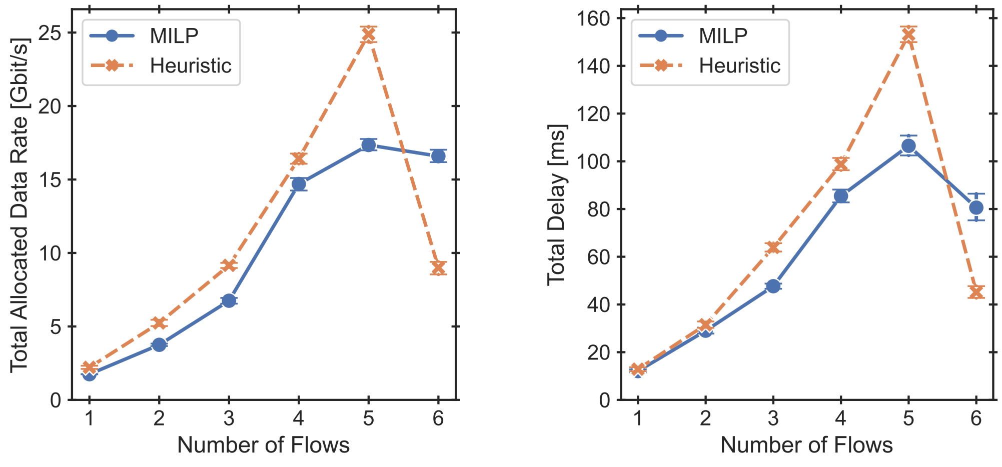


(a) Maximum over-subscription



(b) Number of instances

Figure 4.1.: Comparison between the BSP heuristic and MILP in terms of a) maximum over-subscription over all nodes and b) number of added instances.



(a) Total allocated data rate

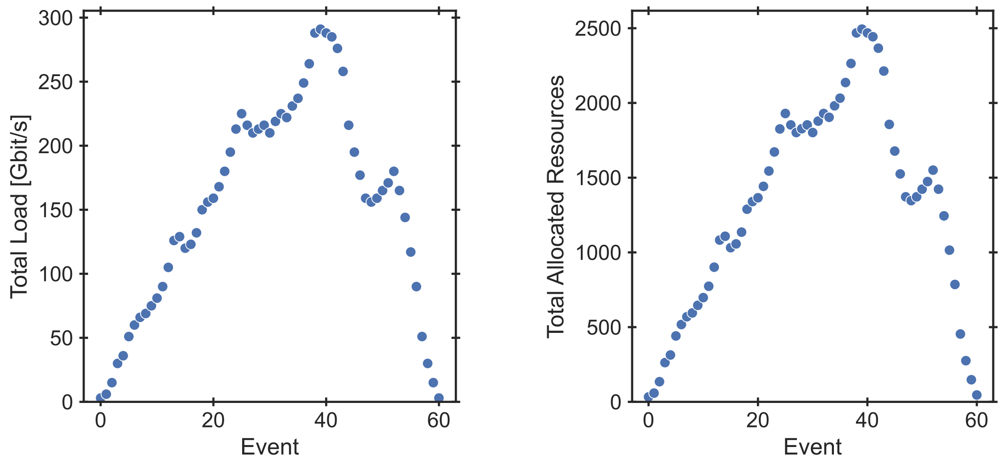
(b) Total delay

Figure 4.2.: Comparison between the BSP heuristic and MILP in terms of a) total allocated data rate and b) total delay over all links.

4.3.2. Comparison Against the MILP Approach

I first compare the BSP heuristic against the MILP approach in the small Abilene west network with 1–6 flows simulating increasing load. Figures 4.1 and 4.2 compare the two approaches in terms of the four optimization objectives stated in Section 4.2, showing the mean and 95 % confidence interval over 300 repetitions. The objective with highest priority is avoiding or minimizing over-subscription of available resources (both at nodes and links). While link capacities are never over-subscribed in the scenario here, Figure 4.1a shows the maximum over-subscription of nodes' compute resources with increasing load. Both approaches completely avoid over-subscription for 1 to 3 flows by distributing the load over different instances and nodes. With 4 or more flows, the available resources no longer suffice and are over-subscribed relative to the number of flows. The figure shows that the heuristic matches the optimal results by the MILP for 1–3 and 6 flows. For 4 and 5 flows, it is slightly worse and finds solutions with higher over-subscription (up to 67 %).

These observations are similar for the other three objectives: The heuristic finds good solutions that closely approximate the optimal results of the MILP with similar numbers of deployed instances (Figure 4.1b), allocated link data rate (Figure 4.2a), and total delay (Figure 4.2b). However, with 6 flows a special case occurs where the heuristic even outperforms the MILP in terms of total allocated data rate and total delay (Figure 4.2). Here, the heuristic uses additional instances compared to the MILP (Figure 4.1b), but places these extra instances closer to the ingress nodes and routes the flows via shorter paths, allocating less data rate in total and reducing total delay.



(a) Total load

(b) Total allocated node resources

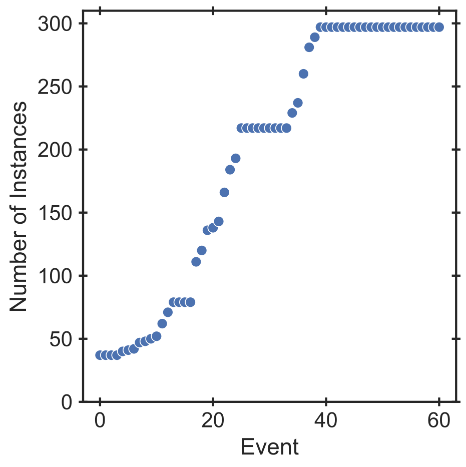
Figure 4.3.: During online coordination with the BSP heuristic, the allocated node resources closely follow the current load.

4.3.3. Scalability and Online Coordination

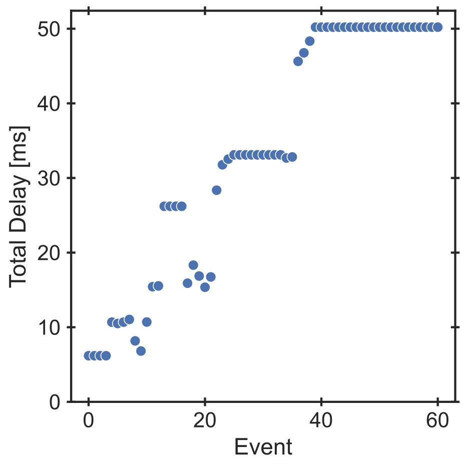
Where the MILP needs minutes to hours, the heuristic finds solutions in milliseconds to seconds. As an example, the worst case runtimes in the small scenario of Section 4.3.2 with 4 flows were 137.1 hours for the MILP approach compared to 6.7 s for the heuristic. In this section, I further evaluate the scalability and online coordination capabilities of the heuristic on the largest network of the SNDlib library (called "Brain" network) with 161 nodes and 664 directed links. Here, I apply the BSP heuristic in an initially empty network where the load first increases and then decreases again, triggering 60 coordination events for each flow arrival or departure.

Figures 4.3 and 4.4 show the results of the BSP heuristic in this large-scale scenario. As shown in Figure 4.3a, the load first increases and then decreases again over the course of 60 events. At the same time, the heuristic adapts to the changing load by scaling the involved service components and allocating resources accordingly. The total compute resources allocated over all nodes (Figure 4.3b) closely follow the changes in load, indicating that BSP handles changing load well and accurately adapts its resource allocation.

Figure 4.4 shows an important consequence of the objectives' prioritization (Section 4.2). With increasing load, the number of deployed instances and the total delay also increase because BSP has to distribute the traffic across more nodes to avoid over-subscription, resulting in longer paths and higher delays. However, the total number of instances and the total delay stay high even after the load starts decreasing around event 40. This is because, here, minimizing the number of *added or removed* instances has a higher priority than minimizing the total delay to limit overhead of migrating instances. Adhering to this prioritization, the existing instances (possibly placed farther from the ingress nodes) are still used with reduced load rather than removing or migrating them closer to the ingress nodes. Intermediate drops in total delay (Figure 4.4b) are due to BSP optimizing routing paths, adjusting to changed load. This indicates that the BSP heuristic successfully optimizes the given optimization objectives and



(a) Number of instances



(b) Total delay

Figure 4.4.: With fluctuating load (Figure 4.3a), the BSP heuristic dynamically adds new instances, leading to increasing total delay.

respects their prioritization. Even in this large-scale scenario, it coordinates network and services successfully, dynamically adapting service scaling and placement as well as flow scheduling and routing. The average runtime per event in this large scenario was 42.3 s, which is acceptable in scenarios with few but long flows that arrive and depart sporadically (Section 3.1.3, Case I). If required, the runtime can be reduced by terminating after fewer non-improving iterations (see hyperparameter settings in Section 4.3.1). However, a shorter iterative improvement phase would likely also lead to reduced service quality.

4.4. Conclusion

BSP is my first approach for solving the network and service coordination problem of Chapter 3 by jointly optimizing scaling, placement, scheduling, and routing. As centralized coordination approach, BSP requires detailed, global knowledge of network, services, and traffic to successfully control network and service coordination. It leverages its central view and control to find optimal solutions with the MILP approach or close-to-optimal solutions with the heuristic algorithm. While the heuristic is much faster than the MILP approach, it still requires detailed, global knowledge of all active flows in the network. Hence, it is more suitable for few, long flows (Case I in Section 3.1.3) than for many, short flows (Case II). Furthermore, its complexity grows with increasing network size. This is a conceptual issue of centralized coordination approaches, which is particularly true for the MILP approach but also holds for the heuristic (e.g., more paths to precompute and more candidate nodes to select from). As a consequence, coordination runtimes of centralized approaches increase with growing network size, possibly limiting their practical applicability in large networks. How to overcome this inherent scalability issue of centralized approaches? The following Chapter 5 tries to answer this question by proposing an alternative coordination approach that does not require global knowledge and control.

5. Hierarchical Coordination

Network and service coordination is commonly addressed through centralized approaches, where a single coordinator knows everything and coordinates the entire network globally (like the Bidirectional Scaling and Placement Coordination Approach (BSP) in Chapter 4). While such centralized approaches can reach global optima, they do not scale well to large, realistic networks. In contrast, distributed approaches scale well, but easily sacrifice solution quality due to their limited scope of knowledge and coordination decisions. To this end, I propose a hierarchical coordination approach in this chapter that combines benefits from both centralized and distributed approaches and tries to find intermediate solutions with good solution quality and good scalability. In doing so, the approach considers a network that is divided into multiple hierarchical domains and optimizes coordination in a top-down manner. I compare this hierarchical approach with a centralized approach in an extensive evaluation on a real-world network topology. My results indicate that hierarchical coordination can find close-to-optimal solutions in a fraction of the runtime of centralized approaches. Similar to BSP in Chapter 4, this hierarchical approach is conventional in that it relies on a detailed procedure designed by human experts, here including the formulation of a Mixed-Integer Linear Program (MILP).

This chapter is based on my paper [S18], which relies on concepts, code, and evaluation results created by Mirko Jürgens for his master's thesis [124]. I proposed the initial topic and idea and advised his thesis, providing feedback and discussing ideas in weekly meetings to structure and develop these ideas for defining and approaching the problem as well as steering the work into the desired direction. Based on the convincing outcome of the thesis, I also compiled the results and wrote the published paper. This chapter contains verbatim copies of the following paper [S18]: Stefan Schneider, Mirko Jürgens, and Holger Karl. "Divide and Conquer: Hierarchical Network and Service Coordination." In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IFIP/IEEE. 2021, © 2021 IEEE. In this chapter and throughout my dissertation, I consistently write in the first-person singular form ("I" rather than "we") for ease of reading. The source code corresponding to this chapter is publicly available on GitHub [S17].

5.1. Introduction

Network and service coordination is relevant for a variety of real-world use cases (Chapter 2) yet challenging due to many influencing factors (Chapter 3). Taking just some of these factors into account already leads to optimization problems that are NP-hard [61, S2]. Even with heuristic, non-optimal approaches, practical scalability of centralized coordination approaches is usually limited to small or medium-sized networks. A natural approach is hence to split large networks into smaller domains, which are then coordinated independently in a distributed fashion. While this provides reasonable results within shorter runtimes, solution quality may be sacrificed due to lack of global knowledge and necessarily suboptimal inter-domain coordination.

To address these issues, I propose a novel hierarchical coordination approach that combines benefits of both centralized and distributed approaches. The approach follows a divide-and-conquer strategy where the network can be divided into arbitrary levels of hierarchy. First, in a bottom-up phase, lower hierarchical levels aggregate and advertise information to the higher levels (cf. "one big switch" abstraction in Software-Defined Networking (SDN) [129]). In the following top-down phase, high-level coordinators make inter-domain coordination decisions based on the advertised information. All child coordinators then refine the coarse-grained decisions of their parents in parallel.

A challenge here is that node and link capacity constraints must be respected on all hierarchical levels. Decisions of higher-level coordinators should guide child coordinators by reducing their decision space but must not keep them from finding a valid solution if any feasible solution exists. If no feasible solution exists for embedding a flow, e.g., because resources are already highly utilized, this should be noticed at the top of the hierarchical levels to quickly reject the corresponding flow without the unnecessary overhead of recursing to lower hierarchical levels. The information advertised from lower hierarchical levels, e.g., about available resources, needs to be detailed and relevant enough to enable meaningful decisions at higher hierarchical levels. Still, it should be aggregated and abstract enough to reduce coordination complexity on higher levels.

To navigate this trade-off, I define a suitable information advertisement scheme. I then formalize the hierarchical coordination approach as MILP and optimize it numerically. In the extensive evaluation on a real-world network topology, I compare the hierarchical approach with an equivalent centralized approach, using the same MILP formulation for a flat, single-level hierarchy. I find that the hierarchical approach achieves comparable solution quality but is more than 10x faster on average than the centralized approach. Overall, the contributions of this chapter are:

- Section 5.4 proposes a generic, hierarchical coordination approach that works with any given hierarchical structure.
- Section 5.5 formalizes an MILP that is solved numerically by coordinators on each hierarchical level.
- In Section 5.6, I evaluate the approach on a real-world network topology, comparing it against a centralized approach.
- To facilitate reproduction and reuse, the implementation is open source [S17].

5.2. Related Work

Most authors propose centralized approaches to make global decisions for coordinating services in the entire network [182, 22, 28, 9, 144, S2, 161]. These approaches typically only work in small networks and do not scale to practical, large-scale networks [28]. For example, Moens and De Turck [182] propose a centralized optimization approach for placing Virtual Network Function (VNF) instances in a hybrid setting, coexisting with physical network functions. Unlike the approach proposed here, the authors do not consider dynamic scaling of services or traffic routing between component instances. Similarly, other proposed approaches [23, 22, 28, 9, 144] focus on placement of instances and/or traffic routing but disregard flexible scaling of services according to the current demand. Furthermore, such centralized numerical optimization approaches are severely limited by their complexity and typically only work in small networks [28]. Thus, authors typically propose heuristic algorithms that trade off reduced solution quality with faster runtime. I argue that centralized heuristic algorithms are still inherently limited by their centralized view and decisions when scaling to practical, large networks. Houidi et al. [115] propose a distributed approach, splitting the network into separate domains, in which services are coordinated independently in parallel. While this approach eliminates the scalability issues of centralized approaches, it is also prone to coordinating services suboptimally since each domain coordinator's knowledge and decisions are limited to its own domain. To the best of my knowledge, I propose the first hierarchical approach for network and service coordination, which solves these inherent limitations. By dividing the network into hierarchical levels it can be optimized in a scalable and distributed, yet coordinated fashion.

Nevertheless, authors have proposed hierarchical approaches in related areas [78, 175]. In particular, Virtual Network Embedding (VNE) is well-studied [79] and closely related to my problem. Samuel et al. [221] propose a distributed and hierarchical approach that greedily solves the VNE problem while different domains hide as much information from each other as possible because they belong to competitors. While this perspective is interesting, I assume domains to cooperate and share relevant topology information to improve coordination. Due to limited information sharing and its greedy nature, the approach by Samuel et al. easily gets stuck at suboptimal solutions or fails to find any feasible solution. In the latter case, embedding steps are repeatedly reverted and applied differently to find a feasible solution. In my approach, I ensure that any solution suggested by the top hierarchical level can be refined into a feasible embedding.

In line with this chapter, Ghazar et al. [90] assume that domains cooperate. However, the authors do not aggregate or abstract any information such that computations on high hierarchical levels become very expensive, comparable to centralized approaches. Hence, the authors skip higher levels and directly select an intermediate hierarchical level to embed an incoming flow in one of its sub-domains. If this is not possible because the domains are too small, the process is repeated on the next higher level. This leads to considerable overhead for large embeddings, e.g., where ingress and egress are far apart. Their approach is very sensitive to the selection of the initial hierarchical level in which a flow is handled. Selecting a hierarchical level that is too low or too high leads to considerable additional overhead, worse solutions, or unnecessarily rejected flows. If it is too low, its sub-domains are very small and might not be able to embed the request, leading to many costly, unsuccessful embedding attempts until a sufficiently high hierarchical level is reached. If the initial hierarchical level is too high, its sub-domains cover large parts of the substrate network increasing embedding complexity. In contrast, my approach always starts at the top level and efficiently coordinates across domains by aggregating and simplifying information of lower hierarchical levels.

In general, the network and service coordination problem addressed here (Section 5.3) is considerably more complex than VNE. Unlike typical VNE approaches, I consider routing from ingress to egress but also dynamic reuse of components across (or within) services and dynamic scaling, i.e., flexible number of instances and resources per instance. Such joint coordination is important to successfully balance trade-offs [203, 61, S16]. Overall, there is a stronger interdependence between decisions of different hierarchical levels and domains, making the problem more challenging than VNE.

Finally, hierarchical approaches are common in traffic engineering, which is a subproblem of network and service coordination. For example, Multiprotocol Label Switching (MPLS) uses a Path Computation Element (PCE) to find shortest paths across different hierarchical domains (called autonomous systems in this context) [137, 55, 58]. Similar to my approach, topology information from these different domains can be abstracted and aggregated through topology aggregation mechanisms [36]. Related to the aggregation approach proposed in this chapter, Secci et al. [229] propose a full mesh aggregation containing the most relevant topology information. Still, the authors assume that advertised intra-domain paths do not overlap, i.e., do not share links. In contrast, the approach in this chapter supports overlapping intra-domain paths and explicitly advertises constraints to avoid overloading shared links. Hence, my approach allows more paths to be advertised such that better embeddings can be found.

5.3. Problem Statement

I address the network and service coordination problem where users request services consisting of chained components in a network of distributed nodes as described in Chapter 3. In this chapter, I assume that flows can be split over multiple paths from ingress to egress, which is possible in many modern SDN architectures [272]. Alternatively, additional constraints could be added to the MILP formulation to ensure unsplittable flows [140]. With splittable flows, the problem of flow scheduling, where each flow is processed by exactly one instance per component (Section 3.2.2), is turned into a load balancing problem, where a single flow can be split and processed by multiple instances in parallel. The coordination approach needs to decide how to split each flow into different portions, which portions of the flow to assign to which instances, and how to route these flow portions through the network. Hence, the coordination problem here entails joint scaling, placement, load balancing, and routing. Overall, the problem and approach of this chapter focuses on scenarios with few and long flows (Case I in Section 3.1.3) but is not strictly limited to such scenarios.

As discussed in Section 3.1.2, services consist of chained components, which may be reused across different services or even within a single service. While the proposed hierarchical coordination approach (Section 5.4) does not require full global knowledge of the entire network, it does require detailed knowledge of the involved service components, including their resource requirements and outgoing data rate in terms of functions $r_c(\lambda)$ and $\lambda_c^{\text{out}}(\lambda)$. To ensure that the MILP formulation (Section 5.5) indeed only contains linear constraints and remains solvable, $r_c(\lambda)$ and $\lambda_c^{\text{out}}(\lambda)$ must be linear functions. To model the relationship between traversing data rate and resource requirements or outgoing data rate even more accurately, the approach could easily be extended to use piece-wise linear functions [60].

The optimization goal in this chapter is to coordinate network and services such that all incoming flows are processed successfully while minimizing, first, the number of placed instances and, second, the total delay. This is challenging because processing many flows successfully may require distributing load across many nodes and instances, also leading longer routing paths and higher delay. Still, placing fewer instances can be important to reduce costs for licensing or resources and shorter delays can improve service quality. If a flow cannot be processed successfully with the given resources, it has to be rejected and is dropped. I formalize this objective in Section 5.5.1.

5.4. Hierarchical Coordination Approach

In this section, I describe the hierarchical coordination approach, first, providing a high-level overview of the overall design, and then going into detail in Sections 5.4.1–5.4.3. The main idea of the approach is to divide the network into smaller domains and coordinate them in a hierarchical manner. Each domain is a part of the network that may recursively consist of sub-domains, forming a hierarchy. This hierarchical approach allows both efficient parallel coordination of different domains yet enables coordination between domains for highly optimized results. I assume that dividing the network into domains and sub-domains may depend on business or legal aspects (or other aspects like node locality) and is therefore out of scope and happens before coordination starts. Hence, it is not possible to rely on two levels of hierarchy (or any other fixed number) when designing the hierarchical coordination approach. Since the approach cannot rely on a specific number of hierarchical levels, it cannot assign fixed predefined tasks to specific hierarchical levels, e.g., coarse-grained scaling and load balancing on the higher level and detailed placement and routing on the lower level. Instead, it is important that the approach is not tied to any structure but works with any given domains and hierarchical levels. This also means that each domain may recursively consist of an arbitrary number of sub-domains, possibly spanning large parts of the substrate network. Hence, aggregating and hiding unnecessary information of lower-level sub-domains is crucial to reduce the problem and decision space and keep coordination tractable. The design of the hierarchical approach ensures a generic coordination procedure that can be applied to any hierarchical level in a unified way, i.e., without special, hard-coded tasks for individual levels, and focuses on hiding unnecessary details of lower hierarchical levels. The approach makes no assumptions about the size or structure of the hierarchy and supports arbitrary numbers of hierarchical levels and domains per level.

Given a specific hierarchy, the approach consists of two phases: First, domains aggregate and advertise relevant information to their coordinators in a bottom-up manner (detailed in Section 5.4.2). Second, based on this information, the coordinators make coordination decisions in a top-down manner (Section 5.4.3). I choose top-down coordination to allow high-level coordinators to optimize inter-domain decisions and guide lower-level coordinators. Higher-level coordinators can leverage their coarse-grained yet wider view of multiple lower-level domains to distribute load between different domains accordingly. Starting coordination directly at a lower level can easily lead to worse solutions if coordinators of lower-level domains coordinate incoming flows independently without guidance from higher-level coordinators and without being aware of surrounding domains. For example, coordinating a flow with an egress node in another domain is difficult without knowing about available resources, data rates, and path delays in the other domain. Should a lower-level coordinator ensure that the flow is processed completely by processing it in its own domain even if the domain is already highly loaded and may become over-subscribed with more flows? Or should the coordinator simply forward the flow to another domain and rely on this domain to process the flow, hoping that there are sufficient resources? Both decisions are risky without knowledge of the other domain or guidance from a higher-level coordinator, which is exactly a limitation of common distributed coordination approaches. Similar questions also arise in terms of available link data rate and path delays when balancing load and routing flows between different domains.

In contrast, the proposed hierarchical coordination approach facilitates lower-level coordination decisions through coarse-grained guidance from higher-level coordinators. I ensure that each high-level coordination decision can be further refined into a feasible solution or directly reject requests at the top level. There is a general trade-off between accepting vs. rejecting incoming flows at the top level if it is unsure whether they can be embedded successfully. Accepting and trying to embed all flows ensures that no flows are rejected unnecessarily but may incur high overhead: It requires recursing to lower hierarchical levels even for flows that are infeasible to embed, possibly even jumping up and down between levels to backtrack and try to fix infeasible embeddings until they are eventually rejected (comparable to type II errors in statistical hypothesis testing). Instead, only accepting flows at the top level that can surly be embedded successfully avoids this overhead but risks rejecting some flows that could potentially be accepted after all (comparable to type I errors). For the proposed hierarchical coordination approach, I chose the latter design option to avoid frequent and significant overhead of recursing to lower hierarchical levels for infeasible flows. I still try to avoid type I errors and to accept as many flows as possible by providing relevant information to top-level coordinators, enabling them to estimate available resources accurately and make informed coordination decisions. To enable effective top-down coordination in phase 2, a main challenge is therefore to advertise enough and relevant yet reduced and aggregated information from lower levels in phase 1.

After introducing the notation for domains and hierarchies in Section 5.4.1, I describe phase 1 in Section 5.4.2 and detail which information is advertised. Section 5.4.3 explains phase 2 in more detail. Algorithm 2 shows the overall algorithm including both phases.

Algorithm 2 Hierarchical Coordination Algorithm

```
1: for $k = 1$ up to $\hat{k} - 1$ do                                  ▷ Phase 1
2:     for $i \in \{1, ..., n_{k-1}\}$ in parallel do
3:         Aggregate sub-domain information as $\bar{D}_i^{k-1}$
4:     Advertise $\bar{D}_j^k = \{\bar{D}_i^{k-1} | \forall i\}$ to coordinator $j$ on level $k+1$
5: for $k = \hat{k}$ down to 1 do                                      ▷ Phase 2
6:     for $j \in \{1, ..., n_k\}$ in parallel do
7:         Embed flow $f_j^k$ into $\bar{D}_j^{k-1}$ by solving the MILP (Section 5.5)
8:         Split flow $f_j^k$ into $f_i^{k-1}$ for all coordinators $i$ on $k-1$
```

5.4.1. Domains and Hierarchies

I denote the total number of hierarchical levels as $\hat{k}$ and a specific level as $k \leq \hat{k} \in \mathbb{N}_0$, where $k = 0$ is the substrate network $G = G^0$. Each domain at a hierarchical level k is coordinated by a coordinator on the next higher level $k+1$. In a typical flat, centralized approach with $\hat{k} = 1$, the entire network forms a single domain at $k = 0$, which is coordinated by a single coordinator at $k = 1$. There are no coordinators at $k = 0$. This hierarchical approach supports arbitrarily deep hierarchies with $\hat{k} \geq 1$. In the example of Figure 5.1, the substrate network $G^0 = (V^0, L^0)$ on $k = 0$ is split into $n_0 = 3$ separate domains D_1^0, D_2^0, D_3^0 with $D_i^0 = (V_i^0, L_i^0)$. Each domain D_i^0 on $k = 0$ is coordinated separately by its coordinator on $k = 1$, in parallel with the other domains D_j^0. At level $k = 1$, nodes are grouped again into domains that are handled by coordinators on $k = 2$ (a single domain D_1^1 in Figure 5.1). This definition recursively extends to an arbitrary number of $\hat{k}$ hierarchies.

While I assume that all nodes V^k on level k belong to some domain D_i^k (i.e., $V^k = \bigcup_{i=1}^{n_k} V_i^k$),

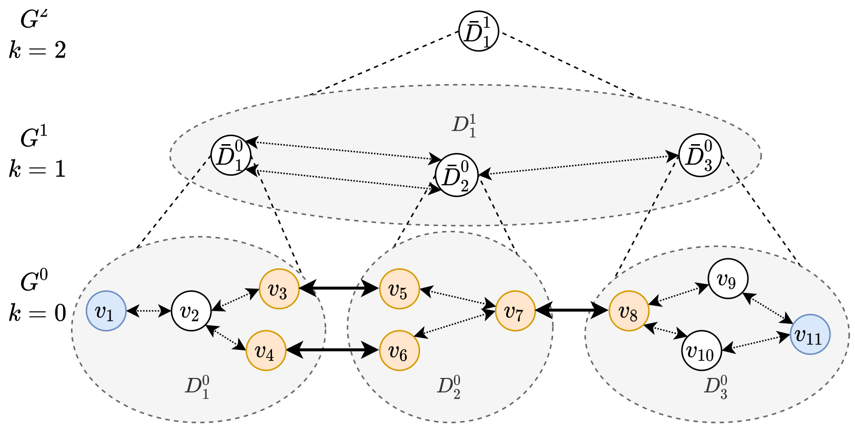


Figure 5.1.: Example with $\hat{k} = 2$ hierarchical levels. Ingress and egress nodes are shown in blue and border nodes in orange. (Figure from [S18]; © 2021 IEEE.)

not all links L^k are part of some domain. In particular, I distinguish between intra-domain and inter-domain links. Intra-domain links L_i^k connect nodes within a single domain D_i^k (lighter in Figure 5.1). Inter-domain links do not belong to any domain but connect nodes across two different domains (thicker in Figure 5.1). I define border nodes $B_i^k \subseteq V_i^k$ as the subset of nodes that have an inter-domain link to another domain (orange in Figure 5.1). Specifically, $B_{i,j}^k \subseteq V_i^k$ are the border nodes of domain D_i^k with links to domain D_j^k. Set $B_i^k = \bigcup_{j=1}^{n_k} B_{i,j}^k$ contains all border nodes of D_i^k with inter-domain links to any other domain (where $B_{i,i}^k = \emptyset$). For example in Figure 5.1, $B_2^0 = \{v_5, v_6, v_7\}$.

5.4.2. Bottom-Up Information Advertisement (Phase 1)

Each domain's coordinator scales and places services as well as routes traffic inside the domain. It needs to know about available compute capacity, data rate limitations, and delays within the domain. A domain on level k may comprise multiple levels of sub-domains and cover large parts of the network. Thus, it is crucial to hide unnecessary information of lower levels from higher-level coordinators to reduce the problem space and ensure scalability. In line with related work [30], I consider an intuitive scalability definition where an approach is scalable if it performs gracefully with increasing input size, e.g., in terms of load or network size. To ensure that the decision space of higher-level coordinators stays manageable even in large-scale networks with many hierarchical levels or large domains, domains aggregate relevant information of their sub-domains and advertise it to their coordinators as follows (lines 1–4 in Algorithm 2).

Advertised Information

For a sub-domain D_i^k domain information $\bar{D}_i^k = (\mathcal{V}_i^k, \mathcal{L}_i^k, \mathcal{P}_i^k, \Phi_i^k)$ is advertised to the coordinator. The key idea is to advertise a reduced set of nodes $\mathcal{V}_i^k$ and replace hidden nodes and corresponding links with paths $\mathcal{P}_i^k$ that are annotated with aggregated information about compute

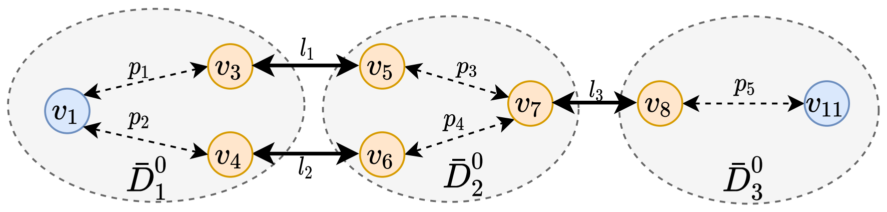


Figure 5.2.: Domain information $\bar{D}_1^1$ advertised to the coordinator on $k = 2$ in Figure 5.1. (Figure from [S18]; © 2021 IEEE.)

resources of the hidden nodes as well as data rate and delay information of the hidden links. In doing so, the advertised network topology of $\bar{D}_i^k$ is significantly smaller and simpler than the original, full topology of D_i^k. The simplified topology allows faster computation of coordination decisions, yet the annotation of paths with aggregated information ensures that coordinators know about the domain's compute resources, data rate limitations, and path delays. This information is important to determine whether a flow can be accepted and embedded successfully and enables meaningful coordination decisions that can be refined into valid embeddings. As a small example, Figure 5.2 illustrates the advertised domain information $\bar{D}_1^1$ to the coordinator on $k = 2$, containing aggregated information about sub-domains D_1^0–D_3^0 from Figure 5.1.

Generally, advertised information $\bar{D}_i^k$ includes a subset $\mathcal{V}_i^k \subseteq V_i^k$ of the domain's nodes. Subset $\mathcal{V}_i^k = \{V^{\text{in}} \cap V_i^k\} \cup \{V^{\text{eg}} \cap V_i^k\} \cup B_i^k$ includes all ingress and egress nodes within V_i^k as well as the domain's border nodes but no intermediate nodes. I selected these nodes to be advertised because they are most relevant for routing flows from their ingress via different domains (through the corresponding border nodes) to their egress. The advertised network in Figure 5.2 includes ingress v_1 and egress v_{11} as well as border nodes v_3–v_8 but not intermediate nodes v_2, v_9, and v_{10}, which are not as relevant for routing. In this small example, the majority of nodes is advertised to the coordinator. However, in larger networks with more intermediate nodes and additional levels of sub-domains, more nodes would be hidden and excluded from $\mathcal{V}_i^k$, simplifying the resulting network topology. To indicate how different domains are interconnected, domain D_i^k advertises all inter-domain links $\mathcal{L}_i^k$, which connect a border node of D_i^k with a border node of another domain.

Finally and crucially, domain D_i^k also advertises annotated intra-domain paths $\mathcal{P}_i^k$, which replace hidden nodes and corresponding links within the domain. These annotated intra-domain paths are complemented by additional restrictions Φ_i^k, which are necessary for overlapping paths and are discussed later on. For example in Figure 5.2, v_1 can reach v_3 via intermediate, hidden node v_2 and thus has a path p_1 to v_3. However, to keep the number of advertised intra-domain paths small, domains do not advertise all possible paths between any two nodes in $\mathcal{V}_i^k$, e.g., based on reachability. Instead, it is most relevant for coordinators to know how to forward and process flows on their way from their ingress to their egress, potentially via intermediate domains. Hence, for a domain D_i^k, I am interested in three kinds of paths inside domain D_i^k:

1. Paths from an ingress node in D_i^k to a neighboring domain D_j^k. For example, for domain D_1^0, from ingress v_1 in D_1^0 to neighboring domain D_2^0 (via border node v_3 or v_4) in Figure 5.1.
2. Paths from a neighboring domain D_j^k to an egress node in D_i^k. For example, for domain D_3^0, from D_2^0 (via border node v_8) to egress v_{11} in D_3^0.

3. Paths from a neighboring domain D_j^k traversing D_i^k and going to another neighboring domain $D_{j'}^k$. For example, for domain D_2^0, from D_1^0 via D_2^0 to D_3^0, i.e., paths from border nodes v_5 and v_6 to v_7.

Intra-Domain Path Selection

To find such paths with maximal data rate, the approach solves the corresponding maximum flow problem using Edmonds-Karp path selection [65] for each of the three described cases, i.e., using ingress, egress, and border nodes as sources and sinks. Note that the maximum flow problem, which requires a single source and sink, is applied individually to each ingress and egress node but not to each border node. When looking for paths to/from a neighboring domain D_j^k, all corresponding border nodes $B_{i,j}^k$ are grouped together and connected to a single, auxiliary source/sink node since traffic can be routed via any border node. In contrast, ingress and egress nodes cannot be grouped together but have to be treated individually since each flow requests a specific ingress and egress, and the coordination approach cannot freely choose an ingress and egress node.

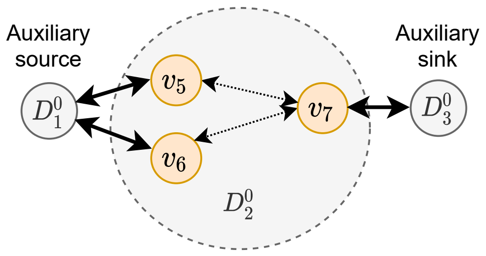


Figure 5.3.: Maximum flow problem for path selection in domain D_2^0 of Figure 5.1 from auxiliary source D_1^0 to auxiliary sink D_3^0, which connect to the corresponding border nodes.

For example, in Figure 5.1, the maximum flow problem is only solved once for domain D_2^0 to find intra-domain paths when routing flows from neighboring domain D_1^0 via D_2^0 to D_3^0. In doing so, border nodes v_5 and v_6 are connected to an auxiliary source node and v_7 to a sink as shown in Figure 5.3. The maximum flow problem is then solved once with the auxiliary source and sink rather than separately for each border node, resulting in two paths p_3 and p_4 shown in Figure 5.2. The selected paths can also be reused for the reverse direction from D_3^0 via D_2^0 to D_1^0 since all links and paths are symmetric and bidirectional. By default, the approach advertises all paths selected by the Edmonds-Karp algorithm, but I also consider restricting the number of advertised paths and evaluate the resulting solution quality and runtime in Section 5.6.4.

By replacing intermediate nodes and links with intra-domain paths, higher-level coordinators can leverage a smaller and simplified view of the network topology. Solving the maximum flow problem, intra-domain paths are selected such that coordinators can route traffic from ingress to egress nodes via intermediate domains with maximal data rate. Furthermore, Edmonds-Karp path selection [65] favors short paths by using Breadth-First Search (BFS) instead of Depth-First Search (DFS), which is used by other variants of the Ford-Fulkerson algorithm [83], leading to paths with fewer links and likely also shorter delay. To support coordination and ensure coordinators find valid solutions, the selected intra-domain paths $p \in \mathcal{P}_i^k$ are annotated with further information about available compute capacity, data rate limitations, and delay along each path. All three are necessary for coordinators to decide where to place instances and how to balance and route traffic within the domain without exceeding capacities and with short delay.

Specifically, an intra-domain path p is annotated with the maximally allowed data rate cap_p^L along the path obtained from solving the maximum flow problem. The path's delay d_p correspond to the sum of link delays along the path. I do not consider queuing delays, but they could be added based on the current load along the path. Similarly, compute resources are advertised as properties of a *path* rather than of individual nodes since intermediate nodes of sub-domains are hidden from the coordinator. The compute capacity of path p is denoted as cap_p^V and represents the sum of compute capacities of all nodes along the path, including the source and destination node. For domains at level $k \geq 2$, the approach works similarly based on the properties of advertised paths from the domains at $k-1$.

Overlapping Intra-Domain Paths

Lastly, domains advertise additional restrictions Φ_i^k, which are necessary for ensuring valid embeddings with overlapping intra-domain paths. Intra-domain paths may overlap and share underlying nodes or links, e.g., paths p_1 and p_2 in Figure 5.4 both share nodes v_1 and v_2 and corresponding link $l_{1,1}$. The previously described, aggregated information about compute resources and data rates includes resources of shared nodes and links multiple times, once for each path. For example, both p_1 and p_2 are annotated with the total compute capacity along the path, counting v_1 and v_2 twice. With just this information, higher-level coordinators are prone to overestimating the available resources in the sub-domain, which easily leads to invalid solutions with overloaded resources. Since, I want to ensure that any suggested, higher-level solution is indeed valid and does not exceed any capacities (to avoid overhead of backtracking and fixing invalid solutions), I introduce additional restrictions $\Phi = (\Phi^V, \Phi^L)$ to consider shared node and link resources of overlapping paths as follows.

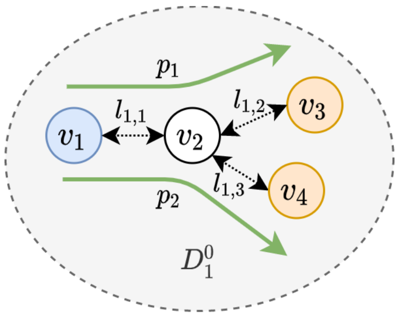


Figure 5.4.: Intra-domain paths p_1 and p_2 of domain D_1^0 in Figure 5.1 overlap and share nodes v_1 and v_2 as well as link $l_{1,1}$.

Lest partially overlapping paths overload shared compute resources, domains advertise compute restrictions $\phi^V \in \Phi^V$ for each segment of shared compute resources. Such segments resemble groups of shared nodes for domains on $k = 0$ or groups of shared sub-domain paths for domains on $k > 0$. A restriction $\phi^V = (\text{cap}_\phi^V, P_\phi^V)$ indicates the total compute resources cap_ϕ^V of the segment as well as the set of overlapping paths P_ϕ^V that share these resources. In the example of Figure 5.4, intra-domain paths p_1 and p_2 overlap and share the compute resources at nodes v_1 and v_2. Hence, a restriction $\phi_1^V = (\text{cap}_{\phi_1}^V, P_{\phi_1}^V)$ is advertised with $\text{cap}_{\phi_1}^V = \text{cap}_{v_1} + \text{cap}_{v_2}$ and $P_{\phi_1}^V = \{p_1, p_2\}$. When advertising the aggregated domain information $\bar{D}_1^0$ (Figure 5.2), the annotated paths together with this compute restriction ensure that higher-level coordinators can make informed decisions without overestimating compute resources. For convenience, I define $\Phi_p^V = \{\phi^V \in \Phi^V | p \in P_\phi^V\} \subseteq \Phi^V$ as subset of compute restrictions concerning path p and $\Phi_{i,k}^V = \{\phi^V \in \Phi_p^V | \forall p \in \mathcal{P}_i^k\} \subseteq \Phi^V$ as subset of compute restrictions concerning paths in domain D_i^k. Overall, the total compute resources cap_p^V of a path p can thus be represented by partly exclusive and partly shared compute resources:

$$\text{cap}_p^V = \text{cap}_p^{\text{excl}} + \sum_{\phi \in \Phi_p^V} \text{cap}_\phi^V \tag{5.1}$$

In the example of Figure 5.4, path p_1 uses v_3 exclusively but shares v_1 and v_2 with path p_2, i.e., $\text{cap}_{p_1}^{\text{excl}} = \text{cap}_{v_3}$ and $\text{cap}_{\phi_1}^V = \text{cap}_{v_1} + \text{cap}_{v_2}$ such that $\text{cap}_{p_1}^V = \text{cap}_{v_3} + \text{cap}_{v_1} + \text{cap}_{v_2}$.

Similar to shared nodes, overlapping paths may also share underlying links. Solving the maximum flow problem ensures that the selected paths do not exceed the available link data rate. For example, paths p_1 and p_2 in Figure 5.4 result from solving the maximum flow problem from ingress v_1 to neighboring domain D_2^0 (via border nodes v_3 and v_4) and are calculated such that they do not overload shared link $l_{1,1}$. However, if a domain has multiple ingress or egress nodes or multiple pairs of neighboring domains, the maximum flow problem is solved multiple times independently for the different source-sink pairs. The intra-domain paths resulting from these different, independent maximum flow problems may also overlap and are not guaranteed to respect link capacities. Indeed, overlapping paths resulting from different maximum flow problems may exceed the available link capacities. To avoid violating link capacities and to ensure valid solutions, I introduce routing restrictions Φ^L similar to compute restrictions Φ^V, which are advertised with the advertised domain information. Whenever multiple paths that resulted from different maximum flow problems overlap, a routing restriction $\phi^L \in \Phi^L$ is advertised for the overlapping routing segment. Again, the segment may consist of consecutive substrate links for domains on $k = 0$ or of consecutive sub-domain paths for domains on $k > 0$. A routing restriction $\phi^L = (\text{cap}_\phi^L, P_\phi^L)$ defines the bottleneck data rate cap_ϕ^L on the segment, which is shared by all paths in set P_ϕ^L. As before, $\Phi_p^L = \{\phi^L \in \Phi^L | p \in P_\phi^L\} \subseteq \Phi^L$ is the subset of routing restrictions concerning path p and $\Phi_{i,k}^L = \{\phi^L \in \Phi_p^L | \forall p \in \mathcal{P}_i^k\} \subseteq \Phi^L$ is the subset of routing restrictions concerning paths in domain D_i^k. Note that no routing restrictions need to be advertised for overlapping paths that resulted from the same maximum flow problem, e.g., paths p_1 and p_2 in Figure 5.4. Choosing the Edmonds-Karp algorithm for path selection is also useful here because it favors paths with fewer links (by using BFS instead of DFS), resulting in fewer overlapping routing segments and thus in fewer routing restrictions. Overall, $\Phi_i^k = (\Phi_{i,k}^V, \Phi_{i,k}^L)$ are the compute and routing restrictions advertised for domain D_i^k.

Design Options

As discussed at the beginning of Section 5.4, a key design decision for the hierarchical coordination approach is to favor rejecting possibly infeasible flows early (at the top hierarchical level) rather than recursing to lower hierarchical levels, producing potentially invalid embeddings. This design decision aims at avoiding unnecessary coordination overhead for infeasible flows that comes from recursing to lower hierarchical levels and potential attempts to fix invalid embeddings (e.g., through backtracking) until eventually rejecting the flow (comparable to type II errors). This decision comes at the cost of possibly rejecting flows that could be embedded successfully after all (cf. type I errors). The presented information advertisement scheme is designed to both ensure that all embeddings are always valid (or the flow is rejected) but also accept as many flows as possible.

Other design options for the information advertisement scheme come with different advantages and disadvantages. Generally, information about compute resources, available data rate, and link delays is crucial for coordinating scaling and placement as well as load balancing and routing. The chosen information advertisement scheme is rather expressive, i.e., including ingress, egress, and border nodes as well as annotated paths and restrictions, to provide detailed enough information to accept as many flows as possible. Advertising even more information, e.g., also intermediate nodes and corresponding paths, would mean that the advertised network topology is almost identical to the true topology without any significant aggregation and simplification, thus compromising scalability and defying the purpose of phase 1.

But why not simply advertise the aggregated compute resources, data rate, and delay for an entire domain rather than the proposed sophisticated scheme with annotated paths? Such a strong aggregation would completely hide the domain-internal topology and significantly simplify the advertised information and improve scalability. However, to ensure that computed embeddings are valid, the information needs to be aggregated and advertised in a very conservative way such that higher-level coordinators do not overestimate and overload the available resources. For example, this could mean advertising the minimal link data rate within the entire domain, the maximal path delay, and only the compute resources that are reachable within this delay. Consequently, coordinators could only rely on routing flows through the domain that do not exceed the minimal link data rate. Other flows with higher data rates would be rejected, even though there might be paths with higher data rates (i.e., without the bottleneck link) that could support these flows but that the coordinators are not aware of due to the strongly aggregated information. Overall, such a strong aggregation either violates the original design decision of ensuring valid embeddings (e.g., when advertising the average link data rate) or is very conservative (e.g., advertising the minimal link data rate), possibly leading to many unnecessarily rejected flows.

The information advertisement scheme proposed in this section does ensure valid embeddings, avoiding unnecessary overhead for infeasible flows, but also strikes a good balance between providing enough information to coordinate flows successfully and still reducing and simplifying the original network topology for scalability. In Section 5.6.4, I further explore this trade-off and evaluate solution quality and runtime of the approach when varying the amount of advertised information.

5.4.3. Top-Down Coordination Decisions (Phase 2)

In phase 2 (lines 5–8 in Algorithm 2), the advertised information from phase 1 is used for coordination as described in the following. Phase 2 starts at the highest hierarchical level $\hat{k}$ and works top-down, where each coordinator optimizes coordination in its own domain using the MILP formulation in Section 5.5. Inside the MILP formulation, the advertised information is used in constraints to ensure that solutions of the MILP are indeed valid embeddings that do not exceed any node or link capacities. The hierarchical coordination approach is generally not tied to this MILP formulation. Alternatively, any other functionally equivalent optimization algorithm, e.g., a heuristic, could be used by the coordinators without affecting the general hierarchical coordination framework. Independent of the specific optimization method, a coordinator on level $k+1$ only knows the advertised information $\bar{D}_i^k = (\mathcal{V}_i^k, \mathcal{L}_i^k, \mathcal{P}_i^k, \Phi_i^k)$. Hence, the coordinator scales and places services and routes traffic on the advertised *paths* rather than directly on substrate nodes or links.

For example, assume the top-level coordinator on $\hat{k} = 2$ in Figure 5.1 needs to handle an incoming flow for a service s consisting of two components $C_s = \langle c_1, c_2 \rangle$ with ingress v_1 and egress v_{11}. The coordinator only knows the advertised domain $\bar{D}_1^1$ shown in Figure 5.2. Based on this information, it may decide to place an instance of c_1 on path p_1 close to the ingress and an instance of c_2 on p_5 close to the egress. It could then route the traffic from v_1 in D_1^0 through D_2^0 to v_{11} in D_3^0 along p_1, l_1, p_3, l_3, p_5 (Figure 5.2).

Refining Coarse-Grained Coordination Decisions

This coordination decision on $k = 2$ then needs to be refined by coordinators on $k = 1$. These child coordinators decide the specific scaling, placement, load balancing, and routing within each domain (D_1^0, D_2^0, D_3^0 in the example), again by solving the MILP. Using the advertised information, including the compute and routing restrictions, as input for the MILP, the parent coordinator ensures that its decisions are feasible and can lead to valid solutions. To build on and refine the decisions of the parent coordinator, the original flow f_j^k handled on level k is adjusted and split into logically separate flows f_i^{k-1} for all child coordinators on $k-1$. In the aforementioned example, c_1 was placed on p_1 and traffic routed from ingress v_1 via p_1, v_3, l_1 to domain D_2^0. Hence, the child coordinator of D_1^0 on $k = 1$ would receive a flow for a service consisting just of $C_s = \langle c_1 \rangle$ with ingress node v_1 and egress v_3. Similarly, the coordinator of D_2^0 would receive a flow that just needs to be routed from v_5 to v_7 (without any placement) and the coordinator of D_3^0 would receive a flow just requesting c_2 with ingress v_8 and egress v_{11}.

To support flows that just require routing but no processing by instances of service components, I augment all services with auxiliary ingress and egress components $c_{\text{in}}, c_{\text{eg}} \in C$. These components are added at the front and end of a service chain, respectively, and neither consume resources ($r_c(\lambda) = 0$) nor alter traversing traffic ($\lambda_c^{\text{out}}(\lambda) = \lambda$). Hence, a flow f that purely requests routing would formally require a service s_f with an empty component chain $C_{s_f} = \langle c_{\text{in}}, c_{\text{eg}} \rangle$.

Coordination on the Lowest Hierarchical Level

Coordinators on $k = 1$ are directly responsible for coordination of parts of the substrate network on $k = 0$ and therefore constitute a special case since there are no further sub-domains. To enable the same consistent coordination workflow with the same MILP used for $k > 1$, I automatically generate advertised domains $\bar{D}_i^0$ for substrate-level domains D_i^0 as follows. For each substrate node $v_j \in V_i^0$, I advertise a separate sub-domain $\bar{D}_j^{\text{sub}}$ containing v_j, an additional dummy node v_j', and intra-domain paths p_j, p_j' between the two nodes. Figure 5.5 shows how domain D_1^0 on Figure 5.1 would be advertised as $\bar{D}_1^0$ to its coordinator on $k = 1$. Intra-domain path p_j is annotated with compute capacity equal to the substrate node's capacity cap_{v_j}. Both paths p_j

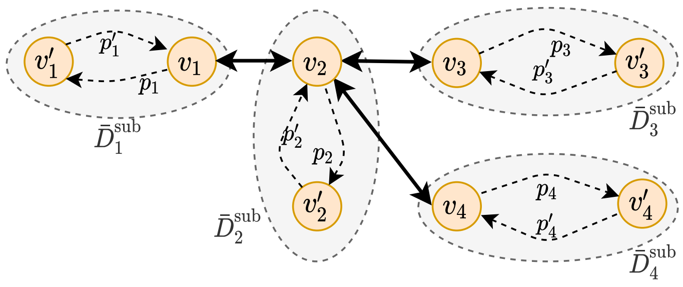


Figure 5.5.: $\bar{D}_1^0$ advertised to the coordinator on $k = 1$ in Figure 5.1. (Figure from [S18]; © 2021 IEEE.)

and p_j' have unlimited data rate and zero delay. In doing so, coordinators on $k = 1$ can scale, place, and route on these advertised paths, similar to coordinators on $k > 1$. Coordination decisions on $k = 1$ are then transparently and automatically mapped to placement solutions on the real substrate network, where any instances placed on paths p_j, p_j' are mapped to and deployed on corresponding substrate node v_j.

Following this top-down coordination approach, decisions by high-level coordinators are further refined by child coordinators. Child coordinators solve the MILP (Section 5.5) in parallel, improving response time, while the parent coordinator's decisions ensure proper coordination between domains. Finally, for actual deployment, instances are deployed in the substrate network as decided by coordinators on $k = 1$ and the solution is returned to the higher-level coordinators, which can install routing rules between the different instances (possibly spanning multiple domains).

5.5. Mixed-Integer Linear Program (MILP)

I formalize the MILP that coordinators on each level k solve for each domain D_i^k based on advertised information $\bar{D}_i^k = (\mathcal{V}_i^k, \mathcal{L}_i^k, \mathcal{P}_i^k, \Phi_i^k)$. Tables 5.1 and 5.2 summarize the notation, where Table 5.1 complements and extends the previously introduced problem parameters in Table 3.1. The scaling and placement-related variables in the first part of Table 5.2 are 0 if no instance of component c is placed at path p. Similarly, the routing-related variables in the second part of the table are only defined for components c, c' where c' is a direct successor of c in the service function chain C_{s_f} requested in f. Otherwise, the corresponding variable is 0. Compared to typical MILPs, I here need additional constraints for approximating and bounding resources and data rates and for routing based on the abstract, advertised paths $\mathcal{P}_i^k$ and corresponding compute and routing restrictions $\Phi_i^k = (\Phi_{i,k}^V, \Phi_{i,k}^L)$. The MILP formulation uses this information to ensure that node or link capacities are not exceeded, even if multiple paths overlap, such that any solution to the MILP can be refined into a valid embedding.

5.5.1. Objective

The objective in Equation 5.2 minimizes the number of placed instances, weighted by w_1, and the total delay for processing all flows, weighted by w_2. This corresponds to lower costs, e.g., for licenses or resources, and better service quality. In the evaluation, I choose a lexicographical

Table 5.1.: Additional parameters

Symbol	Definition
$0 \leq k \leq \hat{k}$	Hierarchical level k and top level $\hat{k}$
$D_i^k = (V_i^k, L_i^k)$	Domain i on hierarchical level k
$\bar{D}_i^k = (\mathcal{V}_i^k, \mathcal{L}_i^k, \mathcal{P}_i^k, \Phi_i^k)$	Advertised information about domain D_i^k
d_p	Delay of advertised path $p \in \mathcal{P}_i^k$
cap_p^L	Allowed data rate of path p
cap_p^V	Total compute capacity of path p
$\Phi_i^k = (\Phi_{i,k}^V, \Phi_{i,k}^L)$	Compute and routing restrictions for domain D_i^k
$\phi^V = (\text{cap}_\phi^V, P_\phi^V) \in \Phi_{i,k}^V$	Overlapping paths in P_ϕ^V share compute resources with capacity cap_ϕ^V
$\phi^L = (\text{cap}_\phi^L, P_\phi^L) \in \Phi_{i,k}^L$	Overlapping paths in P_ϕ^L share links with bottleneck data rate cap_ϕ^L

order, prioritizing the number of instances over total delay. Other objectives can easily be implemented by choosing suitable weights and possibly including additional decision variables from Table 5.2.

$$\min w_1 \cdot \sum_{c \in C, p \in \mathcal{P}_i^k} x_{c,p} + w_2 \cdot \sum_{f \in F} d_f^{\text{total}} \tag{5.2}$$

5.5.2. Constraints

I minimize the objective in Equation 5.2 subject to the following constraints.

Ingress Traffic and Chaining

Equation 5.3 states that the total traffic leaving ingress component c_{in} placed at path p_{in} (at the ingress) to any other instance equals the flow's data rate λ_f. Equation 5.4 ensures flow conservation between chained instances. In particular, all outgoing traffic of c' at p' corresponds to all incoming traffic modified by function $\lambda_{c'}^{\text{out}}(\lambda)$.

$$\sum_{c' \in C, p \in \mathcal{P}_i^k} \lambda_{f,c_{\text{in}},c',p_{\text{in}},p'}^{\text{total}} = \lambda_f \qquad \forall f \in F \tag{5.3}$$

$$\lambda_{c'}^{\text{out}} \left(\sum_{c \in C, p \in \mathcal{P}_i^k} \lambda_{f,c,c',p,p'}^{\text{total}} \right) = \sum_{c'' \in C, p'' \in \mathcal{P}_i^k} \lambda_{f,c',c'',p',p''}^{\text{total}}$$
$$\forall f \in F, c' \in C \setminus \{c_{\text{in}}, c_{\text{eg}}\}, p' \in \mathcal{P}_i^k \tag{5.4}$$

Table 5.2.: Decision variables

Symbol	Domain	Definition
$x_{c,p}$	$\{0,1\}$	Is an instance of component c placed on path p?
$\lambda_{c,p}$	$\mathbb{R}_{\geq 0}$	Total incoming data rate for an instance of c at p
$\lambda_{c,p}^{\text{out}}$	$\mathbb{R}_{\geq 0}$	Total outgoing data rate of an instance of c at p
$r_{c,p}$	$\mathbb{R}_{\geq 0}$	Resource requirements of an instance of c at p
r_{p,ϕ^V}	$\mathbb{R}_{\geq 0}$	Path p's portion of the shared compute resources constrained by restriction $\phi^V \in \Phi_{i,k}^V$
$y_{f,c,c',p,p',p''}^{\text{intra}}$	$\{0,1\}$	Is traffic of flow f routed via path p'' from an instance of c at path p to an instance of c' at p'?
$y_{f,c,c',p,p',l}^{\text{inter}}$	$\{0,1\}$	Is traffic of f routed via inter-domain link l from an instance of c at p to an instance of c' at p'?
$\lambda_{f,c,c',p,p',p''}^{\text{intra}}$	$\mathbb{R}_{\geq 0}$	Data rate of f that is routed via path p'' from an instance of c at p to an instance of c' at p'
$\lambda_{f,c,c',p,p',l}^{\text{inter}}$	$\mathbb{R}_{\geq 0}$	Data rate of f that is routed via inter-domain link l from an instance of c at p to an instance of c' at p'
$\lambda_{f,c,c',p,p'}^{\text{total}}$	$\mathbb{R}_{\geq 0}$	Total data rate of f from instance c at p to c' at p'
$\lambda_{f,c,c',p}$	$\mathbb{R}_{\geq 0}$	Data rate of f traversing instances of c or c' on p
$\lambda_{f,p}^{\max}$	$\mathbb{R}_{\geq 0}$	Data rate upper bound for all traffic of f on p
$\lambda_{p}^{\max}$	$\mathbb{R}_{\geq 0}$	Data rate upper bound for all traffic on path p
d_{f}^{total}	$\mathbb{R}_{\geq 0}$	Total delay for processing and routing flow f

Scaling and Placement

Whenever traffic is sent between two instances, Equations 5.5 and 5.6 ensure that both instances indeed exist and are placed accordingly. Using the Big M method, M is a large constant that ensures binary variable $x_{c,p}$ is set to 1 if any traffic traverses the instance. Since I minimize the number of instances in the objective (Equation 5.2), the solver sets $x_{c,p} = 0$ if the instance is not traversed by any traffic.

$$\lambda_{f,c,c',p,p'}^{\text{total}} \leq M \cdot x_{c,p} \qquad \forall f \in F, c, c' \in C, p, p' \in \mathcal{P}_i^k \tag{5.5}$$

$$\lambda_{f,c,c',p,p'}^{\text{total}} \leq M \cdot x_{c',p'} \qquad \forall f \in F, c, c' \in C, p, p' \in \mathcal{P}_i^k \tag{5.6}$$

Equation 5.7 sets variable $\lambda_{c',p'}$ to the total data rate of incoming traffic for an instance of c' at p'. Equations 5.8 and 5.9 set resource requirements and outgoing data rate for the instance accordingly. Equation 5.10 ensures that the shared compute resources constraint by restriction ϕ^V with capacity cap_ϕ^V are not overloaded by the overlapping paths in P_ϕ^V. Based on this, Equation 5.11 guarantees that the overall compute capacity of path p, consisting of partly exclusive and partly shared resources, is not overloaded by the total resource requirements of instances at p.

$$\lambda_{c',p'} = \sum_{f\in F, c\in C, p\in \mathcal{P}_i^k} \lambda^{\text{total}}_{f,c,c',p,p'} \qquad \forall c' \in C, p' \in \mathcal{P}_i^k \tag{5.7}$$

$$r_{c,p} = r_c(\lambda_{c,p}) \qquad \forall c \in C, p \in \mathcal{P}_i^k \tag{5.8}$$

$$\lambda^{\text{out}}_{c,p} = \lambda^{\text{out}}_c(\lambda_{c,p}) \qquad \forall c \in C, p \in \mathcal{P}_i^k \tag{5.9}$$

$$\sum_{p\in P^V_\phi} r_{p,\phi^V} \le \text{cap}^V_\phi \qquad \forall \phi^V \in \Phi^V_{i,k} \tag{5.10}$$

$$\sum_{c\in C} r_{c,p} \le \text{cap}^{\text{excl}}_p + \sum_{\phi^V \in \Phi^V_p} r_{p,\phi^V} \qquad \forall p \in \mathcal{P}_i^k \tag{5.11}$$

Routing

Equations 5.12 and 5.13 ensure that the data rate routed via intra-domain paths or inter-domain links does not exceed the total data rate between two instances.

$$\lambda^{\text{intra}}_{f,c,c',p,p',p''} \le \lambda^{\text{total}}_{f,c,c',p,p'} \qquad \forall f \in F, c, c' \in C, p, p', p'' \in \mathcal{P}_i^k \tag{5.12}$$

$$\lambda^{\text{inter}}_{f,c,c',p,p',l} \le \lambda^{\text{total}}_{f,c,c',p,p'} \qquad \forall f \in F, c, c' \in C, p, p' \in \mathcal{P}_i^k, l \in \mathcal{L}_i^k \tag{5.13}$$

Equations 5.14 and 5.15 set binary routing variables $y^{\text{intra}}_{f,c,c',p,p',p''}$ and $y^{\text{inter}}_{f,c,c',p,p',l}$ using the Big M method, similar to Equations 5.5 and 5.6.

$$\lambda^{\text{intra}}_{f,c,c',p,p',p''} \le M \cdot y^{\text{intra}}_{f,c,c',p,p',p''} \qquad \forall f \in F, c, c' \in C, p, p', p'' \in \mathcal{P}_i^k \tag{5.14}$$

$$\lambda^{\text{inter}}_{f,c,c',p,p',l} \le M \cdot y^{\text{inter}}_{f,c,c',p,p',l} \qquad \forall f \in F, c, c' \in C, p, p' \in \mathcal{P}_i^k, l \in \mathcal{L}_i^k \tag{5.15}$$

While Equation 5.4 ensures flow is conserved between instances of the whole service chain, Equation 5.16 ensures flow conservation on intermediate nodes during routing. Particularly, I consider routing via a node v from an instance of component c at path p to an instance of c' at p', where the total traffic has data rate $\lambda^{\text{total}}_{f,c,c',p,p'}$. I denote v_{src} as source node and v_{dst} as destination node as illustrated in Figure 5.6.

$$\begin{aligned} &\left(\sum_{p''\in \mathcal{P}_i^k, p'' \text{ends at } v} \lambda^{\text{intra}}_{f,c,c',p,p',p''} + \sum_{l\in \mathcal{L}_i^k, l \text{ ends at } v} \lambda^{\text{inter}}_{f,c,c',p,p',l} \right) \\ - &\left(\sum_{p''\in \mathcal{P}_i^k, p'' \text{starts at } v} \lambda^{\text{intra}}_{f,c,c',p,p',p''} + \sum_{l\in \mathcal{L}_i^k, l \text{ starts at } v} \lambda^{\text{inter}}_{f,c,c',p,p',l} \right) \\ = &\begin{cases} 0 & \text{if } v_{\text{src}} = v = v_{\text{dst}} \text{ or } v_{\text{src}} \ne v \ne v_{\text{dst}} \\ -\lambda^{\text{total}}_{f,c,c',p,p'} & \text{if } v_{\text{src}} = v \ne v_{\text{dst}} \\ \lambda^{\text{total}}_{f,c,c',p,p'} & \text{if } v_{\text{src}} \ne v = v_{\text{dst}} \end{cases} \\ &\forall v \in \mathcal{V}_i^k, f \in F, c, c' \in C, p, p' \in \mathcal{P}_i^k \end{aligned} \tag{5.16}$$

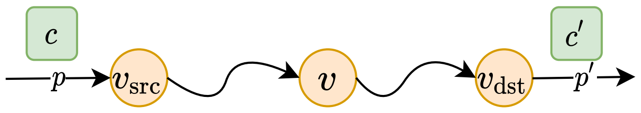


Figure 5.6.: Flow conservation for routing traffic from an instance of component c placed at path p with data rate $\lambda^{\text{total}}_{f,c,c',p,p'}$ through node $v \in \mathcal{V}^k_i$ to an instance of c' at p'. (Figure from [S18]; © 2021 IEEE.)

Link Capacities and Delay

Equation 5.17 ensures that the total traffic on inter-domain links does not exceed their capacity.

$$\sum_{f \in F, c, c' \in C, p, p' \in \mathcal{P}^k_i} \lambda^{\text{inter}}_{f,c,c',p,p',l} \leq \text{cap}_l \qquad \forall l \in \mathcal{L}^k_i \tag{5.17}$$

The corresponding restriction for intra-domain paths is more complex and thus split into multiple constraints (Equations 5.18–5.21). Similar to Equation 5.10 for compute resources, Equation 5.18 ensures that shared routing segments constrained by routing restriction ϕ^L with bottleneck data rate cap^L_ϕ are not overloaded by overlapping paths in P^L_ϕ. The compute and routing restrictions are calculated in phase 1 (Section 5.4.2) and passed as input to the MILP in phase 2 (Section 5.4.3). Equation 5.19 defines an upper bound for the total data rate on path p based on traffic routed through p (first part) and traffic being processed by instances on p, which may modify the data rate of traversing traffic (second part). The latter data rate can be bounded by considering the maximum data rate on p between any two chained components $c, c' \in C$ placed on p (Equation 5.20). In turn, this data rate is calculated in Equation 5.21 based on three overlapping parts of traffic: Traffic from an instance of c on another path p' going to c' on p, traffic within p from c, c' instances on p, traffic from c on p to an instance of c' on another path p'. These bounds slightly over-approximate the actual data rate on intra-domain path p, which depends on the refined coordination decisions from child coordinators. However, the key point is that they ensure that link capacities are not exceeded and routing decisions by parent coordinators can be refined into feasible solutions.

$$\sum_{p \in P^L_\phi} \lambda^{\max}_p \leq \text{cap}^L_\phi \qquad \forall \phi^L \in \Phi^L_{i,k} \tag{5.18}$$

$$\lambda^{\max}_p = \sum_{\substack{f \in F, c, c' \in C,\\ p', p'' \in \mathcal{P}^k_i}} \lambda^{\text{intra}}_{f,c,c',p',p'',p} + \sum_{f \in F} \lambda^{\max}_{f,p} \qquad \forall p \in \mathcal{P}^k_i \tag{5.19}$$

$$\lambda^{\max}_{f,p} = \max_{c,c' \in C} \lambda_{f,c,c',p} \qquad \forall f \in F, p \in \mathcal{P}^k_i \tag{5.20}$$

$$\lambda_{f,c,c',p} = \sum_{p \neq p' \in \mathcal{P}^k_i} \lambda^{\text{total}}_{f,c,c',p',p} + \lambda^{\text{total}}_{f,c,c',p,p} + \sum_{p \neq p' \in \mathcal{P}^k_i} \lambda^{\text{total}}_{f,c,c',p,p'}$$
$$\forall f \in F, c, c' \in C, p \in \mathcal{P}^k_i \tag{5.21}$$

Finally, Equation 5.22 calculates the total delay based on all traversed instances, intra-domain paths, and inter-domain links, which is minimized in the objective (Equation 5.2). It is also possible to bound and minimize the end-to-end delay rather than the total delay using additional variables and constraints. However, preliminary tests showed that it considerably increases complexity and yields similar results as minimizing the total delay. Hence, I focus here on minimizing the total delay rather than the end-to-end delay.

$$d_f^{\text{total}} = \sum_{\substack{c,c' \in C, \\ p,p' \in \mathcal{P}_i^k}} \left(x_{c,p} d_c + \sum_{p'' \in \mathcal{P}_i^k} y_{f,c,c',p,p',p''}^{\text{intra}} d_{p''} + \sum_{l \in \mathcal{L}_i^k} y_{f,c,c',p,p',l}^{\text{inter}} d_l \right) \forall f \in F \tag{5.22}$$

5.6. Evaluation

I evaluate the approach based on a prototype implementation with Python 3.6 and Gurobi 8 [193]. The code is publicly available on GitHub [S17] to encourage reproduction and reuse.

5.6.1. Evaluation Setup

I evaluate the approach on the real-world network topology Janos [194], which is a US network with 39 nodes and 122 links. I set link delays according to the distance and propagation delay between nodes and chose node and link capacities uniformly at random with $\text{cap}_v \in [8, 64], \text{cap}_l \in [20, 40]$. Furthermore, I consider two services s_1, s_2 consisting of a firewall and a Deep Packet Inspection (DPI) with $C_{s_1} = \langle c_{\text{FW}}, c_{\text{DPI}} \rangle$ and $C_{s_2} = \langle c_{\text{DPI}} \rangle$. I generate flows for these services based on a randomly selected subset of 10 ingress and egress nodes and data rate $\lambda_f \in [1, 5]$ GB/s chosen uniformly at random. As evaluation parameter, I increase load by increasing the number of active flows from 1 to 5.

On these scenarios, I compare the following approaches:

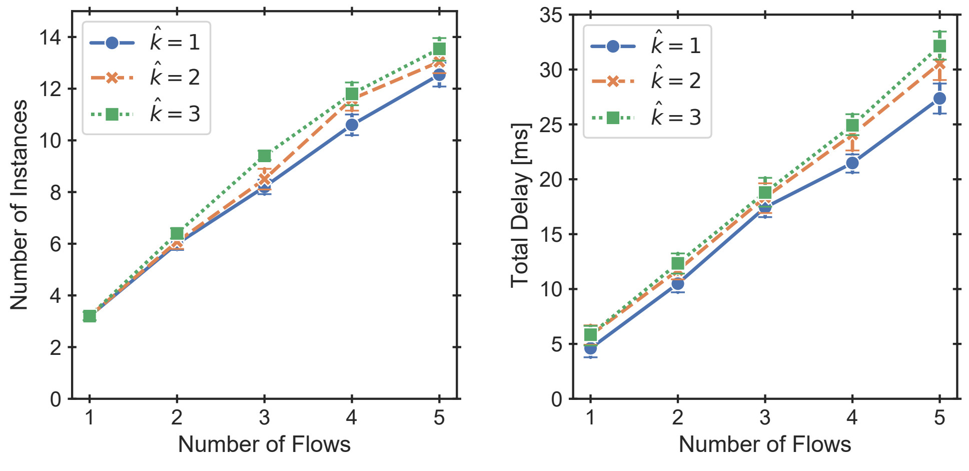
$\hat{k} = 1$: A typical flat approach with a single centralized coordinator, finding globally optimal solutions (for comparison).
$\hat{k} = 2$: The hierarchical approach with two hierarchies: One coordinator on $k = 2$ and two on $k = 1$.
$\hat{k} = 3$: The hierarchical approach with an additional hierarchy: One coordinator on $k = 3$, two on $k = 2$, and four on $k = 1$.

The centralized approach with $\hat{k} = 1$ consists of a single, flat domain containing the entire substrate network. For the hierarchical approach with $\hat{k} = 2$ and $\hat{k} = 3$, I selected domains within a hierarchical level based on node locality, i.e., Global Positioning System (GPS) coordinates, using the k-means clustering algorithm [121]. Given a desired number of domains, the k-means algorithm assigns substrate nodes to these domains such that nodes within a domain have similar (close) GPS coordinates. The top hierarchical level $\hat{k}$ always consists of a single top-level domain. For $\hat{k} = 2$, I configured two sub-domains on the lower hierarchical level. For $\hat{k} = 3$, I extended the hierarchy of $\hat{k} = 2$ with another hierarchical level below that contains four sub-domains in total.

I executed the evaluation on machines with an Intel Xeon E5-2670 CPU, allocating 8 cores at 2.6 GHz and 128 GB RAM per experiment run. The results show the mean and 95 % confidence interval over 25 independent repetitions.

5.6.2. Solution Quality

First, I compare the solution quality of the hierarchical approach ($\hat{k} = 2$ and $\hat{k} = 3$) with the centralized approach ($\hat{k} = 1$) in terms of number of placed instances and total delay, which are both minimized in the objective (Section 5.5.1). The centralized approach finds globally optimal solutions and thus constitutes a lower bound regarding both metrics.



(a) Number of instances

(b) Total delay

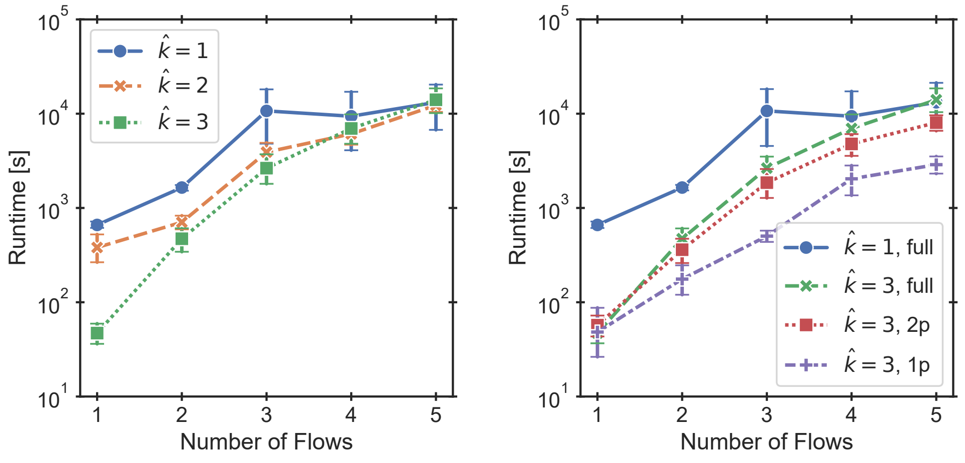
Figure 5.7.: Comparison of solution quality in terms of placed instances and total delay for increasing load. The hierarchical approach finds solutions with quality that is close to that of the optimal, centralized approach. (Figure adapted from [S18]; © 2021 IEEE.)

Figure 5.7a shows the number of placed instances with increasing load for the different approaches. The hierarchical approach finds close-to-optimal solutions with few additional instances compared to the optimal, centralized approach. As expected, the hierarchical approach with $\hat{k} = 2$ is slightly closer to the optimum (3.7 % on average) than the one with $\hat{k} = 3$ (8.1 %). This is because the latter has an additional layer of abstraction where more information is aggregated and hidden from the top-level coordinator.

Figure 5.7b shows similar results for the total delay. To improve comparability, I here set the same fixed number of instances (derived from $\hat{k} = 3$ in Figure 5.7a) for all three approaches and only minimize total delay. Hence, $\hat{k} = 1$ now has the same number of instances as $\hat{k} = 2$ and $\hat{k} = 3$ but represents the optimal lower bound regarding total delay for the given number of instances. The hierarchical approach achieves short total delay for both $\hat{k} = 2$ (within 13 % on average from optimum) and $\hat{k} = 3$ (17 %).

5.6.3. Runtime Analysis

While the solution quality of the hierarchical approach is slightly worse than the optimum, its reduced complexity and improved scalability enable much faster execution. Figure 5.8a shows the total wall-clock runtimes for each approach on a logarithmic scale when executing all coordinators within one hierarchical level in parallel. Hence, the runtime in the hierarchical approach corresponds to the maximum coordination time of any domain per hierarchical level, added up over all levels. While numerical optimization with MILPs is generally slow, the hierarchical approach is much faster than the centralized approach. The additional level of $\hat{k} = 3$ leads to even shorter runtimes than $\hat{k} = 2$ since it hides more complexity from the top-level coordinator. On average, $\hat{k} = 2$ is 88 % and $\hat{k} = 3$ is 470 % faster than $\hat{k} = 1$.



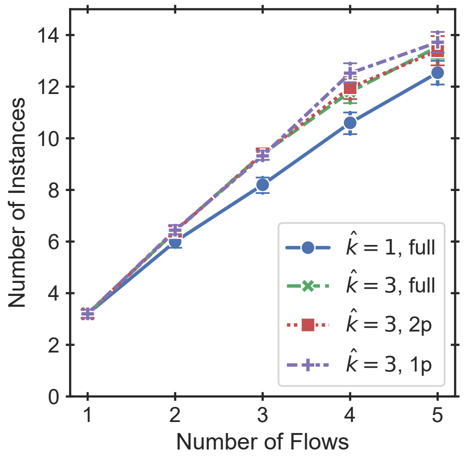
(a) All paths (b) Reduced paths

Figure 5.8.: Comparison of wall-clock runtime (logarithmic scale) with increasing load. a) The hierarchical approach is significantly faster than the centralized approach. b) Advertising fewer paths improves scalability considerably. (Figure adapted from [S18]; © 2021 IEEE.)

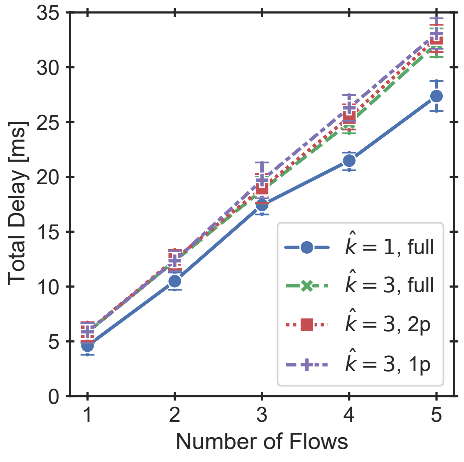
While the runtime grows with more flows for both the centralized and the hierarchical approach, the growth in runtime is stronger for the hierarchical approach than the centralized approach. With more flows, more ingress/egress nodes are included in advertised set $\mathcal{V}_i^k$ and, consequently, more intra-domain paths are advertised in $\mathcal{P}_i^k$ and corresponding restrictions in Φ_i^k. Hence, the problem input size and resulting complexity grows in two ways (flows and advertised information) for the hierarchical approach. Next, I further explore the impact of advertised information.

5.6.4. Impact of Advertised Information

In Sections 5.6.2 and 5.6.3, domains advertise all paths found by solving the maximum flow problem (Section 5.4.2), often multiple paths per source-destination pair. With more nodes $\mathcal{V}_i^k$, this drastically increases the number of advertised paths $\mathcal{P}_i^k$. In this section, I limit the number of advertised paths per source-destination pair and evaluate the impact on solution quality and runtime. In particular, I compare $\hat{k} = 3$ with all, two, or one advertised paths, denoted as full, 2p, and 1p in the figures, respectively.



(a) Number of instances



(b) Total delay

Figure 5.9.: Solution quality with varying amounts of advertised information. Advertising fewer paths only leads to minor reductions in solution quality. (Figure adapted from [S18]; © 2021 IEEE.)

Figure 5.9 shows that, even with fewer advertised paths, $\hat{k} = 3$ still finds close-to-optimal solutions. Fewer advertised paths lead to slightly lower solution quality, i.e., more instances (1.4 % on average with one vs. all paths) or higher delay (2.8 %), since coordinators lack some information. At the same time, Figure 5.8b shows that advertising fewer paths considerably reduces complexity and improves runtime. With just one advertised path per source-destination pair, $\hat{k} = 3$ is on average 3.4x faster than with full advertised paths and 10.7x faster than $\hat{k} = 1$. Moreover, runtime grows slower with more flows when advertising fewer paths, indicating better scalability.

5.7. Conclusion

The hierarchical coordination approach of this chapter mitigates the scalability issues of typical centralized approaches such as BSP in Chapter 4 but still finds close-to-optimal solutions. Instead of complete global knowledge and control of the entire network, all services, and all flows, each coordinator in the hierarchical approach only observes and controls those parts of the network, services, and flows that belong to its own domain. To control the trade-off between optimal solution quality and fast runtime, operators can adjust the number of hierarchical levels and the amount of information advertised from lower to higher levels. While this hierarchical approach greatly improves scalability compared to centralized approaches, its sophisticated 2-phase procedure and solving the MILP is still rather compute-intensive and slow. Hence, this approach is most suitable for scenarios with comparably few and long flows, where coordination decisions are required infrequently (Case I in Section 3.1.3). To complement this approach, the following Chapter 6 proposes approaches with dramatically reduced complexity and runtime for frequent coordination decisions in scenarios with many, short flows (Case II).

6. Fully Distributed Coordination

Typical centralized approaches for network and service coordination (e.g., Bidirectional Scaling and Placement Coordination Approach (BSP) in Chapter 4) require up-to-date global knowledge, which may be unavailable in large networks. In large networks with many, frequently arriving, short flows, coordination decisions are required rapidly and even hierarchical coordination (Chapter 5) may be too slow. To this end, this chapter proposes two fully distributed coordination approaches. These algorithms are designed to be simple and fast, to run individually at each node in parallel, and to require only very limited global knowledge. I evaluate and compare both algorithms against the centralized BSP approach in extensive simulations on a large, real-world network topology. My results indicate that the two distributed algorithms can compete with centralized approaches in terms of solution quality but require less global knowledge and are magnitudes faster (more than 100x). Similar to the other approaches in Part I, the approaches in this chapter are conventional as they follow predefined and manually designed heuristic algorithms.

This chapter is based on my paper [S24], which relies on concepts, code, and evaluation results created by Lars Klenner for his bachelor's thesis [141]. I proposed the initial topic and idea and advised his thesis in weekly discussions and written feedback to structure and develop the approaches. I also compiled and extended the results through additional evaluation experiments and wrote the published paper. This chapter contains verbatim copies of the following paper [S24]: Stefan Schneider, Lars Dietrich Klenner, and Holger Karl. "Every Node for Itself: Fully Distributed Service Coordination." In: *IFIP/IEEE Conference on Network and Service Management (CNSM)*. IFIP/IEEE. 2020, © 2020 IEEE. In this chapter and throughout my dissertation, I consistently write in the first-person singular form ("I" rather than "we") for ease of reading. The source code corresponding to this chapter is publicly available on GitHub [S23].

6.1. Introduction

Current approaches for network and service coordination mostly coordinate services globally, i.e., they centrally decide service scaling and placement as well as flow scheduling and routing for all flows in the entire network and rely on global knowledge [22, 9, 144, S2]. Since service demands of incoming flows and resource utilization can change frequently, up-to-date global knowledge is often not realistic and would require prohibitive overhead. In many practical scenarios with large networks and rapidly arriving or departing flows (Case II in Section 3.1.3), centralized decisions and even hierarchical coordination are too slow. Existing distributed coordination approaches typically split the network into clusters and still require central information and coordination within each cluster (Section 6.2).

To address these issues, I go one step further and propose two fully distributed approaches for online network and service coordination. These approaches can be executed individually by each node in the network, are simple and fast, and only require very limited knowledge. As there is no single coordinator that could fail or become disconnected, they are more robust to failures than existing centralized approaches. These properties make them ideal for practical large-scale networks with many flows.

One of the proposed algorithms is greedy in that nodes process incoming flows locally if possible and forward them along the shortest path to their egress nodes. The other algorithm leverages more available knowledge, e.g., utilization of neighboring nodes, to identify suitable nodes for processing incoming flows. Rather than processing all flows on the shortest path from ingress to egress, it distributes load to avoid congestion. Overall, the contributions of this chapter are:

- Section 6.4 proposes the two novel, fully distributed approaches for network and service coordination with many, short flows (Case II). The algorithms can be executed individually by each node in the network, are simple and fast, and only require limited global knowledge.
- The evaluation based on a large, real-world network topology shows that the proposed fully distributed algorithms can compete with BSP, a state-of-the-art centralized coordination approach presented in Chapter 4, but require less global knowledge and are magnitudes faster (Section 6.5).
- The code is publicly available on GitHub [S23] to encourage reproduction and reuse.

6.2. Related Work

Most related work proposes centralized approaches that require global knowledge and make global decisions, using optimization solvers and/or heuristic algorithms [182, 22, 28, 9, 144, S2, 161, 149, 106, 139]. For example, Luizelli et al. [161] propose an efficient heuristic for large-scale networks using a variable neighborhood search metaheuristic. Nevertheless, the authors still rely on centralized knowledge and coordinate services offline, i.e., requiring all problem inputs ahead of time. These limitations make their heuristic unsuitable for practical scenarios with large networks and constantly fluctuating load, which require fast coordination decisions. In contrast, my fully distributed approaches do not have these limitations and are suitable for fast, online network and service coordination.

Virtual Network Embedding (VNE) is a well researched problem related to network and service coordination [79], where authors have proposed distributed approaches [76, 115, 235, 239, 24]. Feng et al. [76] combine central coordination with a distributed mapping procedure. Houidi et al. [115] split the network into different clusters. Similar to the approaches presented in this chapter, they process multiple incoming flows in parallel. However, they rely on excessive message flooding of coordination messages to construct global state. Similarly, Shi et al. [235] split the network into clusters but aggregate network information centrally. A novel distributed VNE approach using particle swarm optimization also splits the network into clusters with separate coordination nodes [239]. Likewise, Beck et al. [24] split the network into different clusters that are coordinated in parallel by separate coordinators. The aforementioned approaches distribute and parallelize coordination into clusters, but each cluster is still coordinated centrally. In contrast, my approaches fully distribute coordination decisions to each node in the network without any centralized coordinating entity. This allows even more parallelization and faster processing of flows. It also requires less knowledge and is more robust since there is no single coordinator that could fail. Moreover, the most common VNE variants only focus on placement of already scaled and chained services but disregard scaling and online flow scheduling [79], which are an important part of network and service coordination (Chapter 3).

Fully distributed approaches have been considered in networking areas such as routing, where each router makes routing decisions locally in a distributed manner. Indeed, the approaches proposed in this chapter are related to link-state routing protocols, e.g., Open Shortest Path First (OSPF) [184], as they also use knowledge about the topology for calculating shortest paths using Dijkstra's algorithm. However, in addition to routing, I also consider joint scaling, placement, and flow scheduling. These additional interdependent tasks make the problem considerably more challenging.

6.3. Problem Statement

The approaches proposed in this chapter solve the network and service coordination problem as defined in Chapter 3. They rely mostly on local information of a node v and adjacent links L_v as detailed in Section 6.4. Such local up-to-date information is realistically available, containing currently utilized resources of node v with $r_v(t) \leq \text{cap}_v$ and of link l with $r_l(t) \leq \text{cap}_l$ at time t. Similarly, each node observes and controls incoming flows locally. For an incoming flow f at time t, $c_f(t) \in C_{s_f} \cup \{\varnothing\}$ denotes the currently requested component, indicating the flow's current processing state inside component chain C_{s_f} (cf. network service header [210]). Once the flow traversed instances of all components in the requested service, $c_f(t) = \varnothing$, which implies that the flow finished processing and needs to be forwarded to its egress node. Furthermore, m_f denotes metadata of flow f, which the proposed approaches use for coordination, e.g., to define the next target node for processing f (again, detailed in Section 6.4). Overall, the approaches in this chapter focus on fast coordination decisions for scenarios with many, short flows (Case II) but are not limited to such scenarios.

The coordination goal in this chapter is to set decision variables $x_{c,v}(t)$, $y_{f,c}(t)$, and $z_{f,v}(t)$ (Section 3.2) such that incoming flows are processed successfully and with short end-to-end delay. I formalize this high-level goal as primary objective o_f and secondary objective o_d:

$$\max o_f = \frac{|F_{\text{succ}}|}{|F_{\text{succ}}| + |F_{\text{drop}}|} \tag{6.1}$$

$$\min o_d = \frac{1}{|F_{\text{succ}}|} \sum_{f \in F_{\text{succ}}} \left(\sum_{c \in C_{s_f}} d_c + \sum_{\substack{(v,v')=l \in L,\\ t \in \{1,\ldots,T\}}} \mathbb{1}_{\{z_{f,v}(t)=v'\}} d_l \right) \tag{6.2}$$

Maximizing objective o_f means to successfully process as many flows (F_{succ}) as possible, avoiding dropped flows (F_{drop}), thus maximizing the percentage of successful flows (Equation 6.1). To avoid dropping flows due to lack of available node or link capacities, load should be balanced across multiple nodes and links according to their capacities. The secondary goal is to minimize o_d, which is the end-to-end delay averaged over all successful flows (Equation 6.2). The end-to-end delay of a flow f consists of two parts. First, the fixed service-specific sum of processing delays d_c of components c whose instance f traversed. And second, the sum of link delays d_l that f experienced during routing.

Note that the two objectives o_f and o_d may be conflicting. For example, distributing flows over more nodes and links to balance the load helps with processing more flows successfully (improves o_f) but also leads to longer paths and higher end-to-end delays (degrades o_d). In this chapter, I approach this trade-off by optimizing o_f and o_d in lexicographical order, i.e., prioritizing o_f but still trying to optimize o_d as far as possible.

6.4. Fully Distributed Coordination Approaches

To solve the network and service coordination problem of Section 6.3, I propose two fully distributed algorithms. In both cases, the algorithm is executed independently in parallel on each node in the network, where nodes directly decide how to treat incoming flows. Both algorithms require knowledge of the network topology including link delays, which I assume to be rather static and globally available. Neither algorithm, however, relies on full, up-to-date, global knowledge of fast changing information such as the current utilization of all nodes or links in the network. Instead, Greedy Coordination with Adaptive Shortest Paths (GCASP) (Section 6.4.1) does not require any further global information and Score-Based Coordination (SBC) (Section 6.4.2) can be configured to leverage any available useful information. In practice, these approaches could be deployed using mechanisms for local control such as dynamic flow rules in Software-Defined Networking (SDN) [257].

6.4.1. Greedy Coordination with Adaptive Shortest Paths

The main design goals for GCASP were to be simple, effective, fast, and frugal, in that it works without any global up-to-date information about node or link utilization. Instead, each node only knows the utilization of its own compute resources and outgoing links. Nodes process flows greedily and forward them along the shortest path to their egress nodes, trying to minimizing end-to-end delay. To avoid dropping flows due to congestion, GCASP adjusts routes when a link is congested, i.e., when sending a flow via the link would violate its capacity cap_l. As GCASP only requires local knowledge and makes local decisions, it does not require any information exchange between nodes. Information about a flow's current destination and path is directly included in its metadata such that no additional communication overhead is necessary.

Algorithm

Algorithm 3 shows the GCASP algorithm, which is run by a node v once a flow f starts arriving (passed as arguments in line 1). First, v locally processes as many components of f as possible (lines 2–4). Specifically, v processes f at an (existing or newly started) instance of requested component c_f if there are enough resources available. Here, $r_v(t|x_{c_f,v}(t) = 1 \wedge y_{f,c_f}(t) = v)$ denotes the total resources required at node v and time t if f were to be processed there, which can be calculated locally based on function $r_c(\lambda)$. After processing, c_f points to the next requested component in C_{s_f}. Again, v processes f locally at an instance of c_f if it has sufficient resources. This local, greedy processing continues until either v's compute resources are fully utilized or after f traversed all requested service components ($c_f = \varnothing$).

Algorithm 3 GCASP Algorithm

```
1: procedure GCASP($v, f$)
2:     while $r_v(t|x_{c_f,v}(t) = 1 \wedge y_{f,c_f}(t) = v) \leq \text{cap}_v$ and $c_f \neq \varnothing$ do
3:         $x_{c_f,v}(t) \leftarrow 1$
4:         $y_{f,c_f}(t) \leftarrow v$
5:     if $v = v_f^{\text{in}}$ then
6:         $m_f.\text{dst} \leftarrow v_f^{\text{eg}}$
7:         $m_f.\text{path} \leftarrow \texttt{shortest\_path}(v, m_f.\text{dst}, L)$
8:     if $v = m_f.\text{dest}$ and $c_f \neq \varnothing$ then
9:         $m_f.\text{dst} \leftarrow \texttt{random\_choice}(V)$
10:        $m_f.\text{path} \leftarrow \texttt{shortest\_path}(v, m_f.\text{dst}, L)$
11:    if $c_f = \varnothing$ and $m_f.\text{dst} \neq v_f^{\text{eg}}$ then
12:        $m_f.\text{dst} \leftarrow v_f^{\text{eg}}$
13:        $m_f.\text{path} \leftarrow \texttt{shortest\_path}(v, m_f.\text{dst}, L)$
14:    FORWARD_FLOW($v, f, L$)
```

Algorithm 4 Flow Forwarding Using Adaptive Shortest Paths

```
1: procedure FORWARD_FLOW($v, f, L$)
2:     $v' \leftarrow m_f.\text{path.pop}()$
3:     if $r_{(v,v')}(t) + \lambda_f \leq \text{cap}_{(v,v')}$ then
4:         $z_{f,v}(t) \leftarrow v'$
5:     else
6:         $L_{\text{free}} \leftarrow L \setminus \{l \in L_v | r_l(t) + \lambda_f > \text{cap}_l\}$
7:         $m_f.\text{path} \leftarrow \texttt{shortest\_path}(v, m_f.\text{dst}, L_{\text{free}})$
8:         $z_{f,v}(t) \leftarrow m_f.\text{path.pop}()$
```

In the former case, f continues processing in a similar fashion on the next node. In the latter case, f finished processing successfully and leaves the network once it reaches its egress v_f^{eg}. In both cases, v tries to forward f to neighbor v' on the shortest path to f's egress node. If v is f's ingress node, it calculates the shortest path to v_f^{eg} based on the static topology and weighted by link delays in L, e.g., using Dijkstra's algorithm [57] (lines 5–7). The destination and calculated path are saved in metadata m_f, which is used when forwarding the flow (line 14).

Forwarding is done in Algorithm 4, which retrieves the next node v' on the computed path (line 2 in Algorithm 4). If the link to v' still has enough available data rate, v sends f to v' (lines 3–4). Otherwise, v recomputes the shortest path without using any of its currently congested outgoing links and sends f along the adapted path (lines 5–8). Note that just excluding all congested links up front (at $t = 0$) does not suffice since link utilization changes dynamically over time according to flow arrival and the algorithm's decisions. In case all outgoing paths are congested, I assume that v cannot buffer the entire flow f and thus drops it.

If a flow arrives at its egress before being fully processed, it is sent to and processed at surrounding nodes and returns to the egress once it finished processing completely. To balance the load among the surrounding nodes, GCASP randomly chooses a new, temporary destination node (cf. Valiant Load Balancing [251]), computes the shortest path, and sends f towards the new destination (lines 8–10 in Algorithm 3). As before, nodes on the path to the new destination process f greedily. Once f is fully processed, it is no longer sent towards its temporary destination but is immediately rerouted to its egress node (lines 11–13). In doing so, remaining processing is done close to the egress node as far as possible, keeping the path delay low. If nodes surrounding the egress are already fully utilized, f is sent farther away towards the temporary destination until a node with free capacity is found. If f reaches the temporary destination and still needs more processing, the same procedure is repeated and a new, random destination is chosen temporarily. At this point, the previous temporary destination becomes irrelevant as f is forwarded directly to its original egress node v_f^{eg} upon complete processing. In line with the design goal of GCASP to be frugal, this load balancing scheme only requires the current utilization of local node v and knowledge of the static network topology to choose a temporary destination and compute the path. If the current utilization of neighboring nodes V_v is also available, GCASP could easily be adjusted to favor neighbors with low utilization for processing f (comparable to SBC in Section 6.4.2).

Complexity

The most common coordination decisions concern processing an incoming flow (lines 2–4 in Algorithm 3) and then forwarding it along its path (lines 2–4 in Algorithm 4). Both operations only take constant time, i.e., $O(1)$. Computation of the initial or adapted shortest path is more expensive but happens much more rarely. Using Dijkstra's algorithm, it takes $O(|V|^2)$ or $O(|L| + |V| \log |V|)$, depending on the implementation. Selecting a (new) temporary destination node also happens rather rarely and is dominated by the time complexity for computing the shortest path towards the destination.

GCASP only relies on the static network topology for computing shortest paths and choosing temporary destination nodes, leading to a space complexity of $O(|V| + |L|)$. Except for shortest path calculation, all other decisions only take $O(1)$ space, e.g., based on a node's local resource utilization or a flow's attributes. The flow metadata m_f holds the assigned destination and computed path by GCASP. The latter may grow with larger networks and higher congestion but could be limited in size to limit overhead. In huge networks, where the full shortest path exceeds this size limit, GCASP could just save the next hops of the shortest path that fit into the size limit and then recompute the remaining hops on demand. This trade-off between maximum size of m_f and computational overhead for recomputing paths is mostly relevant in very large networks and could be further investigated in future work.

Example

Figure 6.1 illustrates GCASP's adaptive routing for a flow f. In step 1, f is sent along the shortest path from its ingress towards its egress. On the path, intermediate nodes (omitted in the figure) process f greedily. At v_1, the outgoing link on the shortest path is congested and the path is recomputed (step 2). In this example, f reaches its egress without being fully processed. Hence, GCASP randomly selects v_3 as new, temporary destination and sends f towards v_3 (step 3). Nodes on the path continue to process f greedily such that f is fully processed when reaching intermediate node v_2, which is still close to the egress. At that point, f is immediately rerouted to its egress node and leaves the network successfully (step 4).

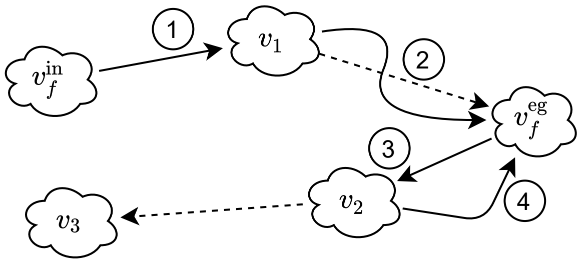


Figure 6.1.: Example illustration of GCASP's adaptive routing. (Figure from [S24]; © 2020 IEEE.)

6.4.2. Score-Based Coordination with Adaptive Shortest Paths

While full, up-to-date global knowledge may be unrealistic, it is likely that some information is available through monitoring or beaconing. I designed SBC to leverage such available information for its coordination decisions. In particular, SBC calculates node scores based on the available information that indicate each node's suitability to process a given flow. Rather than sending all flows to their egress nodes along the same shortest path, SBC tries to distribute flows among different paths and actively selects promising nodes for processing. SBC's score calculation can be configured according to metrics that are being monitored by a system like Prometheus [46]. In contrast to centralized approaches, which require global knowledge, SBC also works with limited information, e.g., just from neighboring nodes. Hence, the amount of collected and exchanged information to be used by SBC can be adjusted freely.

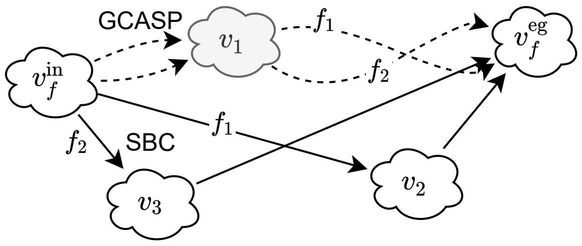


Figure 6.2.: Rather than sending all flows along the same shortest path, SBC avoids congestion by distributing flows across different nodes based on their calculated score. (Figure from [S24]; © 2020 IEEE.)

Motivating Example

Figure 6.2 illustrates the benefit of SBC over GCASP in an example. GCASP greedily sends all flows with the same ingress and egress node (here, f_1, f_2) along the same shortest path (dashed lines). All flows compete for node and link resources on the path which may lead to congestion, overloaded nodes (e.g., v_1), and dropped flows. Instead, SBC can use available monitoring metrics, e.g., current node utilization or number of dropped flows per node. For each node, SBC calculates a score based on the weighted metrics and selects the most suitable destination nodes (with the highest score) for processing each flow. Here, nodes v_2, v_3 receive the highest score for f_1, f_2, respectively. Processing the flows there avoids congestion and dropped flows.

Algorithm

Algorithm 5 SBC Algorithm

```
1:  procedure SBC($v, f$)
2:      if $v = v_f^{\text{in}}$ then
3:          SET_DEST($v, f, V, L$)
4:      if $v = m_f$.dst then
5:          if $r_v(t | x_{c_f,v}(t) = 1 \land y_{f,c_f}(t) = v) \leq \text{cap}_v$ and $c_f \neq \varnothing$ then
6:              $x_{c_f,v}(t) \leftarrow 1$
7:              $y_{f,c_f}(t) \leftarrow v$
8:          SET_DEST($v, f, V, L$)
9:      FORWARD_FLOW($v, f, L$)
10: procedure SET_DEST($v, f, V, L$)
11:     if $c_f = \varnothing$ then
12:         $m_f$.dst $\leftarrow v_f^{\text{eg}}$
13:     else
14:         $V_{v,f} \leftarrow$ candidates($v, f, V, L$)
15:         for all $v' \in V_{v,f}$ do
16:             $g(v') \leftarrow \sum_{i \in [1,|\mathcal{A}|]} w_i a_i(v')$
17:         $m_f$.dst $\leftarrow \arg\max_{v'} [g(v')]$
18:     $m_f$.path $\leftarrow$ shortest_path($v, m_f$.dst$, L$)
```

Algorithm 5 shows the SBC algorithm, which is called when a new flow f starts arriving at a node v. If v is f's ingress node, v computes the first destination node for processing f (lines 2–3). The calculation of the destination node based on available information is the core procedure of SBC (lines 10–18). If f is already fully processed ($c_f = \varnothing$), its egress node is set as destination (lines 11–12). Once f reaches its egress fully processed, it successfully leaves the network. Otherwise, SBC chooses a new suitable destination node from a set of candidate nodes based on their calculated score (lines 15–16). Specifically, for each candidate node v', it considers a set $\mathcal{A} = \{a_1, ..., a_k\}$ of k weighted (by w_i) node attributes a_i to calculate an overall node score $g(v')$ (lines 15–16). Attributes $\mathcal{A}$ and candidate nodes $V_{v,f}$ can be selected depending on the available information in the network, e.g., $V_{v,f}$ could be all nodes V or just neighbors V_v (line 14). Attributes a_i should be scaled to $[0, 1]$ for comparability and reflect available metrics of interest that help asses a node's suitability for processing. Examples are the shortest path delay to the egress via v', the current utilization of v', the amount of successfully processed flows at v', etc. The configuration used for evaluation is described in Section 6.5.1. Ultimately, the candidate v' with the highest score $g(v')$ is chosen as next destination and the shortest path is calculated, e.g., using Dijkstra's algorithm [57] (lines 17–18). Similar to GCASP, SBC then forwards

f along the computed path (line 9) using Algorithm 4. Rather than processing f greedily on the path, f is processed at its selected destination node if it has enough available resources (lines 4–7). Afterwards, a new destination is selected (line 8).

Complexity

Like GCASP, SBC makes fast forwarding and processing decisions in $O(1)$. Only adapting the shortest paths due to congestion and setting a new destination require more time. The latter requires $O(|V_{v,f}|\phi)$ where ϕ is the time complexity of calculating $g(v)$, which depends on the complexity of computing the chosen node attributes. If all attributes are simple measurements that can be retrieved in $O(1)$, it results in $\phi = O(k) = O(1)$ and complexity $O(|V_{v,f}|)$ for setting a new destination. Calculation of the shortest path can be done in $O(|V|^2)$ or $O(|L| + |V| \log |V|)$. In addition to the static network topology, SBC also relies on k attributes and the computed score for all candidate nodes $V_{v,f}$. Hence, SBC's space complexity is $O(|V| + |L| + k|V_{v,f}|) = O(|V| + |L| + |V_{v,f}|)$.

6.5. Evaluation

6.5.1. Evaluation Setup

I evaluate the two proposed algorithms, GCASP and SBC, using extensive simulations on the real-world DFN network topology [142] with 58 nodes and 87 links, which I consider to be a representative example of medium to large networks in practice. Link capacities are $\text{cap}_l = 50 \forall l \in L$ and node capacities are either homogeneous or heterogeneous, depending on the evaluation scenario. In the homogeneous case, all nodes have capacity $\text{cap}_v = 10$. In the heterogeneous case, $\text{cap}_v \in \{0, 10, 50\}$ is picked in a random, weighted manner such that the average node capacity is similar to the homogeneous case.

Furthermore, I consider three services $s_1 = \langle c_{\text{IDS}}, c_{\text{proxy}}, c_{\text{web}} \rangle$, $s_2 = \langle c_{\text{proxy}}, c_{\text{IDS}}, c_{\text{web}} \rangle$, and $s_3 = \langle c_{\text{FW}}, c_{\text{DPI}} \rangle$. Instances of each service component require resources linear in the currently processed load. Furthermore, they incur a per-flow processing delay that is normally distributed with $\mathcal{N}(5\,\text{ms}, 1\,\text{ms})$, where values are cut off at $0\,\text{ms}$ to prevent negative delays. Flows arriving at the network's ingress nodes request one of the three services chosen uniformly at random. For each ingress, flow inter-arrival times vary randomly between 1, 2, 5, and 10 time steps, flow duration $\delta_f \in \{1, 2\}$, and flow data rate $\lambda_f \in \{1, 2, 4, 6\}$, representing scenarios with many, short flows (Case II in Section 3.1.3). The duration per experiment is $T = 1000$ time steps.

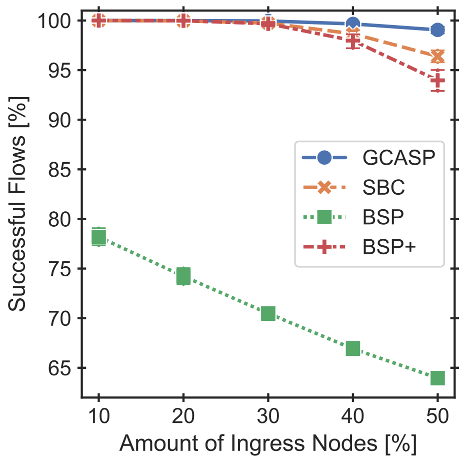
SBC's configurability allows numerous different variants of candidate node selection and attributes $\mathcal{A}$. For the variant used in the evaluation, the score of each candidate node v' is calculated based on $k = 3$ equally weighted ($w_i = 1$) attributes:

1. The shortest path delay from current node v to v' and from there to the egress.
2. The number of dropped flows at v' so far.
3. The total data rate of flows currently being processed or forwarded at v'.

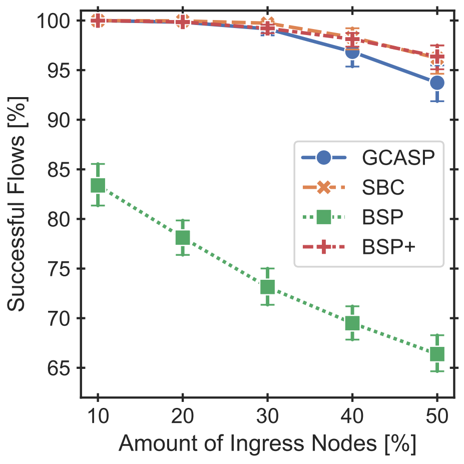
Candidate nodes $V_{v,f}$ are all nodes other than current node v with sufficient remaining resources for processing f. This requires frequently recomputing set $V_{s,f}$ according to the current node utilization and requirements of flow f. Consequently, SBC is indeed significantly slower than GCASP but still much faster than a typical centralized approach (Section 6.5.4).

I compare GCASP and SBC with the centralized coordination algorithm BSP from Chapter 4. I call BSP once for each new flow entering the network to compute where to process and how to route the flow. In doing so, BSP requires global knowledge of the current node and link utilization. Even with up-to-date global knowledge at the time of flow arrival, BSP may choose a currently free node for processing that is later fully utilized by other flows when the scheduled flow arrives there. Hence, flows may be dropped when they arrive for processing at an over-utilized node. Thus, I also consider a variant, BSP+, where I call BSP again to recompute flow processing and routing when a flow cannot be processed at an overloaded node.

All experiments were repeated 50 times with different random seeds. Figures show the mean and 95 % confidence interval of these repetitions. For running the evaluation, I used machines with Intel Xeon W-2145 CPU and 32 GB RAM. For reproducibility, I publish the code of both proposed fully distributed algorithms, the used simulator, and all evaluation results on GitHub [S23]. Similarly, BSP is publicly available [S12].



(a) Homogeneous node capacity



(b) Heterogeneous node capacity

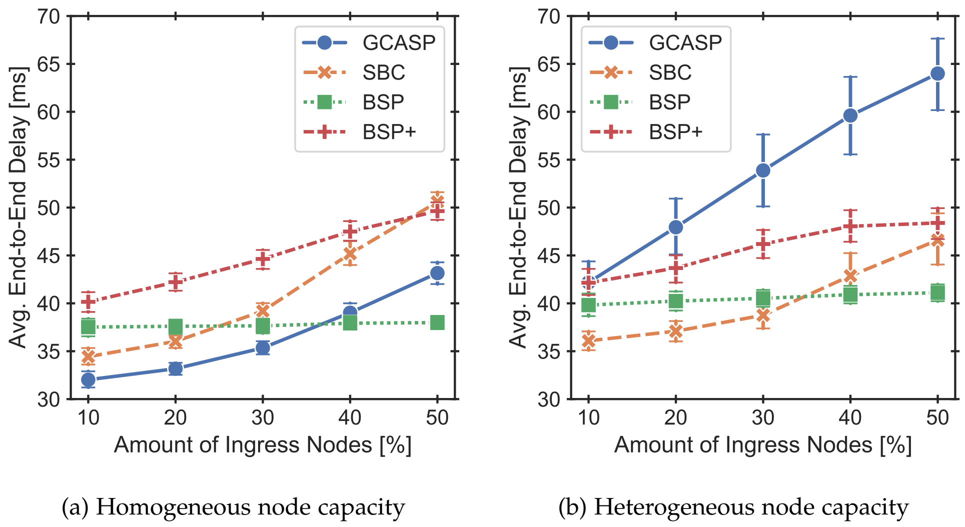
Figure 6.3.: The fully distributed coordination approaches process similar or even more flows successfully than the centralized BSP+ approach. GCASP is better with homogeneous and SBC with heterogeneous node capacities. (Figure from [S24]; © 2020 IEEE.)

6.5.2. Solution Quality

First, I compare the achieved solution quality of the proposed fully distributed coordination algorithms (GCASP and SBC) and the centralized approach (BSP and BSP+). As evaluation parameter, I vary the ingress node percentage, i.e., the percentage of nodes in the network at which flows arrive. With an increasing ingress node percentage, more nodes are randomly selected as ingress nodes, leading to increasing overall load. The percentage of egress nodes is fixed to 30 %, to which flows are assigned uniformly at random. I also vary between homogeneous vs. heterogeneous node capacities. As metrics to evaluate the coordination quality, I consider the percentage of successfully processed flows o_f and their average end-to-end delay o_d at the end of each experiment as defined in Section 6.3. I also consider resource utilization of both nodes and links since coordination algorithms should use resources economically.

Successful Flows

Figure 6.3 shows the percentage of successful flows achieved by the different algorithms. The percentage of successful flows decreases with increasing load as the network becomes more congested and some flows cannot be processed or forwarded. BSP drops comparatively many flows as it only makes coordination decisions once per flow when flows enter the network, based on information that may be outdated soon after. This is due to the considered scenario with many, rapidly arriving, sequential, and partially overlapping flows (Case II), where information such as resource utilization changes very frequently. In contrast, typical centralized coordination approaches, including BSP, are designed to focus mainly on few, long-running flows (Case I). BSP+, which recalculates flow processing and routing whenever a flow would otherwise be dropped due to lack of node capacities, performs much better and achieves close to 100 % successful flows.



(a) Homogeneous node capacity (b) Heterogeneous node capacity

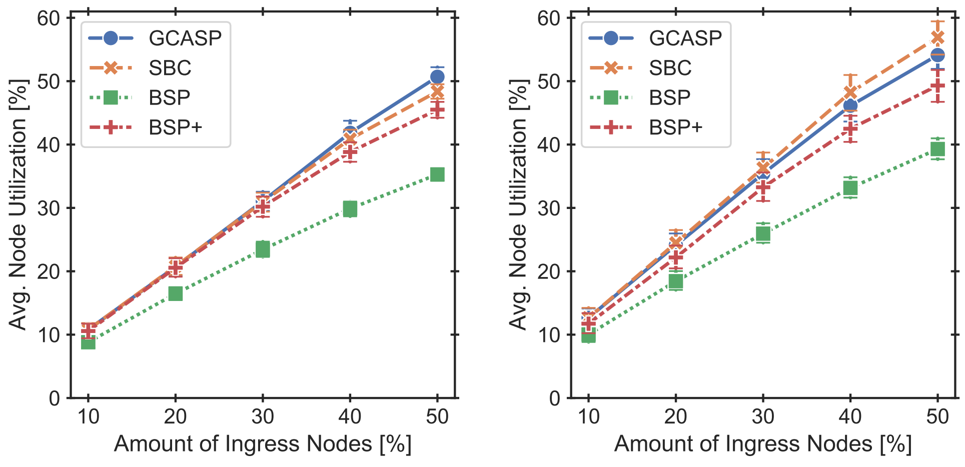
Figure 6.4.: While average end-to-end delay increases with increasing load, the fully distributed coordination approaches can compete with or even outperform centralized BSP and BSP+. (Figure from [S24]; © 2020 IEEE.)

Nevertheless, the two fully distributed algorithms process a similar number or even more flows successfully than BSP+. Clearly, the many fast and individual per-node decisions of these fully distributed algorithms work well and process close to 100 % of flows successfully. In the network with homogeneous node capacities (Figure 6.3a), GCASP even outperforms all other algorithms, even though it uses no global knowledge except for the static network topology.

This is because, with homogeneous node capacities, there are likely enough compute capacities on the shortest path from ingress to egress and around the egress node. In contrast, in the case of heterogeneous node capacities (Figure 6.3b), SBC is slightly better than GCASP and on par with BSP+ because it leverages available knowledge to actively select nodes with sufficient processing capacities. Still, GCASP processes more than 90 % of flows successfully even under high load.

End-to-End Delay

Figure 6.4 shows the average end-to-end delay of successfully processed flows. With heterogeneous capacities, the approaches have to route along longer paths to reach nodes with sufficient resources, generally leading to higher delays than in the homogeneous case. While BSP drops an increasing percentage of flows with increasing load, it ensures relatively low and constant end-to-end delay for the remaining successful flows. For the other algorithms, average end-to-end delay increases with increasing load as more flows are rerouted due to congestion, leading to longer paths. Again, the fully distributed algorithms can compete with and even outperform the centralized BSP and BSP+ approaches. The figure also confirms that the greedy GCASP is best in the homogeneous case (Figure 6.4a). In the heterogeneous case (Figure 6.4b), SBC is better again and achieves lower end-to-end delay.

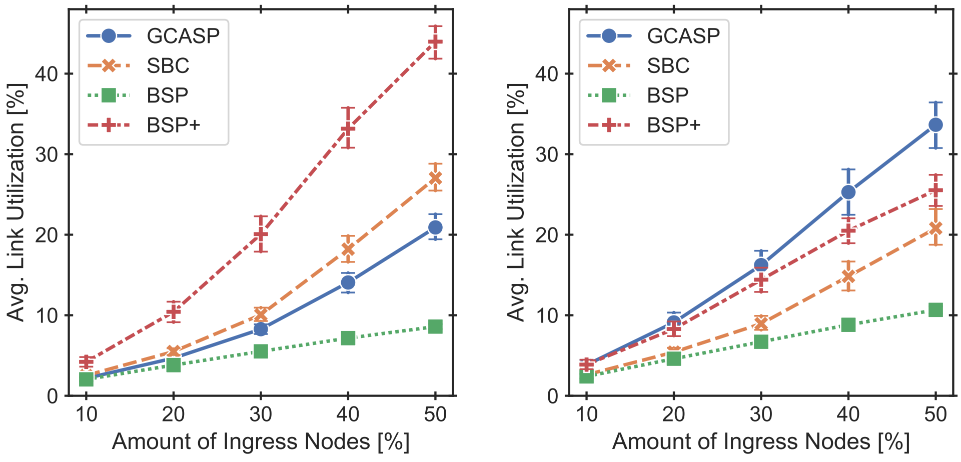


(a) Homogeneous node capacity (b) Heterogeneous node capacity

Figure 6.5.: The fully distributed coordination approaches utilize node resources comparably to centralized BSP+. (Figure from [S24]; © 2020 IEEE.)

Resource Utilization

Figure 6.5 shows the node resource utilization averaged over all nodes. The node utilization correlates with the percentage of successful flows in Figure 6.3 as processing more flows completely (i.e., successfully) requires more node resources. Accordingly, BSP requires less node



(a) Homogeneous node capacity (b) Heterogeneous node capacity

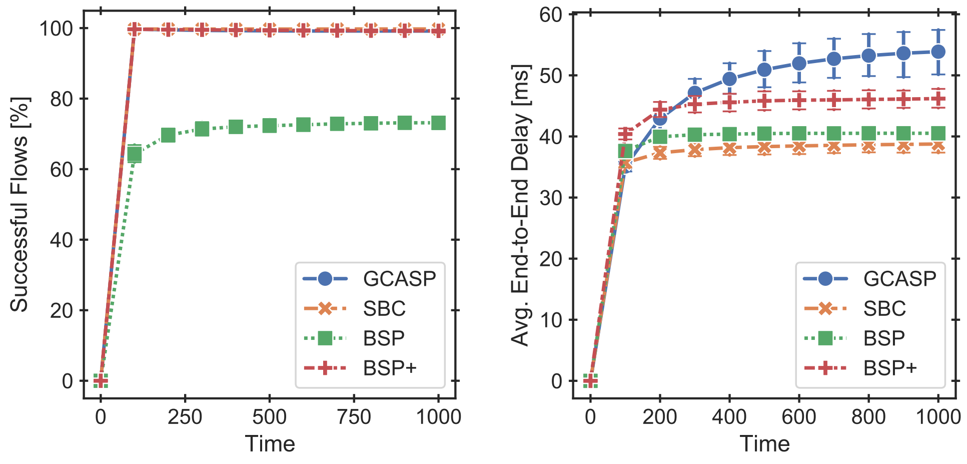
Figure 6.6.: The fully distributed coordination approaches utilize similar or less link resources than BSP+. Again, GCASP works best with homogeneous node capacities, whereas SBC excels in heterogeneous scenarios. (Figure from [S24]; © 2020 IEEE.)

resources than the other algorithms because it drops more flows. The two distributed algorithms utilize similar percentages of node resources as the centralized BSP+ approach.

Figure 6.6 shows the link resource utilization averaged over all links. Again, by dropping more flows, BSP reduces the overall load in the network and thus also the link utilization compared to the other algorithms. More interestingly, the link utilization also reflects how efficiently the algorithms route traffic. Longer detours lead to more traversed links and higher link utilization. For homogeneous traffic (Figure 6.6a), both distributed algorithms utilize less than 30 % link resources and far less than BSP+, which reroutes traffic whenever flows could not be processed. As discussed before, GCASP performs worse for heterogeneous traffic (Figure 6.6b) and routes flows along detours to complete processing. Nevertheless, the link utilization of GCASP is not much higher than of BSP+. SBC uses available information to route traffic more effectively and outperforms BSP+ in terms of link utilization.

6.5.3. Coordination Stability

The results in Section 6.5.2 show the quality metrics at the end of each experiment, i.e., after $T = 1000$ time steps. In this section, I evaluate the coordination stability over these 1000 time steps during which flows keep arriving randomly as described in Section 6.5.1. I consider the DFN network with heterogeneous node capacities and 30 % ingress nodes. Figure 6.7 shows that both the percentage of successful flows (Figure 6.7a) and the average end-to-end delay (Figure 6.7b) are very stable over time for all algorithms. The initial jump from $t = 0$ to $t = 100$ is simply because all metrics are initialized to 0. While GCASP performs worse on heterogeneous than on homogeneous node capacities, its achieved end-to-end delay still only increases slightly in the beginning but then converges and stabilizes.



(a) Successful flows (b) Average end-to-end delay

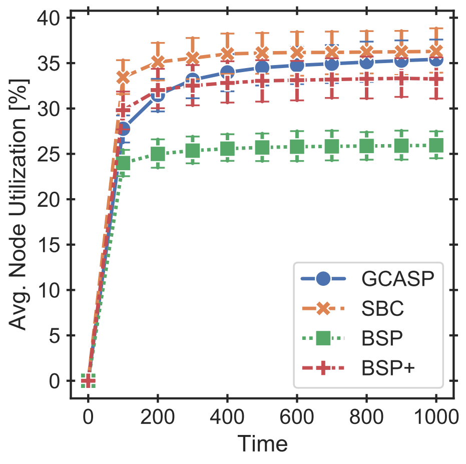
Figure 6.7.: The fully distributed coordination approaches are stable in terms successful flows and end-to-end delay, similar to centralized approaches. (Figure from [S24]; © 2020 IEEE.)

Figure 6.8 shows the average node and link utilization over time. Again, the utilization is fairly stable for all algorithms. Only for BSP+ the link utilization slowly grows over time as more and more flows are being rerouted due to congestion. Overall, the results indicate that the two distributed algorithms not only achieve competitive solution quality but also converge early and maintain stable coordination over time.

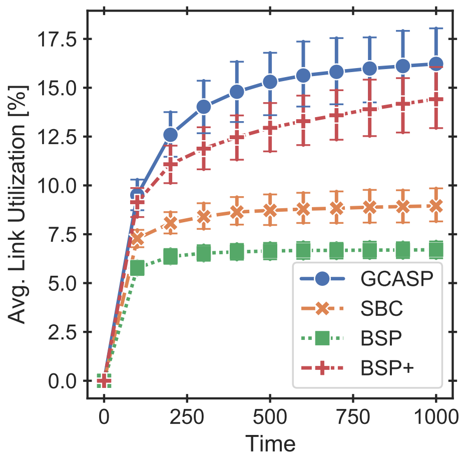
6.5.4. Scalability

In addition to ensuring high and stable service quality, coordination should be fast and scalable to handle rapidly arriving flows in practical scenarios. To evaluate coordination scalability, I compare the number of coordination decisions (i.e., algorithm calls) and algorithm runtime of the different approaches. Again, I consider the DFN network with heterogeneous node capacities and 30 % ingress nodes. Figure 6.9 shows the algorithms' number of coordination decisions on a logarithmic scale. The proposed fully distributed algorithms (GCASP and SBC) coordinate each incoming flow individually at each node. In contrast, BSP and BSP+ are only called once when a new flow enters the network and, in case of BSP+, when a flow cannot be processed due to lack of node resources. Hence, the fully distributed algorithms make far more coordination decisions in total and per flow than the centralized approaches (Figure 6.9a). I assume that BSP and BSP+ make all decisions at a single centralized node (at least logically). On the other hand, GCASP and SBC distribute coordination over all nodes in the network such that they still make significantly fewer coordination decisions *per node*.

The runtime for these coordination decisions is even more important than the number of decisions. Due to their simplicity, the runtime (per flow, per node, and in total) of the fully distributed algorithms is much lower than of BSP and BSP+ (Figure 6.9b). While I do not consider



(a) Average node utilization



(b) Average link utilization

Figure 6.8.: The fully distributed coordination approaches converge quickly to a solution with stable resource utilization. (Figure from [S24]; © 2020 IEEE.)

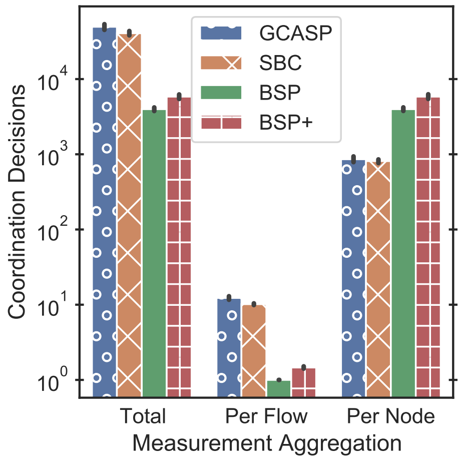
this coordination time as part of a flow's end-to-end delay in Sections 6.5.2 and 6.5.3, it would still impact overall delay and service quality in practice.

I also tested GCASP and SBC successfully on the large GTS CE network [142] with 149 nodes and 193 links. Compared to the DFN network, the algorithms' total decisions and runtimes were higher, but the numbers per flow and per node were comparable to the ones in the DFN network. Due to the prohibitive long runtimes of BSP and BSP+, I do not present a full performance evaluation on GTS CE. Nevertheless, the results on the two networks indicate that the proposed fully distributed algorithms scale well to practical, large-scale networks.

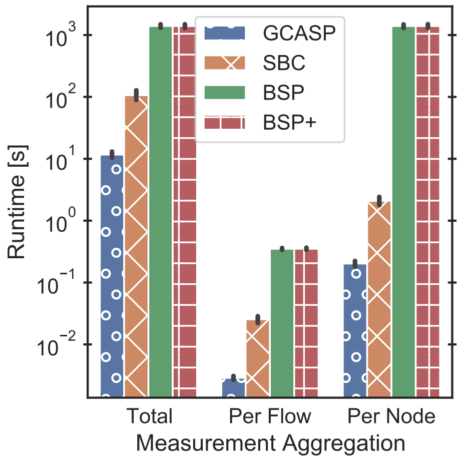
6.6. Conclusion

The proposed fully distributed algorithms coordinate services well and achieve service quality comparable to centralized coordination algorithms. At the same time, they require less or no global network information, are faster, can be massively parallelized, and are more robust to failures. These attributes make them ideal for fast-paced coordination in large networks with many, short, and rapidly arriving flows (Case II in Section 3.1.3).

In such scenarios (Case II), centralized approaches like BSP in Chapter 4 are unsuitable since they rely on global network information, which is quickly outdated or even completely unavailable due to fast fluctuations. Coordinating all incoming flows centrally also becomes too slow or even intractable for many, short flows, particularly in large networks. In contrast, centralized approaches like BSP are well suited for scenarios with fewer, longer flows, which arrive and depart infrequently and where the network is stable in between (Case I). In scenarios with few and long flows, global information is realistically available and remains up-to-date for longer



(a) Coordination decisions



(b) Coordination runtime

Figure 6.9.: The fully distributed coordination approaches only make few decisions per node and are magnitudes faster than BSP and BSP+ (logarithmic scale). (Figure from [S24]; © 2020 IEEE.)

time. Here, centralized decisions are feasible and the overhead of more complex coordination algorithms is justified to reach better coordination results as flows stay in the network longer.

In comparison, the hierarchical coordination approach in Chapter 5 represents an intermediate approach. It requires less global information, decides faster, and scales to larger networks than typical centralized coordination approaches. Yet, the sophisticated two-phase procedure still requires substantial computational effort and may be too slow in scenarios where flows arrive rapidly (Case II). Instead, it is ideal for intermediate cases with several medium to long flows in large networks, where centralized approaches are slow and fully distributed coordination approaches sacrifice solution quality.

In future work, these different techniques could be combined into a hybrid coordination approach that classifies incoming flows (e.g., elephant and mice flows [42]) and dynamically selects a suitable coordination technique. Important coordination decisions for long-lasting flows could be highly optimized by a central coordination approach or, in large networks with more flows, by a hierarchical coordination approach. Fast, short flows could be coordinated quickly with low overhead in a fully distributed manner. The fully distributed coordination approaches could also be used as fallback mechanism if the centralized or hierarchical coordination becomes too slow, e.g., reaching a timeout before making a coordination decision.

Overall, the conventional approaches presented in Part I enable efficient and automated network and service coordination for high service quality with low costs. In principle, the approaches can be applied to scenarios with any network, services, and traffic as described in Chapter 3. Still, they follow hard-wired algorithms or solve predefined Mixed-Integer Linear Programs (MILPs) designed by human experts with certain built-in assumptions, e.g., known resource utilization $r_c(\lambda)$. Part II complements these conventional approaches with novel coordination approaches using machine learning, which rely less on human expertise and instead leverage available data to *learn* network and service coordination and *adapt* to any given scenario.

Part II.

Machine Learning Coordination Approaches

7. Machine Learning for Dynamic Resource Allocation

An important part of network and service coordination is dynamic resource allocation, i.e., deciding the amount of allocated resources for each placed instance. Existing coordination approaches often allocate fixed amounts of resources per instance, which easily leads to either over-allocation or under-allocation of resources and consequently to either waste of resources or bad service quality. My coordination approaches in Part I are more flexible and allocate resources relative to the load that each instance has to process. To do so, these approaches still rely on a priori knowledge of function $r_c(\lambda)$ to model the resource requirements for each involved service component c. Estimating such resource requirements manually is hard and prone to errors, even for domain experts, such that human operators often resort to allocating fixed, large amounts of resources.

To solve this problem, I propose the first machine learning approach in Part II. It trains machine learning models on data from real service components, comprising measurements of allocated resources and resulting performance. For each component, the trained models can then accurately predict the required resources to handle a certain load. I integrate these machine learning models into the Bidirectional Scaling and Placement Coordination Approach (BSP) from Chapter 4 and evaluate their impact on resulting service placement. The evaluation based on real-world data shows that using suitable machine learning models effectively avoids over-allocation and under-allocation. Compared to typical fixed resource allocation, it reduces resource consumption by up to 12 times and total delay by up to 4.5 times.

This chapter is based on my paper [S35], for which I developed the ideas and approaches, wrote code, and produced all experiment results. Narayanan Puthenpurayil Satheeschandran did some preliminary work on the same topic in his master's thesis [222] that also contributed to the paper. I designed the topic and work plan of his thesis and advised it closely, providing the key ideas and strongly guiding development and evaluation. Based on lessons learned from this master's thesis, I reimplemented, extended, and evaluated all ideas and approaches and finally prepared and wrote the paper. This chapter contains verbatim copies of this paper [S35]: Stefan Schneider et al. "Machine Learning for Dynamic Resource Allocation in Network Function Virtualization." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2020, pp. 122–130, © 2020 IEEE. In this chapter and throughout my dissertation, I consistently write in the first-person singular form ("I" rather than "we") for ease of reading. The corresponding source code is publicly available on GitHub [S14].

7.1. Introduction

Network and service coordination requires service scaling and placement as well as flow scheduling and routing as described in Chapter 3. For each placed instance, coordination approaches need to allocate or reserve a suitable amount of resources. Conventionally, each service component c is instantiated with a fixed amount of resources r_c that is predefined by the developer or operator. For example, in Network Function Virtualization (NFV) where network services consist of chained Virtual Network Functions (VNFs), VNF developers can specify fixed resource requirements per instance in the VNF descriptor [71]. These descriptors are then uploaded to the orchestration system and used for resource allocation. However, VNFs and other components typically do not require constant resources but need more or less resources depending on the traffic load [S9, 252]. Thus, allocating a fixed amount of resources easily leads to either under-allocation or over-allocation of resources. In case of under-allocation, placed instances lack resources to process all packets, leading to packet drops and reducing service quality or even completely disrupting service functionality. Thus, a plausible strategy for operators is to avoid under-allocation by allocating fixed, large amounts of resources to each instance. This, in turn, leads to over-allocation, wasting resources, blocking other instances from using these resources, and resulting in unnecessarily high costs.

To this end, some approaches for network and service coordination [37, 95, 214, 59, S2, S18, S24], including those presented in Part I, consider resource requirements relative to the load an instance has to handle. This relationship between load and required resources can be modeled by a function $r_c(\lambda)$ for instances of a component c with incoming traffic of data rate λ (Section 3.1.2), where resources $r_c(\lambda)$ may be required to meet a predefined Service-Level Agreement (SLA). While this approach allows dynamic and more fine-grained resource allocation, it relies on accurately defining function $r_c(\lambda)$ for all components to model their resource requirements. Even with domain knowledge of a specific service component, it can be very difficult to manually estimate its exact resource requirements and to model $r_c(\lambda)$ accurately.

To mitigate these issues and take appropriate resource allocation decisions, an understanding of the relationship between the load an instance is supposed to handle, the resources dedicated to it, and the resulting performance is necessary, which can be expressed in a so-called *performance profile*. Performance profiling is an approach to measure the performance for instances of a given component with varying resource configurations [201, 216, 132]. The relationship between a component's performance and resource requirements is often non-linear [252] and hard to extract from raw measurement data, which can contain millions of data points [S9].

These properties make performance profiles ideal candidates for machine learning, automatically deriving suitable, accurate, and compact models from measurement data. I compare machine learning approaches from different families of regression algorithms (e.g., linear, tree-based, kernel-based) as well as neural networks. For each approach, I evaluate the approximation accuracy and ability to generalize to values that lie between measured data points. Rather than proposing a completely new coordination approach, I focus on how to model and determine the required amount of resources per component instance. The derived models can then be used inside existing or future coordination approaches to accurately approximate resource requirements for each instance and allocate resources accordingly. This dynamic resource allocation using machine learning helps to ensure good service quality while avoiding over-allocation and under-allocation. Overall, the contributions of this chapter are:

- I design a modular workflow for deriving trained machine learning models from raw performance measurements and for using them in coordination approaches to accurately approximate resource requirements (Section 7.4).
- In Section 7.5.1, I use six different machine learning algorithms on real-world data to train and compare resulting models for approximating resource requirements.

- I integrate these models for dynamic resource allocation into the existing BSP coordination approach [S2], evaluate their impact on resulting solution quality, and investigate potential trade-offs regarding computational runtime (Sections 7.5.2 and 7.5.3).
- The corresponding source code and models are publicly available on GitHub [S14] to encourage reproduction and reuse.

7.2. Related Work

7.2.1. Approaches Without Machine Learning

Most existing coordination approaches consider predefined, fixed resource requirements per placed instance [160, 47, 88, 56, 91, 189]. While these algorithms may use horizontal scaling, i.e., adding or removing instances, to adjust to changing demand, they completely neglect vertical scaling, i.e., dynamically adjusting the allocated (or reserved) resources per placed instance. As mentioned before, predefined, fixed resource allocation can easily lead to poor service quality or wasted resources.

Recent approaches [37, 214, 59, S2] improve this simplistic model of fixed resource requirements by considering a linear relationship between the amount of allocated resources and the resulting performance. While this increases the flexibility of the model, the linear function requires proper parametrization, which is non-trivial. It is often also unrealistic, since components usually do not have a strictly linear relationship between allocated resources and their performance [252, 60]. To this end, some authors [60, 214, 192] propose the use of piece-wise linear functions instead. Piece-wise linear functions approximate non-linear functions by splitting them into separate pieces that are then each approximated by different linear functions. Optimization problems with piece-wise linear functions (but not with non-linear functions) can still be solved by standard Mixed-Integer Linear Program (MILP) solvers and are thus useful for MILP approaches, e.g., in Chapter 5. A major drawback, especially for highly non-linear functions [67], is that the approximation with piece-wise linear functions either becomes increasingly inaccurate or increasingly cumbersome as the function has to be split in smaller and smaller pieces. Similarly, Eramo et al. [68] use a quite complex model considering packet length, flow data rate, and the traffic state (assuming cycle-stationary traffic) to compute a realistic model for performance and resource requirements of a given component. Due to complex required a priori knowledge (e.g., about the traffic states), such a model is challenging to use.

Overall, most models for resource allocation that are used in existing coordination approaches are either very simplistic or require human expert knowledge of each used service component. In contrast, the proposed machine learning approach in this chapter learns accurate models automatically and directly from real performance measurements. To this end, this chapter builds on existing work on performance profiling, particularly in the context of NFV [201, 216, 132]. The proposed machine learning approach is complementary to most coordination approaches as the resulting trained machine learning models can be integrated into existing approaches with little overhead.

7.2.2. Machine Learning Approaches

As one of the open challenges for machine learning in networking, Ayoubi et al. [20] mention the mapping from high-level requirements to low-level configurations. This chapter addresses this challenge by automatically mapping desired performance (i.e., processing a certain data rate) to required resource configurations. Similarly, other machine learning-based approaches have been used successfully to predict instances' processing delays [147] and traffic demands [277, 75]. The approach in this chapter is complementary and could be combined with this related work.

Other authors have also proposed machine learning for predicting resource requirements [230, 177] but with important limitations: Sembiring et al. [230] only rely on historical data but do not take traffic load into account for predicting resource requirements. Mijumbi et al. [177] use graph neural networks to derive a model for resource requirements from the topology of sub-components within a given component. In contrast to my approach, theirs requires detailed insight into internals of each component. It also disregards that different instances of the same component may process different traffic loads and thus may have very different resource requirements, limiting the accuracy of their model.

Most similar to the approach in this chapter, van Rossem et al. [252] use performance profiling to collect performance measurements and propose machine learning to learn from these measurements. Compared to my work, van Rossem et al. focus more on the process of performance profiling but also mention dynamic resource allocation as high-level use case. Complementing and going beyond their work, I apply machine learning on available performance measurements and focus on integrating the trained models into existing coordination approaches for dynamic resource allocation. Additionally, I evaluate the impact of different models on the resulting service placements.

7.3. Problem Statement

The goal of this chapter is to use machine learning to approximate the required resources for instances of each service component. The resulting machine learning models can be used inside existing coordination approaches for improved dynamic resource allocation.

Figure 7.1 illustrates the problem using example data, which is similar to the real-world Nginx dataset used for evaluation (Section 7.5.1). Here, the true resource requirements increase superlinearly with increasing traffic load (black, solid line). The typical approach of assigning fixed resources based on the maximum expected traffic (here, $r_c = 0.8$) leads to severe over-allocation (red, dotted line). Such severe over-allocation not only wastes resources but may also affect service placement and resulting Quality of Service (QoS) as illustrated in Figure 7.2. The figure shows an example with two placement options of two chained instances, depending on the resource allocation model of Figure 7.1. The true required resources $r_c(5) = 0.3$ would allow placing both instances close to the user with low delay (dark boxes). However, when allocating fixed resources $r_c = 0.8$ to each instance, they have to be distributed over different nodes (here, with $\text{cap}_v = 1$), leading to unnecessary link delay and reduced service quality (red boxes). This is particularly relevant for network and service coordination in edge computing scenarios, where nodes have very limited capacities.

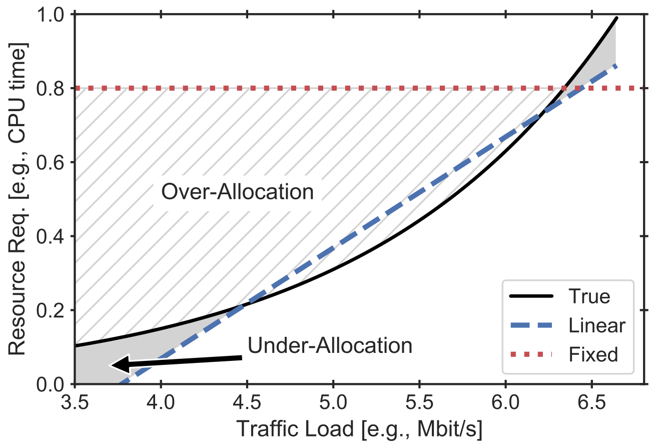


Figure 7.1.: Example of true resource requirements in comparison to fixed and linear resource allocation. (Figure adapted from [S35]; © 2020 IEEE.)

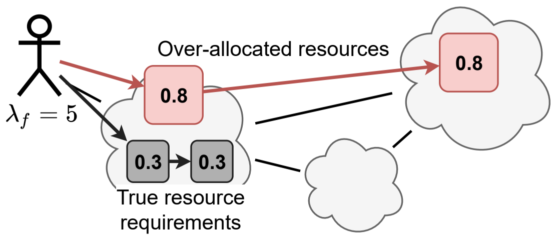


Figure 7.2.: Two example placement options depending on the instances' resource allocation. Severe over-allocation leads to placement of instances at separate nodes and additional link delay. (Figure adapted from [S35]; © 2020 IEEE.)

To overcome the problems of typical fixed resource allocation, coordination approaches need to allocate resources dynamically by approximating components' true resource requirements as closely as possible. The key challenge here is to model resource requirements $r_c(\lambda)$ accurately by choosing a suitable approximation function and to parameterize it correctly. To quantify the approximation accuracy, I consider the Root Mean Squared Error (RMSE). The RMSE averages all approximation errors over n samples with y_i being the true target value and $\hat{y}_i$ the approximated value (here, required Central Processing Unit (CPU) time): $\text{RMSE} = \sqrt{\sum_{i=1}^{n} \frac{(\hat{y}_i - y_i)^2}{n}}$. Lower RMSE indicates more accurate approximation of the true resource requirements.

Linear functions are simple and commonly used to approximate $r_c(\lambda)$ but can still be hard to parameterize manually. Therefore, I propose to automatically derive suitable parameter settings from real-world measurement data through automated regression techniques (Section 7.4). Even with proper parametrization, linear approximation still easily leads to under-allocation or over-allocation if the true resource requirements are non-linear, e.g., in Figure 7.1. Hence, I also discuss non-linear approximation models that can be parameterized automatically by using machine learning on available measurement data. In Section 7.5, I then compare both approximation accuracy (in terms of RMSE) and resulting service placements when integrating different approximation models with an approach for network and service coordination.

7.4. Machine Learning Approach

To ensure that the allocated resources closely match the real requirements of each instance, I propose to use performance profiles derived from measurement data using machine learning. I provide an overview of the proposed approach in Section 7.4.1 and describe its details in Sections 7.4.2–7.4.4.

7.4.1. Overview

Figure 7.3 illustrates the proposed overall approach, which consists of four steps. It starts with profiling a given component (e.g., a VNF in NFV), testing it systematically with different resource configurations and measuring its corresponding performance (step 1). In step 2, the resulting raw performance measurements are used to train machine learning models. In doing so, these machine learning models learn to approximate the amount of resources a specific component requires to successfully handle a given traffic load. The trained models generalize the measured data such that it can approximate resource requirements for traffic loads that lie between measured data points. In step 3, the machine learning models are integrated into a coordination approach, where they are used to approximate the resource requirements $r_c(\lambda)$ for an instance of a component c that needs to process traffic load λ. Hence, in step 4, these coordination approaches can leverage the machine learning models for accurate resource allocation, mitigating the problem of over-allocation and under-allocation.

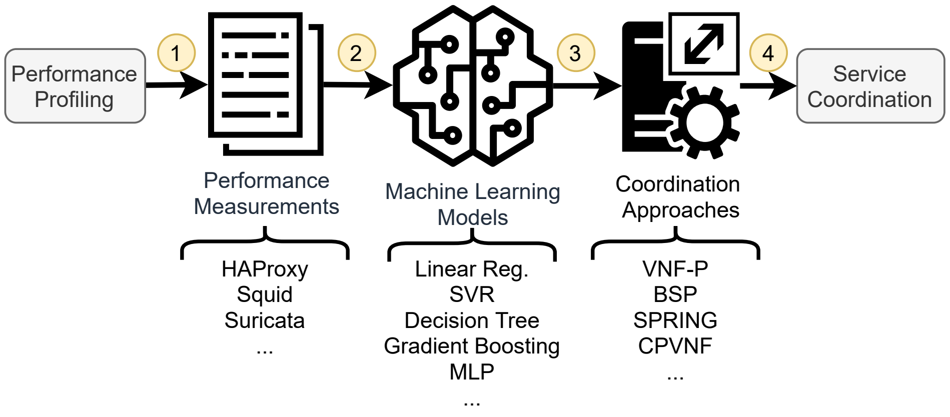


Figure 7.3.: Modular workflow from raw performance measurements to machine learning-based resource allocation and placement. (Figure adapted from [S35]; © 2020 IEEE.)

The whole process from step 1 to 3 can be performed offline in advance, e.g., directly after developing and publishing a new component. While the process is time-consuming, it only has to be performed once per component since trained machine learning models can be saved and reused later. In scenarios with limited time, e.g., where new services need to be deployed quickly, limited but relevant data could be collected through time-constrained profiling [200] and be used for short pretraining. The machine learning models could then be continuously improved and retrained on more data collected during deployment [256]. In step 4, the coordination approach loads the trained machine learning models for all requested components and uses them to dynamically allocate resources to each placed instance. In contrast to training machine learning models, inferring approximated resource requirements is very fast (less than 1 ms in the evaluation) and can be done online.

All elements in the described process are freely exchangeable. In particular, the approach is not limited to a specific kind of component, machine learning model, or coordination approach (Figure 7.3). Instead, performance profiling can automatically measure the performance of any available component and the measurement results can be used for training any machine learning regression model; I evaluate different options in Section 7.5. Finally, the trained machine learning models can be integrated into existing coordination approaches with relatively little overhead.

7.4.2. Performance Profiling

Performance profiling is the process of systematically testing a given component under different resource (and other) configurations while measuring its performance, e.g., in terms of maximum sustainable traffic load, when sending as much traffic as possible through an instance of the component. Particularly in the context of NFV, several authors have proposed how to automate this process [201, 216, 132] and perform it effectively within limited time [200]. Performance profiling is typically performed offline, i.e., before deployment of a component, either by the developer or by the operator directly after onboarding a new component. In principle, more profiling data could also be collected online during service deployment [185, 89]. There is also a trend to open datasets of community-created performance profiles [S9].

Performance profiling leads to massive amounts of raw measurements per component (often millions of data points [S9]), reflecting the real behavior and dynamics of a component. These recorded datasets usually contain many, potentially component-specific configuration parameters and measurement metrics. For the purpose of resource allocation, I am only interested in metrics related to resource requirements and the corresponding performance. Typically, various configurations of CPU time (in percent) and memory (e.g., in MB) are tested per component, recording performance in terms of maximum sustainable traffic load (e.g., in Mbit/s). In addition, other parameters possibly affecting performance (e.g., configurable thread count) could be considered.

7.4.3. Learning Resource Requirements

For training machine learning models on performance measurement data, the dataset needs to be split into a feature set containing the performance measurements (e.g., maximum sustainable traffic load) and a set of target values containing the resource requirements. Due to possible errors in the performance profiling process, some values may be missing in the dataset and can be replaced through imputation (e.g., with the median) [123]. I also found that it is important to scale all features and targets to values between 0 and 1, particularly for neural networks, which are sensitive to the ranges of input data (Section 7.5.1). Without scaling, the measurement data is often in vastly different value ranges, depending on the unit of recorded data, e.g., data rate in the thousands of kB/s vs. required CPU time in $[0, 1]$. Similarly, hyperparameter tuning can greatly improve approximation accuracy, especially for more complex models with many hyperparameters, e.g., neural networks (Section 7.5.1).

Generally, any machine learning regression algorithm can be used for training models on performance measurements. For components with a linear or almost linear relationship between resource requirements and resulting performance, linear regression or ridge regression [113] are suitable. Such linear approximation is simple and fast without requiring much (or any) hyperparameter tuning (Section 7.5.1). If the relationship is clearly non-linear or unknown, other non-linear approximation techniques are preferred, e.g., Support Vector Regression (SVR) [62], decision trees, or neural networks. Often, a combination of multiple techniques using ensemble learning works better than each individual technique [220]. In Section 7.5, I evaluate and compare four individual regression techniques and two ensemble learning approaches.

Independent of the selected machine learning algorithm, the trained models can be saved and reused later without retraining. Similarly, the scaler used for preprocessing the data can be saved and reused to ensure consistent processing of any new data. I suggest to upload the trained models in an online repository (e.g., extending SNDZoo [S9]) to enable reuse by others. Orchestration platforms can then retrieve the trained models directly from these repositories and use them inside their coordination approaches.

7.4.4. Integration with Coordination Approaches

The trained models for all components of interest (e.g., all components C) can be integrated into an existing or new coordination approach with minor technical overhead. Each service component $c \in C$ could be annotated with the corresponding trained model, which is used to approximate $r_c(\lambda)$. The coordination approach simply needs to load the associated model into memory and then query the model to approximate the required resources for all instances depending on their component and total traffic rate. Loading and querying works similarly for all kinds of models, independent of whether the loaded model is a simple linear function or a trained neural network (or any other approximation function). In fact, swapping different machine learning models, e.g., for different components, is simple and completely transparent to the rest of the coordination approach due to the consistent interface of common machine learning models using the sklearn Application Programming Interface (API) [33].

As an example for a state-of-the-art coordination approach, I adjust BSP from Chapter 4 to use machine learning models for dynamic resource allocation. The integration with BSP is fairly simple and only requires to change less than 100 lines of code (available online [S14]). Specifically, BSP tries to schedule new flows to existing instances to minimize the number of placed instances. To avoid over-subscribing node resources, BSP uses the trained models to approximate and predict the required resources of each instance (within a certain delay bound) if it were to process the new flow. Among the available instances with sufficient resources, it chooses an instance that is reachable with short path delay. If no instances have sufficient resources, it starts a new instance nearby, selecting a node with sufficient resources according to the approximated total resource requirements. Hence, the resource requirements approximated by the trained machine learning models directly affect BSP's scaling, placement, and scheduling decisions. More details of the BSP embedding procedure are described in Section 4.2.2.

Using machine learning models for approximating resource requirements also enables more algorithmic improvements of existing coordination approaches. Being aware of the potentially non-linear relationship between allocated resources and resulting performance, coordination approaches could actively optimize the total traffic load and resulting resource requirements per instance through their scheduling and scaling decisions. This an interesting direction for future research, extending existing work in non-linear resource allocation [198, 250] to the use of machine learning models.

7.5. Evaluation

To illustrate and evaluate the proposed machine learning approach for dynamic resource allocation, I use performance measurements of two real-world service components. In particular, I use performance measurements of the popular Squid cache [240] and the Nginx proxy [188] from SNDZoo [S9]. Each of these datasets has 4.6 million data points with a subset being performance measurements under varying resource configurations.

In Section 7.5.1, I use six different machine learning algorithms to learn the relationship between instances' performance (maximum sustainable traffic load) and resource requirements (CPU) from these datasets. I illustrate the importance of hyperparameter tuning and compare the resulting approximation accuracy. In Sections 7.5.2 and 7.5.3, I use the adjusted BSP coordination approach described in Section 7.4.4 with different machine learning models to approximate resource requirements and allocate resources accordingly. I evaluate the impact of these different models on the resulting service placements and on algorithm runtime using machines with an Intel Xeon W-2145 CPU and 32 GB RAM.

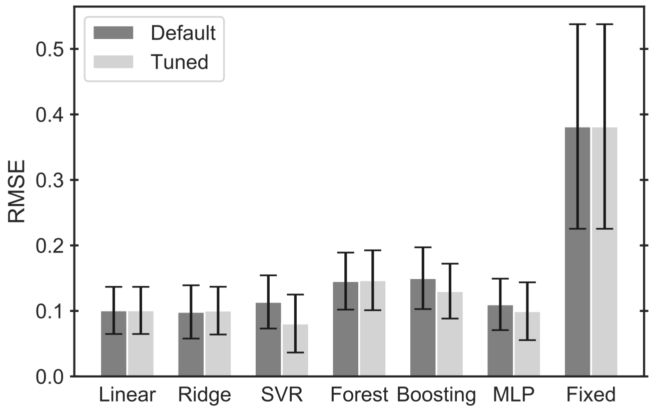


Figure 7.4.: RMSE of different machine learning models with default and tuned hyperparameters on the Squid dataset. (Figure adapted from [S35]; © 2020 IEEE.)

7.5.1. Approximation Accuracy

To find the most accurate machine learning algorithm for learning from performance measurements, I test and compare six different algorithms from different families of machine learning approaches: Linear regression, ridge regression [113], SVR [62], random forest [31], gradient boosting [84], and neural networks. Random forest and gradient boosting are examples of commonly used ensemble methods. For the neural network, I consider the common Multi-Layer Perceptron (MLP) architecture [196] with a single, densely connected hidden layer (128 hidden

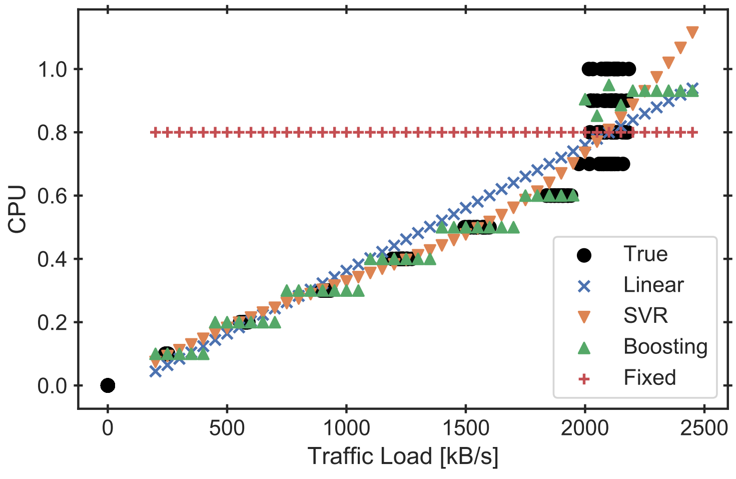


Figure 7.5.: Squid performance measurements and approximations of different machine learning models. (Figure adapted from [S35]; © 2020 IEEE.)

units). In addition, I compare these machine learning models with the typical fixed model that always assigns a predefined, fixed amount of resources to all placed instances.

Before being able to train machine learning models, the used measurement data has to be preprocessed (Section 7.4.3). For faster processing and training, I filter the data to a subset only containing relevant features, i.e., measurements of maximum sustainable traffic load for varying CPU configurations. Additionally, I fill any missing values and scale the features, here, the maximum sustainable traffic load, to a range of $[0, 1]$, similar to the target values (CPU time). Before scaling, the maximum sustainable traffic load was in the range of $[0, 3000]$kB/s, i.e., in a vastly different range than the configured CPU times ($[0, 1]$). Feature scaling is especially crucial for neural networks [127] but also for SVR. Without feature scaling, the considered MLP neural network performs terribly—leading to an RMSE that is roughly 150 times worse than with feature scaling. For other machine learning algorithms, e.g., linear regression or ensemble learning, feature scaling only brings negligible benefits.

After preprocessing, I train and evaluate each machine learning model using 5-fold cross validation [17]. For each model, I optimize the loss in terms of RMSE as defined in Section 7.3. The figures show the mean RMSE with the 95 % confidence interval as error bars.

Squid Cache

Figure 7.4 shows the RMSE of the six machine learning models for the Squid cache dataset. In addition to the six machine learning models, the figure also shows the error for the currently prevailing model of assigning fixed resources to each placed instance (here, configured to 0.8 CPU). Not surprisingly, the fixed model does not approximate the true resource requirements of the Squid cache well, which leads to a very high average RMSE of around 0.4. All machine learning models perform significantly better (RMSE between 0.1 and 0.2). Figure 7.4

also compares the RMSE of all models with default hyperparameters and after automated hyperparameter tuning using grid search. With default hyperparameters, the linear models (linear and ridge regression) perform best, but after hyperparameter tuning, SVR has a lower average RMSE, closely followed by MLP. SVR and MLP have many hyperparameters such that tuning can improve their performance significantly. In contrast, the simpler, linear models only have very few hyperparameters and do not noticeably benefit from hyperparameter tuning.

As visualized in Figure 7.5, the performance measurements of the Squid cache show a strong linear relationship between maximum sustainable traffic load and CPU for most measurements (up to 2000 kB/s), explaining the high accuracy of linear models. Figure 7.5 also shows the approximations of some of the tuned and trained machine learning models as well as the fixed model. In contrast to the fixed model, all machine learning models fit the data quite well. SVR fits a non-linear function, which is quite smooth thanks to regularization. Conversely, the gradient boosting model increases its approximation stepwise, which is due to the decision trees being used within the model, breaking the data into different parts with similar approximated value.

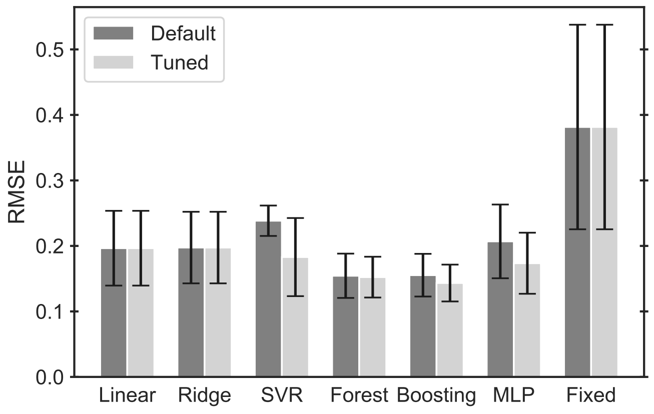


Figure 7.6.: RMSE of different machine learning models with default and tuned hyperparameters on the Nginx dataset. (Figure adapted from [S35]; © 2020 IEEE.)

Nginx Proxy

Similar to the Squid cache, I train all machine learning models on the Nginx dataset, with and without hyperparameter tuning, and show their RMSE in Figure 7.6. Again, hyperparameter tuning mostly improves the performance of the more complex models, here SVR, MLP, and gradient boosting. For Nginx, the two ensemble methods perform better than all other machine learning models.

As can be seen in Figure 7.7, Nginx' true required CPU for increasing maximum sustainable traffic load first increases roughly linearly but then rises dramatically for traffic loads above 2000 kB/s. While this is also visible for the Squid cache (Figure 7.5), it is much more pronounced in the Nginx dataset, making linear approximation less suitable here.

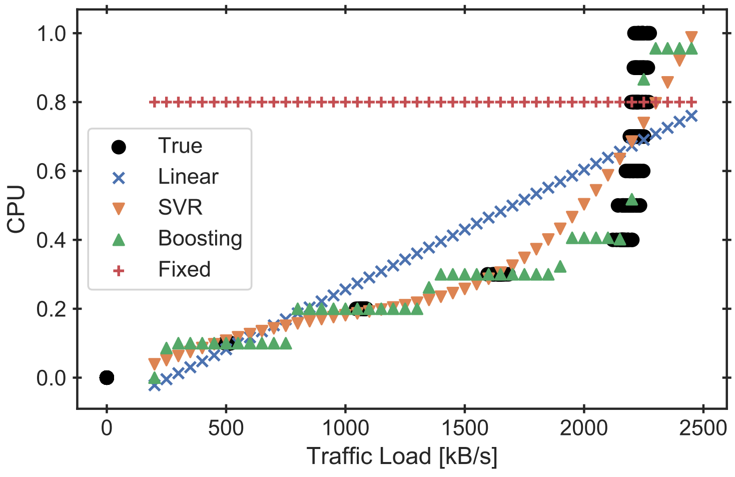


Figure 7.7.: Nginx performance measurements and approximations of different machine learning models. (Figure adapted from [S35]; © 2020 IEEE.)

7.5.2. Impact on Network and Service Coordination

The previous section reveals significant differences in approximation accuracy between different machine learning models and, in particular, their remarkable advantage over a fixed model. In this section, I evaluate the impact of different models on network and service coordination when using them for dynamic resource allocation inside an existing coordination approach. Here, I consider BSP from Chapter 4 for coordination in the real-world Abilene network topology [142] with unit compute capacity at each node. Furthermore, I consider three traffic sources with an increasing number of flows leaving the sources, leading to growing traffic loads. I assume a web-based network service consisting of the Squid cache and the Nginx proxy chained together.

I consider the adjusted BSP algorithm in three different variants with different models to approximate resource requirements and allocate resources for instances of Squid and Nginx, respectively: In the first variant, BSP uses the currently prevailing fixed model with 0.8 CPU per instance. In the second, it uses the trained linear models for both components (Squid and Nginx), which represent the best case, i.e., with proper parametrization, that is achievable with recent approaches using linear functions for $r_c(\lambda)$. In the third variant, BSP uses the best models from Section 7.5.1, i.e., the trained SVR for approximating Squid's resource requirements $r_{\text{Squid}}(\lambda)$ and gradient boosting for Nginx ($r_{\text{Nginx}}(\lambda)$).

The true resource requirements of Squid and Nginx are only known for measurements in the two datasets but not their true function $r_c(\lambda)$. Compared to other models, SVR and gradient boosting achieved the smallest RMSE in Section 7.5.1. Thus, I assume SVR and gradient boosting to be closest to the true resource requirements of the two components, respectively, with negligible over-allocation or under-allocation.

In the following, I evaluate the three variants while varying the traffic load. In these experiments, BSP dynamically scales the number of placed instances, schedules flows to these instances, and allocates resources according to the different models (Section 7.4.4). I compare the experiment outcomes in terms of three metrics of interest: The total amount of allocated resources, the number of placed instances, and the total delay.

Impact on Total Resource Allocation

Figure 7.8 (left) shows the total allocated CPU with increasing traffic load for the three different algorithm variants. With more traffic to process, BSP places more instances (compared next), which require more resources in total to properly handle the increasing traffic load (considered here). The amount of allocated resources depends on the traffic load, but is also strongly affected by the approximated resource requirements $r_c(\lambda)$ of each model. The figure shows that assigning a predefined, fixed amount of CPU resources to each instance leads to a huge amount of total allocated resources that grows rapidly with increasing traffic load, quickly filling up and overloading the network's resource capacities. Here, allocating a fixed amount of resources leads to up to 12 times more allocated CPU resources than with SVR and gradient boosting, signifying massive over-allocation.

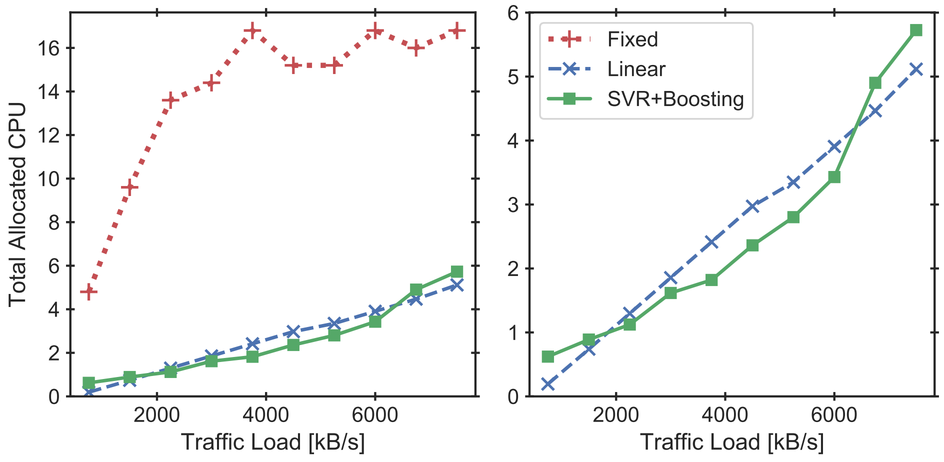


Figure 7.8.: Total allocated CPU in resulting service placements computed with different models. Right: Close-up comparison. (Figure adapted from [S35]; © 2020 IEEE.)

Compared to the fixed model, the differences in total allocated CPU are much smaller between linear approximation and the combined SVR and gradient boosting. Still, Figure 7.8 (right) shows that the total allocated CPU resources in resulting service placements with the linear models is up to 33 % higher or up to 69 % lower than in service placements computed with SVR and gradient boosting. Since the SVR and gradient boosting models are more accurate than the linear models (Section 7.5.1), this implies over-allocation and under-allocation for service placements computed with the linear models.

Impact on Number of Placed Instances

Next, I consider the same experiment outcomes but compare the number of placed instances instead of the total allocated resources. As described in Sections 7.4.4 and 4.2, BSP tries to minimize the number of used instances by scheduling flows to existing instances if the approximated resource requirements do not exceed node capacity. In the considered scenario with limited node capacities, BSP needs to start more and more instances at different nodes to balance the increasing load. Figure 7.9 shows that this is most clearly visible for the case of fixed resource allocation, where 2-3 times more instances are placed than in the SVR and gradient boosting case. Again, the results with the linear models and the combined SVR and gradient boosting are quite similar with respect to the number of placed instances. For higher load, the slight CPU over-allocation of the linear models compared to SVR and gradient boosting adds up and forces BSP to instantiate slightly more instances.

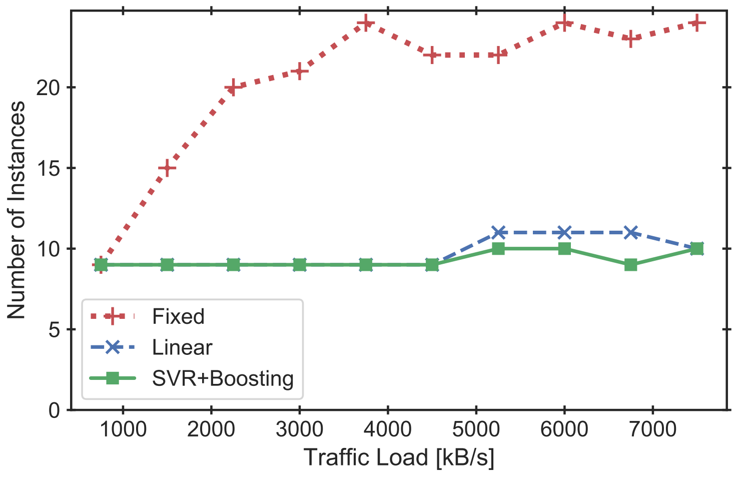


Figure 7.9.: Model impact on the number of placed instances. (Figure adapted from [S35]; © 2020 IEEE.)

Impact on Total Delay

Finally, Figure 7.10 shows the total link delay of the computed service placements. As before, service placements with the fixed model lead to much worse results and up to 4.5 times higher total delay than with SVR and gradient boosting. Here, the higher total delay is caused by the higher number of instances in the corresponding service placements. The more instances are placed at different nodes in the network, the more traffic has to be sent via links between them, leading to higher total delay. Again, the linear models and the SVR and gradient boosting models lead to similar total delay for low traffic. For higher traffic, the total delay with the linear models is up to 23 % higher than with SVR and gradient boosting due to more instances in the service placement with linear approximation.

7.5.3. Runtime Analysis

Short computational runtime is important to quickly react to changes in demand. Training and hyperparameter tuning can be performed offline up front. In contrast, coordination approaches may require the trained models to quickly and frequently approximate and predict potential resource requirements during online operation.

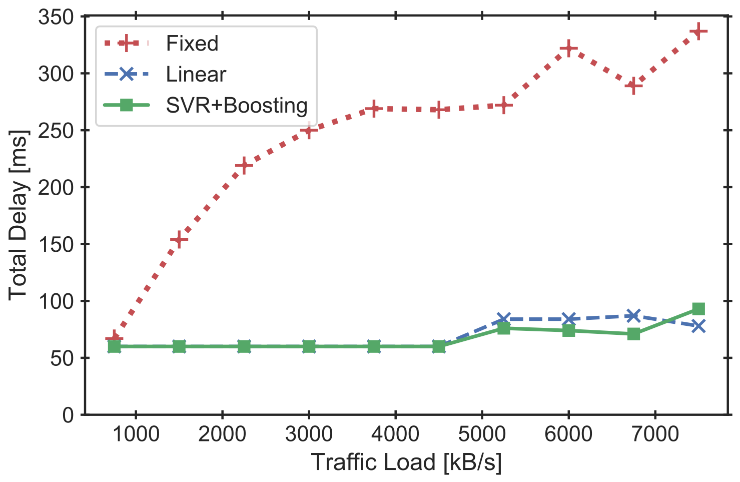


Figure 7.10.: Model impact on the total delay. (Figure adapted from [S35]; © 2020 IEEE.)

Time per Prediction

Figure 7.11 (left) shows the measured time per prediction for each of the trained and tuned models of Section 7.5.1. The fixed model returns the same predefined value instantly without any computation. Also, all six machine learning models make very fast predictions within 1 ms and, except for random forest, even within 0.1 ms. Differences in prediction time reflect the complexity of the models: The two simple linear models (linear and ridge regression) are fastest and the two ensemble learning models (random forest and gradient boosting) are slowest. For each prediction, random forest and gradient boosting need to query multiple (here, 100) decision trees and merge their approximated values, making them slower than the individual regression techniques. While random forest allows its decision trees to grow very deep, making it relatively time expensive, gradient boosting is much faster because it limits the tree depth. Similarly, the MLP model uses a shallow neural network (1x 128 hidden units), enabling fast predictions.

Time for Service Placement

Using these models in the adjusted BSP coordination algorithm, Figure 7.11 (right) shows the algorithm's total runtime on a logarithmic scale. The roughly 10+ times higher prediction time of the random forest model compared to other models also leads to much higher algorithm runtimes of over 1 min. All other machine learning have largely similar algorithm runtimes between 1 s and 2 s.

Interestingly, the BSP algorithm runtime is not just affected by the prediction times of the individual models but also by their approximation accuracy. Models that over-approximate the required resources force BSP to balance these resources over more instances at different nodes. As BSP repeatedly iterates over all instances, e.g., to schedule new flows to existing instances, its total algorithm runtime increases with more instances (algorithm details in Section 4.2). This effect is most noticeable for the fixed model: Even though it has almost zero prediction

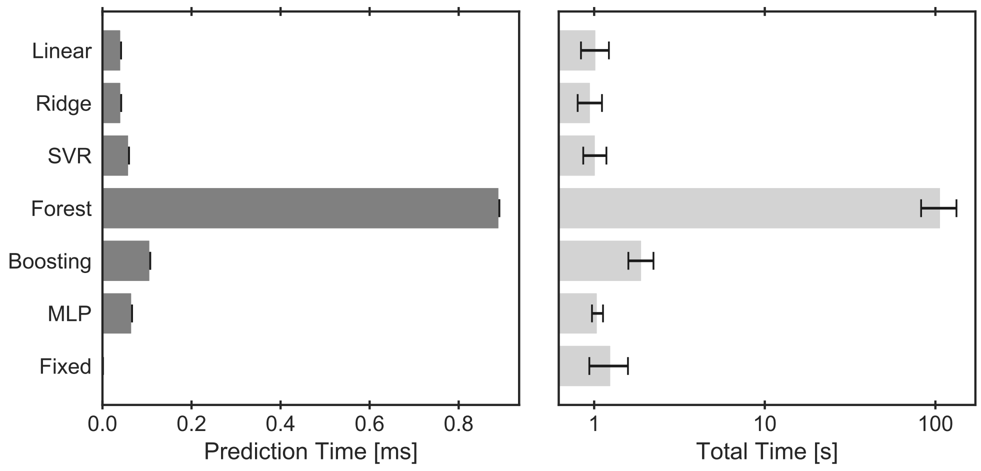


Figure 7.11.: Prediction and placement times for different models. (Figure adapted from [S35]; © 2020 IEEE.)

time, the total algorithm runtime with the fixed model is higher than with many machine learning models. This is because the fixed model's severe over-allocation leads BSP to place many instances, increasing its total runtime. Hence, while there is a trade-off between prediction time and accuracy, many machine learning models are accurate but still fast, which clearly pays off in overall service quality and even algorithm runtime.

7.6. Conclusion

This chapter shows how machine learning can be used to automatically learn from real-world data, deriving models that can accurately approximate components' resource requirements depending on the traffic load. Using these models in coordination approaches can significantly impact the solution quality of resulting service placements: The evaluation shows that using typical fixed resource allocation can lead to massive over-allocation of resources, many placed instances, and high delay, potentially leading to high costs and bad service quality. Instead, using trained machine learning models inside a coordination approach (e.g., the conventional heuristics from Part I) improves dynamic resource allocation and results in less wasted resources and better service quality.

In future work, the approach could be extended to also derive and predict other important parameters from real-world data, e.g., components' processing delays. When integrating these machine learning models into existing coordination approaches, the scaling and scheduling decisions of these approaches could be further improved taking the trained models into account. This work also opens interesting research directions to reduce the initial training overhead through transfer learning or continuous retraining.

In general, a key characteristic of this machine learning approach is that it relies less on human expertise (here, knowledge of components' resource requirements) than typical conventional coordination approaches. Instead, it leverages available data to accurately and automatically derive insights, here, learning components' resource requirements. The following chapters of

Part II extend this general idea to coordination approaches that learn network and service coordination entirely by themselves without relying on conventional heuristics or human instructions.

8. Self-Learning Coordination

Network and service coordination is usually solved with custom, expert-designed approaches such as the conventional approaches in Part I. While this typically works well for the considered scenario, the models often rely on unrealistic assumptions or on knowledge that is not available in practice (e.g., a priori knowledge). Chapter 7 partially mitigates this problem by replacing expert knowledge of service components' resource requirements with automatically derived machine learning models. This chapter goes one step further and proposes DeepCoord, a novel Deep Reinforcement Learning (DRL) approach that learns network and service coordination from its own experience when interacting with the network. It is geared towards realistic assumptions and relies on available, possibly delayed monitoring information. Rather than defining a complex model or an algorithm on how to achieve an objective, the model-free approach adapts to various objectives and traffic patterns. An agent is trained offline without expert knowledge or human instructions and then applied online with minimal overhead. Compared to the Bidirectional Scaling and Placement Coordination Approach (BSP) from Chapter 4, i.e., a state-of-the-art conventional heuristic, DeepCoord significantly improves flow throughput (up to 76 %) and overall network utility (more than 2x) on real-world network topologies and traffic traces. It also supports optimizing multiple, possibly competing objectives, learns to respect Quality of Service (QoS) requirements, generalizes to scenarios with unseen, stochastic traffic, and scales to large real-world networks.

This chapter is based on my papers [S21, S22], which build upon outcomes of the RealVNF project [S20]. In the project, student assistants helped with developing prototypes and collecting evaluation results. I led the RealVNF project over two years, proposed all key ideas, discussed them with project partners, and drove their development. I closely advised and guided the student assistants and wrote both published papers. This chapter contains verbatim copies of these papers [S21, S22]: Stefan Schneider et al. "Self-Driving Network and Service Coordination Using Deep Reinforcement Learning." In: *IFIP/IEEE Conference on Network and Service Management (CNSM).* 🏆 **Best Student Paper Award** 👥 **Invited Talk at the Internet Engineering Task Force (IETF) 110 Meeting**. IFIP/IEEE. 2020, © 2020 IEEE. Stefan Schneider et al. "Self-Learning Multi-Objective Service Coordination Using Deep Reinforcement Learning." In: *IEEE Transactions on Network and Service Management (TNSM)* 18.3 (2021), pp. 3829–3842, © 2021 IEEE. In this chapter and throughout my dissertation, I consistently write in the first-person singular form ("I" rather than "we") for ease of reading. The DeepCoord source code and the corresponding simulation environment are publicly available on GitHub [S25, S26].

8.1. Introduction

While network and service coordination has been the focus of intensive study [114, 110], most existing work (Section 8.2) has three major limitations. First, existing work mostly focuses on long-/medium-term planning on how to scale and place service components based on given service deployment requests, (e.g., [182, 179, 28, 144]). In doing so, hard-wired chains of component instances are placed in the network to process all incoming flows. This is problematic as operational reality often diverges from any initial plan. For example, actual service demand by users likely differs from the expected load in a given service deployment request. Hence, scaling, placement, and flow scheduling should be adjusted dynamically and online according to the actual service demand.

Second, existing approaches typically use heuristics or numerical solvers (e.g., [182, 91, S2]) and rely on carefully designed models, tailored to specific scenarios, and build on corresponding assumptions. Applying them to scenarios with slightly different assumptions or new optimization objectives (e.g., for QoS) may again require time-consuming manual adjustments by experts.

Third, these models often rely on information that is not available in practice, such as a priori traffic knowledge. In reality, complete knowledge about incoming traffic is not available instantly or even a priori but only after monitoring, often done periodically (e.g., default 1 min in Prometheus [204]). Within such a monitoring interval, numerous flows may arrive stochastically from users at various ingress nodes and need to be processed by service components even before information about these flows is globally available. This is particularly the case in scenarios with many, short flows (Case II in Section 3.1.3).

To overcome these limitations, I propose a novel approach for autonomous network and service coordination using model-free DRL. In the proposed approach, called DeepCoord, a centralized DRL agent is trained offline through interaction with the network environment, using its previous actions and experience as feedback. The trained DRL agent then autonomously provisions services online and controls dynamic flow scheduling. For scalability, the agent does not decide scheduling of each flow individually. Instead, it selects and periodically updates scheduling rules that are deployed at each network node in a distributed fashion and applied purely locally at runtime to incoming flows.

Practical application of DRL is known to be challenging and requires careful design of the corresponding observations, actions, and reward as well as integration into the overall system [64]. To address these challenges, I formulate a Partially Observable Markov Decision Process (POMDP) with observations based on realistically available monitoring data that is only intermittently available and that contains only aggregated, slightly delayed, and uncertain information. In particular, DeepCoord can be used without expert knowledge and it requires neither a priori traffic knowledge nor detailed knowledge of the network or involved services. DeepCoord uses continuous actions to allow fine-grained online control of service scaling and placement as well as for dynamic flow scheduling. Its reward function is designed to support multiple objectives, e.g., optimizing flow success and QoS, based on available monitoring data. Finally, I propose a framework architecture for integrating DeepCoord with a given network environment through custom adapters and publish an open-source prototype [S25]. While DeepCoord is trained in a centralized, offline fashion, the trained agent can handle rapidly arriving stochastic and bursty traffic, generalizes to new and unseen scenarios, and scales to large, real-world network topologies while making decisions within milliseconds. Overall, the contributions of this chapter are:

- In Section 8.3, I discuss the problem of online network and service coordination based on delayed monitoring information and propose different methods and example formulations for optimizing multiple, possibly opposing objectives.
- In Section 8.4, I address network and service coordination by formalizing the corresponding POMDP and designing the framework for DeepCoord, the proposed novel, self-learning DRL approach based on Deep Deterministic Policy Gradient (DDPG) [155].

- The evaluation in Section 8.5 shows that DeepCoord consistently outperforms existing approaches, requiring significantly fewer resources to ensure high success rates on real-world network topologies. It also generalizes to unseen traffic patterns, learns to optimize multiple objectives, and scales to networks of realistic size while only relying on information that is available in practice. Specifically, I observe that DeepCoord reaches up to 76 % more successful flows and more than 2x higher total utility when optimizing multiple objectives than BSP.
- For reuse and reproducibility, all code is publicly available [S25].

8.2. Related Work

In this section, I discuss related work, first focusing on conventional approaches without DRL, e.g., heuristics or Mixed-Integer Linear Programs (MILPs), in Section 8.2.1. In Section 8.2.2, I compare existing approaches that, similar to DeepCoord, use DRL for self-learning network and service coordination.

8.2.1. Conventional Approaches Without DRL

The large majority of existing research builds on conventional approaches like heuristics or MILPs, similar to the coordination approaches in Part I. Authors often rely on unrealistic a priori knowledge and focus on a subset of the full network and service coordination problem (Chapter 3), which includes online service scaling and placement as well as runtime scheduling and routing of incoming flows. For example, multiple authors [182, 179, 28, 144] place services offline, assuming full a priori traffic knowledge. Other authors [85, 91] consider online service scaling and placement but disregard runtime scheduling of flows. Multiple authors [99, 29, 40] consider runtime scheduling of incoming flows but assume a given fixed placement. More related to my approach, Zhang et al. [276] consider joint online scaling and placement of services as well as flow scheduling. Such joint coordination is important to successfully balance trade-offs [203, 61].

While all aforementioned approaches work well in their considered scenarios, each approach is tightly bound to its underlying assumptions. As I show in the evaluation (Section 8.5), applying such conventional approaches to scenarios with slightly different assumptions (e.g., different, stochastic traffic patterns) can easily lead to significantly decreased performance. Adapting to new assumptions requires time-consuming manual adjustments by experts. In contrast, the proposed self-learning and self-adaptive, model-free DRL approach learns joint scaling, placement, and scheduling in varying scenarios without any prior knowledge.

In recent years, multiple authors proposed machine learning for predicting required resources [219, S35, 211] or upcoming traffic demands [105], which can be combined with online heuristic algorithms for proactive network and service coordination [75, 277]. I see this as a promising, complementary research direction that could be combined with DRL in future work.

8.2.2. Self-Learning DRL Approaches

Luong et al. [162] survey recent self-learning DRL approaches for networking. Related to this chapter, multiple authors address online service placement under changing load [199, 255, 261, 209], some considering stochastic traffic and QoS [261]. In contrast to my approach, Pei et al. [199] rely on a simulated copy of the network to quickly test, evaluate, and revert different actions to ultimately choose the best one in each time step. Furthermore, the final coordination decisions are made by a separate heuristic. Similarly, Solozabal et al. [238] rely on combining their DRL approach with a heuristic and restrict their approach to service placement in networks forming a star topology. Wang et al. [255] focus on scaling and placement of a single service with unknown background traffic using a multi-armed bandit. They assume up-front traffic knowledge per time step and equal flow scheduling between all instances, which could lead to high end-to-end delays and bad service quality. Similar to my approach, Quang et al. [209] place services online and Xiao et al. [261] consider processing and link delays as well as stochastically changing traffic. In contrast to DeepCoord, Quang et al. [209] and Xiao et al. [261] process incoming flows sequentially, making individual decisions per flow and service component. To address resulting infeasible or suboptimal placements, the authors roll back and undo previous decisions [261] or apply a separate heuristic algorithm to fix the DRL agent's proposed solution [209]. Making such expensive decisions per flow would not work in the problem considered here, where flows arrive rapidly (Case II in Section 3.1.3) and the DRL agent only observes delayed, aggregated, and partial network state.

Instead of per-flow decisions and similarly to how DeepCoord schedules incoming flows, Xu et al. [265] assign different portions of traffic to different paths. However, they focus on traffic engineering and do not consider service scaling and placement. The authors further assume traffic knowledge ahead of each time step and assist their DRL agent with a heuristic. Nasir and Guo [186] do consider delayed and partial observations but focus on power allocation in wireless networks.

The recent work by Gu et al. [98] is most related to DeepCoord. Similar to DeepCoord, they consider network and service coordination with flow scheduling, optimize a generic network utility, and build on DDPG [155], which I shortly describe in Section 8.4.3. Still, there are three main differences: 1. Gu et al. focus on compute costs but neglect resource and communication constraints such as node and link capacities and link delays. In contrast, I consider such constraints to support QoS optimization and scenarios with limited resources (cf. edge computing). 2. They assume traffic knowledge ahead of each time step. 3. They support their DRL approach with a custom heuristic to help with action selection and exploration. The authors show that both too much or too little support by the heuristic degrades performance, making it difficult to tune correctly.

Overall, to the best of my knowledge, DeepCoord is the first approach to address online network and service coordination in realistic scenarios with rapidly arriving flows and delayed, only partially observed network state. It works without support from a heuristic, which makes it more versatile and less error-prone. I hence believe the approach to be much better applicable to real-world systems than existing work, especially since it does not rely on a priori information.

8.3. Problem Statement

I consider the problem of network and service coordination as described in Chapter 3. Note that the DRL approach does not require explicit knowledge of the full, detailed network state including all parameters (Section 3.1). Instead, it observes only partial and delayed information that is available via monitoring (detailed in Section 8.4.2) and implicitly learns to adapt to a given scenario through feedback from its actions.

In this chapter, I mainly focus on scenarios with many, short flows (Case II in Section 3.1.3), potentially with QoS requirements. Particularly, I consider $\Theta_s = \{d_s^{\text{soft}}, d_s^{\text{hard}}\}$ with optional soft and/or hard deadlines $d_s^{\text{soft}}, d_s^{\text{hard}} \in \mathbb{R}_{\geq 0} \cup \varnothing$. Flows requesting service s must complete before hard deadline d_s^{hard}, defined relative to flow arrival t_f^{in}, or otherwise are dropped automatically. Achieving an end-to-end delay within soft deadline d_s^{soft} is desirable for best service quality but not imperative for flow success. The approach is not limited to short, QoS-sensitive flows but also supports fewer, longer flows (Case I) and flows without QoS requirements (i.e., $d_s^{\text{soft}} = d_s^{\text{hard}} = \varnothing$).

I assume that a monitoring system collects and synchronously reports metrics about each node $v \in V$ in fixed intervals of $\Delta > 1$ time steps. As many flows may arrive within Δ, the monitoring system only reports aggregated information over the last interval Δ but no per-flow details. In particular, I assume the monitored network state to merely include the number and aggregated rate of incoming, processed, and dropped flows at v and the average end-to-end delay of completed flows d_{avg} in the last Δ time steps (from $t - \Delta$ to t). This is in contrast to most related work, which assumes complete per-flow and often even a priori knowledge in every single time step.

DeepCoord maximizes the long-term utility U_T over all T time steps, reflecting the desired coordination goals. Utility U_T may consist of a single optimization objective $U_T = o_j$, a weighted (by w_j) sum $U_T = \sum_j w_j o_j$ of multiple objectives, or a more complex custom utility function based on one or multiple objectives. In the evaluation (Section 8.5), I consider examples of all three cases, detailed next.

A useful example for a *single optimization objective* is $U_T = o_f$ where o_f is the total amount of successful vs. dropped flows over T time steps:

$$o_f = \frac{|F_{\text{succ}}| - |F_{\text{drop}}|}{|F_{\text{succ}}| + |F_{\text{drop}}|} \in [-1, 1] \tag{8.1}$$

It encourages more successful flows F_{succ}, fewer dropped flows F_{drop}, and thus higher throughput.

In practice, operators are often interested in optimizing multiple, possibly competing objectives at once (e.g., for QoS, energy, costs). Here, the *weighted sum of objectives* $U_T = \sum_j w_j o_j$ is useful. I consider three additional objectives, each conflicting with o_f:

$$o_i = 2 \cdot \frac{-\sum_{c \in C, v \in V} x_{c,v}(t)}{|C| \cdot |V|} + 1 \in [-1, 1] \tag{8.2}$$

$$o_n = 2 \cdot \frac{-\sum_{v \in V} \mathbb{1}_{\{\sum_{c \in C} x_{c,v}(t) \geq 1\}}}{|V|} + 1 \in [-1, 1] \tag{8.3}$$

$$o_d = \max\left\{-1, \frac{-d_{\text{avg}}}{D_G} + 1\right\} \in [-1, 1] \tag{8.4}$$

Objective o_i minimizes the total number of placed instances (indicated by $x_{c,v}(t)$) to minimize costs, e.g., for licensing. Objective o_n minimizes the number of compute nodes used by at least one instance, where $\mathbb{1}_{\{\sum_{c \in C} x_{c,v}(t) \geq 1\}}$ is 1 if an instance is placed at node v and 0 otherwise. This may reduce costs and energy consumption if unused nodes are turned off, e.g., in an edge scenario. Objective o_d minimizes the average end-to-end delay d_{avg} per completed flow in time T for better QoS. If no flow is successful, d_{avg} is undefined, and I set $o_d = -1$. To ensure all objectives are in the same range $[-1, 1]$, I scale them by dividing by the respective maximum value. In o_d, I normalize d_{avg} with network diameter D_G in terms of delay. I further add 1 since $-d_{\text{avg}} \leq 0$ and cap any values below -1, which may occur if flows traverse the entire network multiple times back and forth due to bad service coordination (i.e., $d_{\text{avg}} > D_G$). I investigate trade-offs between these objectives and o_f as well as the impact of weights w_j in Section 8.5.4. Objectives o_f, o_i, o_n, and o_d are examples for typical optimization goals, but it is also possible to choose and optimize other objectives based on the desired goals and available monitoring information.

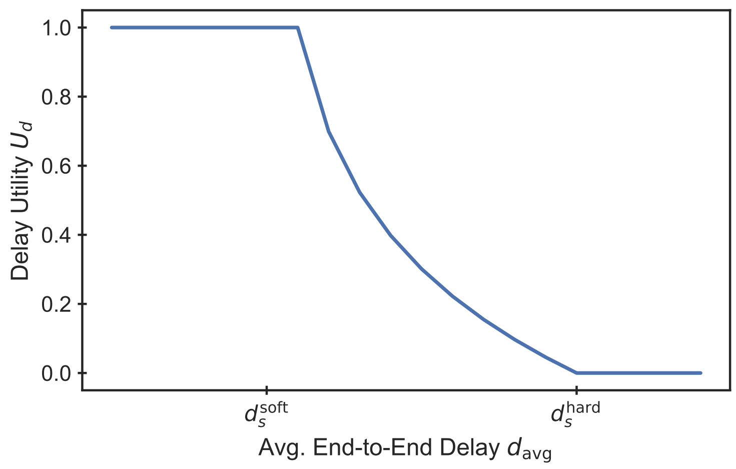


Figure 8.1.: DeepCoord supports complex custom utility functions, in the example here, for meeting soft deadlines. (Figure adapted from [S22]; © 2021 IEEE.)

Finally, as third and most generic option, DeepCoord allows defining and maximizing a *custom utility function* U_T based on any available monitoring information. This enables operators to define arbitrary, complex utility functions based on their business insights. As an example, I consider a custom utility function $U_T = U_f \cdot U_d \in [0, 1]$, where $U_f = \frac{|F_{\text{succ}}|}{|F_{\text{succ}}| + |F_{\text{drop}}|} \in [0, 1]$ is the flow success ratio and $U_d \in [0, 1]$ the delay utility shown in Figure 8.1. The delay utility is maximal for an end-to-end delay within soft deadline d_s^{soft} and then gradually diminishes with increasing delay up to hard deadline d_s^{hard}. In Section 8.5.4, I show that DeepCoord also learns to optimize such a custom utility function to meet soft deadlines for optimal QoS.

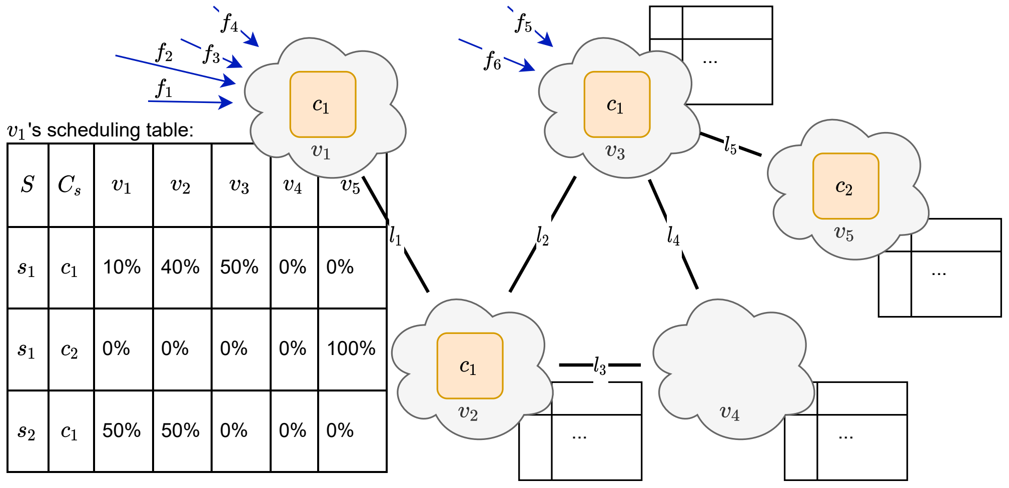
8.4. DeepCoord DRL Approach

I propose DeepCoord to address network and service coordination using model-free DRL. DeepCoord does not know the network topology, link delays, service or per-flow details. Instead, it relies on aggregated yet incomplete, slightly delayed, and uncertain information about incoming flows available through periodic monitoring (updated every Δ time steps). It learns network and service coordination without expert knowledge from its own experience.

I describe the coordination approach in Section 8.4.1 and formalize a POMDP in Section 8.4.2. Section 8.4.3 outlines the DRL coordination framework and algorithm.

8.4.1. Joint Scheduling, Scaling, and Placement

DeepCoord is designed to work in realistic, dynamic networks and to support many rapidly arriving flows (Case II in Section 3.1.3). Hence, making per-flow coordination decisions centrally at the DRL agent would be highly inefficient and not scalable for large numbers of flows. Moreover, it would require up-to-date, per-flow knowledge, which is not available centrally. Instead, each node schedules incoming flows immediately and locally according to rules that are installed at all nodes in the network. DeepCoord updates these rules every Δ time steps, whenever new monitoring data becomes available.



S	C_s	v_1	v_2	v_3	v_4	v_5
s_1	c_1	10%	40%	50%	0%	0%
s_1	c_2	0%	0%	0%	0%	100%
s_2	c_1	50%	50%	0%	0%	0%

Figure 8.2.: DeepCoord periodically updates scheduling tables that are deployed at each node in a distributed fashion. Flows continuously arrive at ingress nodes and are scheduled according to these scheduling tables at runtime. (Figure adapted from [S22]; © 2021 IEEE.)

To this end, I introduce *scheduling tables* for each node that indicate where incoming flows should be processed (similar to Amazon Web Services (AWS) traffic dials [14] but with dynamic rather than fixed quotas). As illustrated in Figure 8.2, each scheduling table contains entries for every service $s \in S$ and every corresponding component $c \in C_s$ (here, $S = \{s_1, s_2\}$, $C_{s_1} = \langle c_1, c_2 \rangle$, $C_{s_2} = \langle c_1 \rangle$). The table entries specify at which destination node to process c by means of a probability distribution over all nodes (not just neighbors). For example in Figure 8.2, incoming flows at node v_1 requesting component $c_1 \in C_{s_1}$ of s_1 are scheduled according to the probabilities in v_1's scheduling table. Here, each flow is processed locally at v_1 with 10 % probability, scheduled to be processed at v_2 with 40 %, and scheduled to v_3 with 50 %. Flows belonging to s_1 that finish processing c_1 at v_1 and are then requesting c_2 are all scheduled to v_5. I assume shortest path routing between nodes, e.g., from v_1 to v_5.

The same component c_1 also appears in service s_2, where it could require different scheduling. Hence, I consider separate scheduling entries for different services in S. If a component c is used multiple times within a single service, the same scheduling rules are applied for each appearance of c within the service, i.e., flows are scheduled to the same instances of c with the same probabilities at each position within C_s. If this is not desired, using distinct identifiers to distinguish different positions of c within the service (e.g., $c, c', c'', ...$) leads to separate instances and scheduling rules for each position. I assume service set S and components C_s to be rather static and contain all available services and their components, even if some services are not currently in use. Scheduling table entries for unused services are simply ignored. If sets S or C_s do change, e.g., because a completely new service s or component c is released, the scheduling tables and DeepCoord's neural networks have to be restructured to include s or c, respectively. In principle, the learned weights for $S \setminus s$ and $C_s \setminus c$ could be retained in the restructured neural networks.

By deploying these scheduling tables at each node, incoming flows are scheduled immediately (in $O(\log |V|)$) at runtime according to these probabilities. That means, $y_{f,c}(t)$ is set to v_i with probabilities given for v_i and $c \in C_{s_f}$ in a distributed manner. After deciding a destination node for processing a flow according to the scheduling probabilities, the entire flow is sent there. Using separate scheduling tables for each node allows to schedule flows differently depending on where they arrive in the network. In doing so, flows can be scheduled to close-by nodes, reducing end-to-end delay.

I also derive variable $x_{c,v}(t)$ for scaling and placement from the scheduling tables but update it only every Δ time steps. To avoid dropped flows, I ensure that instances of component c are available at every node v to which flows may be scheduled. Specifically, DeepCoord starts at the ingress nodes with the first component $c_1 \in C_s$ for each service s and sets $x_{c_1,v}(t) = 1$ if there is a non-zero probability for $y_{f,c_1}(t) = v$ based on the scheduling tables. Since scheduling probabilities sum up to one, DeepCoord always places at least one instance per service component to process incoming flows. It then continues in a similar fashion for the next component c_2, checking the scheduling tables of the nodes where instances of c_1 were placed. Based on v_1's scheduling table in Figure 8.2, DeepCoord would set $x_{c_1,v_1}(t) = x_{c_1,v_2}(t) = x_{c_1,v_3}(t) = x_{c_2,v_5}(t) = 1$.

For routing, DeepCoord uses predefined paths and sets $z_{f,v}(t)$ accordingly. Similar to BSP (Section 4.2.1), I precompute these paths using the Floyd-Warshall algorithm [82] for shortest paths with link weight $w_l = \frac{1}{\text{cap}_l + \frac{1}{d_l}}$, i.e., favoring paths with high capacity and low delay. Alternatively, paths could be selected using any other desired metric. Either way, the forwarding rules are deployed and applied directly at the nodes without DeepCoord being explicitly aware of these paths. Following this approach, DeepCoord can jointly decide flow scheduling probabilities, scaling, and placement by periodically (every Δ time steps) updating the scheduling tables for all nodes. In practice, these updates could be done consistently across the network using Software-Defined Networking (SDN) technology [225].

8.4.2. Partially Observable Markov Decision Process (POMDP)

In real networks, the full network state is huge and can only be observed partly through monitoring. Hence, I design a POMDP to create and periodically update the scheduling tables as described in Section 8.4.1. Using a POMDP is novel compared to related DRL approaches (Section 8.2), which mostly assume a fully observable Markov Decision Process (MDP). In a POMDP, an agent interacts with an environment to obtain rewards, which allows the agent to learn highly-rewarded behavior. Formally, a POMDP $(\mathcal{O}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ is defined by observation space $\mathcal{O}$, i.e., parts of the full network state, the agent's action space $\mathcal{A}$, the environment dynamics $\mathcal{P}$, which are typically unknown, and the reward function $\mathcal{R}$. In the proposed approach, described in Section 8.4.1, the agent interacts with the environment every Δ time steps. It receives observations from the last Δ time steps (e.g., through monitoring), applies an action to update the scheduling tables for the next Δ time steps, and, after these Δ time steps, receives a reward together with the next observation. I define $\mathcal{O}$, $\mathcal{A}$, and $\mathcal{R}$ as follows.

Observations $\mathcal{O} = \langle \lambda_{v,s} | v \in V, s \in S \rangle$, where $\lambda_{v,s}$ is the data rate summed up over all flows arriving at ingress node v and requesting service s, averaged over the previous interval of length Δ. If v is not an ingress node, $\lambda_{v,s} = 0$.

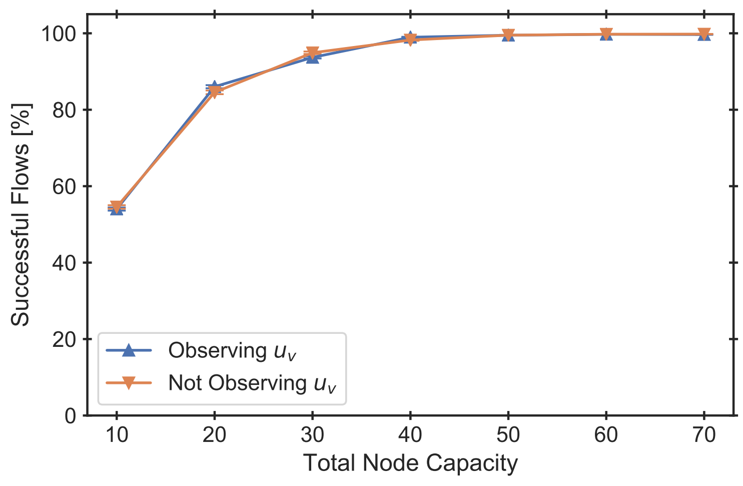


Figure 8.3.: Delayed observations of node load u_v are not useful for DeepCoord. (Figure adapted from [S22]; © 2021 IEEE.)

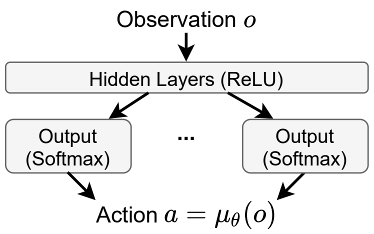
I also considered adding node load $u_v = \frac{r_v(t)}{\text{cap}_v} \in [0,1]$ at v over the last Δ time steps to the observations, where $u_v = 1$ if $\text{cap}_v = 0$. While observing u_v intuitively seems useful, I found that it usually does not improve DeepCoord's performance (Figure 8.3). In fact, the agent can completely change all scheduling rules with a single action, such that the observed node load u_v from the previous interval is often no relevant indication for the node load in the next interval, even with constant ingress data rates. This is especially true for scenarios with many, short flows (Case II), considered here. Even without this observation, the agent implicitly still learns about the network's capacities. It is rewarded for successful flows and punished for dropped flows when scheduling to nodes or links with insufficient capacity. Thus, I do not include u_v in observations $\mathcal{O}$ here to avoid unnecessary complexity for DeepCoord and to further reduce the requirements regarding monitoring. When applying DeepCoord to scenarios with very long flows that last for multiple intervals ($\delta_f > \Delta$; cf. Case I), observing u_v may become useful and can be included in the observations without requiring further changes of the approach.

Actions $\mathcal{A} = \langle p_{v,s,c,v'} | v, v' \in V, s \in S, c \in C_s \rangle$, where $p_{v,s,c,v'} \in [0,1]$ is the probability for scheduling a flow arriving at node v, requesting component c of service s to be processed at node v'. This results in a probability distribution with $\sum_{v' \in V} p_{v,s,c,v'} = 1$. As different scheduling probabilities are explored in the POMDP, it is unlikely that probabilities are set to exactly 0 %. To avoid sending small fractions of traffic to many nodes, I further process these probabilities as follows. I introduce a threshold p_{thres} and set all probabilities $p_{v,s,c,v'} < p_{\text{thres}}$ to 0 during post-processing. I then normalize each scheduling table row to ensure that the probabilities again sum up to 1, i.e., flows are still scheduled and processed at nodes other than v'. Finally, I apply these processed scheduling tables to the network and also use them for training (Section 8.4.3).

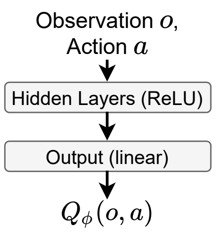
Reward $\mathcal{R} = U_\Delta$. Here, U_Δ can correspond to any of the three types of utility functions defined in Section 8.3, i.e., optimizing either an individual objective, a weighted sum of multiple objectives, or a custom utility function. In either case, the reward signal is computed based only on metrics collected in the last Δ time steps. In the example of $U_\Delta = o_f$, U_Δ would only consider the successful and dropped flows in the last monitoring interval Δ, which are most affected by the previous action. Internally, DeepCoord maximizes the sum of discounted rewards to optimize long-term utility. I evaluate DeepCoord with different utility functions for optimizing varying objectives in Section 8.5.4.

8.4.3. DRL Coordination Framework

DeepCoord is based on DDPG [155], which can handle large, continuous action spaces such as $\mathcal{A}$ in the POMDP, unlike previous algorithms like deep Q-learning [181]. DDPG is an off-policy actor-critic algorithm, i.e., it learns from buffered batches of previous experience using neural networks for both actor μ_θ and critic Q_ϕ. The critic approximates the long-term value $Q_\phi(o, a)$ of action a after observation o based on immediate reward r and expected future rewards. Critic Q_ϕ is used to train actor μ_θ. Actions produced by μ_θ represent the probabilities of each node's scheduling table, which should sum up to 1 for each row (Section 8.4.2). To this end, I split the output layer of μ_θ into separate parts for each row in each scheduling table and apply the softmax activation separately (Figure 8.4a). In the critic network, I do not apply any activation function to the output layer (Figure 8.4b).



(a) Actor μ_θ



(b) Critic Q_θ

Figure 8.4.: Neural network architecture of actor and critic. The actor's output layer is split into separate parts for each row in the scheduling tables, applying softmax separately.

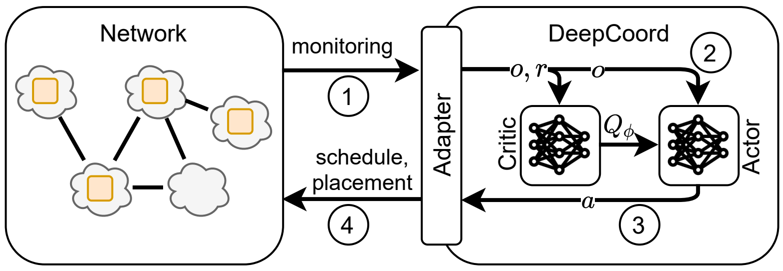


Figure 8.5.: In the proposed framework, the actor-critic DRL agent iteratively interacts with the network through an adapter. (Figure from [S22]; © 2021 IEEE.)

To ensure fast, consistent, and good service coordination, I first train DeepCoord offline until convergence and then apply the trained agent online (inference). Figure 8.5 shows the proposed framework for training and applying DRL for network and service coordination. The network provides monitoring data in regular intervals (step 1). In step 2, an adapter processes the monitoring information, retrieving the relevant observation o and calculating reward r for the previous interval as described in Section 8.4.2. With this architecture, DeepCoord can connect to different kinds of networks or monitoring systems (including simulation environments for training), simply by implementing a new adapter. In step 3, o and r are used to train critic Q_ϕ and actor μ_θ and to choose the next action a as defined in Section 8.4.2. During online inference, o is directly fed to actor μ_θ to quickly retrieve the next action without computing r and without any training. In step 4, the adapter uses a to compute the final scheduling tables for all nodes, derives the scaling and placement, and applies it to the network.

Algorithm 6 DeepCoord Training and Inference

```
 1: $k \leftarrow$ number of CPU cores available for training                ▷ Training
 2: for $k$ DRL agents in parallel do
 3:     Initialize $\mu_\theta, \mu_{\theta'}, Q_\phi, Q_{\phi'}, B$
 4:     for all $\Delta$ time steps $\in T$ do
 5:         $o, r \leftarrow$ adapter.process(monitoring)
 6:         $B \xleftarrow{\text{add}} (o_{\text{prev}}, a_{\text{prev}}, r, o)$
 7:         $b \leftarrow$ sample$(B, N)$
 8:         Train $Q_\phi$ minimizing the Bellman error [155]
 9:         Train $\mu_\theta$ maximizing $\mathbb{E}_o[Q_\phi(o, \mu_\theta(o))]$
10:         $Q_{\phi'} \leftarrow \tau Q_\phi + (1-\tau) Q_{\phi'}$
11:         $\mu_{\theta'} \leftarrow \tau \mu_\theta + (1-\tau) \mu_{\theta'}$
12:         $a \leftarrow \mu_\theta(o) + \mathcal{N}$
13:         Network $\xleftarrow{\text{apply}}$ adapter.process$(a)$
14: Select best trained agent $(\mu_\theta, Q_\phi)$                            ▷ Inference
15: for all $\Delta$ time steps $\in T$ do
16:     $o, r \leftarrow$ adapter.process(monitoring)
17:     $a \leftarrow \mu_\theta(o)$
18:     Network $\xleftarrow{\text{apply}}$ adapter.process$(a)$
```

Algorithm 6 shows the resulting DRL algorithm for training and inference. DDPG is known for its high training variance depending on the random seed [108]. To mitigate this problem, I propose to train k DRL agents in parallel with different random seeds, e.g., one per available CPU core (lines 1–2 in Algorithm 6). After training, the best agent can be selected automatically based on the achieved reward. During training, new experience is added to the buffer B and batches b of size N are sampled to train critic and actor (lines 5–9). For training stability, target critic $Q_{\phi'}$ and actor $\mu_{\theta'}$ are updated slowly according to τ (lines 10–11). Then, the next action a is selected using the trained actor and adding Gaussian noise $\mathcal{N}$ to encourage exploration (line 12). Finally, a is post-processed as described in Sections 8.4.1 and 8.4.2 to derive the final scheduling, scaling, and placement decisions (line 13). I store the processed and actually applied actions in buffer B for training. After training, I use the best trained agent for fast inference during online network and service coordination (line 14). New observations are directly passed to the trained actor μ_θ to obtain the next action (line 17). For best performance during inference, I do not add noise but exploit the best action. The selected action is then post-processed and applied to the network as before (line 18).

Offline training of DeepCoord is time-intensive and depends on random exploration. In contrast, online inference is very fast [107]. Its complexity is defined by the matrix multiplication of the observations and neural network weights, which depend on the observation and action spaces (Section 8.4.2). I empirically evaluate training and inference complexity for varying network sizes in Section 8.5.5.

8.4.4. Implementation and Deployment

A Python prototype of DeepCoord is publicly available as open source [S25]. DeepCoord's actor and critic neural networks are built with Keras [100] and TensorFlow [4]. DeepCoord follows the common OpenAI Gym interface [32] and interacts with the network through an adapter that collects and computes the required observations and the reward. The adapter implementation depends on the specific network environment. For training and evaluation, I use the lightweight open-source simulator `coord-sim` [S26]. For production deployment, systems like Prometheus [204] and Kubernetes [45] could be interfaced by the adapter for centralized monitoring and orchestration. Scheduling rules could be installed and applied in a distributed fashion at all nodes using SDN [257]. The location of the orchestrator and SDN controller could be optimized using established approaches [145, 280] to minimize the overhead of collecting monitoring data and updating scheduling rules in very large scenarios.

8.5. Evaluation

I evaluate DeepCoord through extensive simulations using real-world network topologies and realistic traffic patterns. I describe the details of the evaluation setup in Section 8.5.1. In Section 8.5.2, I evaluate how well DeepCoord self-adapts to scenarios with varying load, traffic patterns, node and link capacities, and QoS requirements. For each scenario, I train DeepCoord offline from scratch in a training environment and then evaluate the trained agent in a separate testing environment with different random seeds. The agent adapts to each scenario through self-learning without human intervention or expertise, simply from experience when interacting with each environment. Section 8.5.3 investigates DeepCoord's generalization capabilities, where it is trained on one scenario and then tested on other new and unseen scenarios without any additional training. While the aforementioned experiments focus on optimizing a single optimization objective, Section 8.5.4 explores trade-offs when optimizing multiple competing objectives. Finally, Section 8.5.5 evaluates the scalability of DeepCoord to large, real-world network topologies.

8.5.1. Evaluation Setup

Evaluation Scenarios

I perform extensive simulations on real-world network topology Abilene [142], which connects nodes at 11 cities across the United States. In Section 8.5.5, I also evaluate scalability on three larger real-world network topologies from Europe, China, and across continents, taken from the Internet Topology Zoo [142]. These topologies contain information about the position and interconnection of the network nodes but not about their type (e.g., data center or small edge server) nor about their compute capacity. Since DeepCoord does not distinguish between different node types and is only affected by a node's capacity cap_v, I assign heterogeneous node capacities cap_v between 0 and 2 compute units (e.g., Central Processing Unit (CPU) cores) uniformly and independently at random. By default, I use very high link capacities $\text{cap}_l = 1000$ but also consider scenarios with more restricted link capacities (Section 8.5.2). Link delays are based on the distance between connected nodes. While I successfully tested DeepCoord with multiple services, for simplicity, I here focus on and show the results of coordinating a single service s with components $C_s = \langle c_{\text{IDS}}, c_{\text{proxy}}, c_{\text{web}} \rangle$. Instances of each component require resources that increase linearly with increasing total data rate of flows to process. Again, DeepCoord is neither explicitly aware of components' resource requirements nor limited to linear resource requirements. Instead, it learns service characteristics (including resource requirements) implicitly through feedback from its actions. I assume all flows requesting this service to have unit data rate ($\lambda_f = 1$) and flow length ($\delta_f = 1$) but consider scenarios with increasingly complex and realistic flow arrival patterns.

In the evaluation, flows arrive over $T = 20000$ time steps according to different traffic patterns at the network's ingress nodes. Ingress nodes are selected randomly per network and do not change over time. I further set $\Delta = 100$ time steps, after which DeepCoord receives new observations and applies actions. As described in Section 8.4, this means that information in observations may be delayed by up to 100 time steps. This is more realistic than the common assumption in related work of having up-to-date information at each time step.

DRL Hyperparameters

For each scenario, I first train $k = 10$ DRL agents in parallel until convergence (500 episodes). Then, I automatically select the best DRL agent for inference (Section 8.4.3). I train DeepCoord from scratch for each scenario but configure fixed values for all hyperparameters that are used across all scenarios. Thus, no manual adjustments are required for solving different scenarios with DeepCoord.

For both actor and critic, I train dense neural networks with a single fully connected hidden layer (64 hidden units, Rectified Linear Unit (ReLU) activation function [93]) using the Adam optimizer [138]. I further configured the following hyperparameters:

- Discount factor $\gamma = 0.99$.
- Soft target updates with $\tau = 0.0001$.
- Learning rate $\alpha = 0.01$ with decay 0.001.
- Buffer size $|B| = 10000$ with batch size $|b| = 64$.
- For exploration, I use Gaussian noise with $\mathcal{N}(0, 0.2)$.
- Threshold $p_{\text{thres}} = 0.1$ (Section 8.4.2).

Compared Algorithms

I compare DeepCoord against the following algorithms:

- *BSP* from Chapter 4 as an example for a state-of-the-art conventional heuristic, which jointly optimizes service scaling and placement as well as flow scheduling.
- *Shortest Path Coordination Approach (SP)*: For each ingress, SP places exactly one instance per component $c \in C_s$. It follows a simplified first-fit strategy by instantiating the first component at the ingress node and each following component at the neighbor closest to the previous instance. In doing so, SP favors nodes with fewer existing instances and skips nodes without any compute capacity (independent of current utilization).
- *Load Balancing Coordination Approach (LB)*: LB instantiates all components at all nodes with non-zero capacity and schedules flows equally.

SP and LB are similar to the baselines used by Xu et al. [265]. All three algorithms choose actions from action space $\mathcal{A}$—but, unlike DeepCoord, do not learn from these actions.

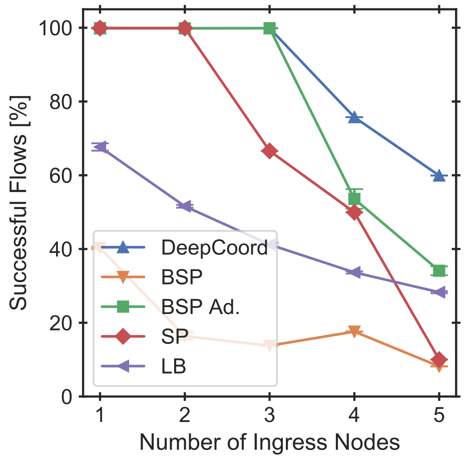
Directly applying BSP to the scenarios considered here works poorly. BSP focuses on scenarios with few and long flows (Case I in Section 3.1.3), where all flows run in parallel and compete for resources. In the scenarios considered here, many, short flows arrive sequentially at each ingress and only overlap partially (Case II). For a fair comparison, I adjusted the input processing of BSP to estimate the overlapping flows per Δ time steps. This is an example of how built-in assumptions limit the applicability of a conventional, model-based algorithm, requiring manual adjustments by experts. I show both the default and adapted version of BSP in the evaluation. Unlike DeepCoord, related DRL approaches (Section 8.2) are not available publicly. Thus, a direct comparison is difficult.

Execution & Figures

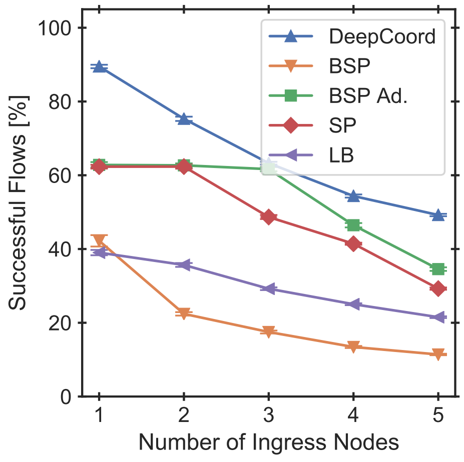
I repeated all experiments with 30 different random seeds on machines with an Intel Xeon W-2145 CPU and 32 GB RAM. The figures in this section show the mean over these 30 repetitions. The error bars depict the 95 % confidence interval.

8.5.2. Self-Adaptation to Varying Scenarios

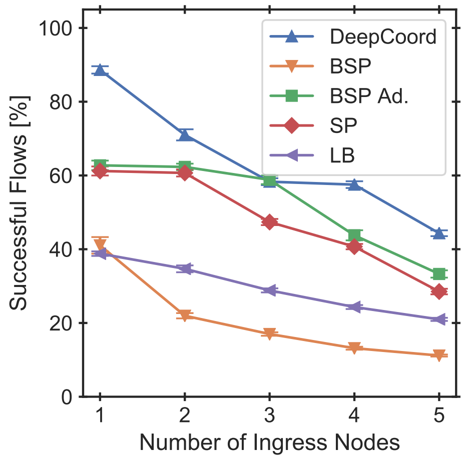
Here, I focus on maximizing the number of successful flows ($U_T = o_f$; Section 8.3) in the Abilene network. I systematically vary different problem parameters (traffic, capacities, QoS requirements) and compare the percentage of successful flows after T time steps for each algorithm. I train DeepCoord from scratch for each scenario to evaluate how well it adapts itself to these scenarios.



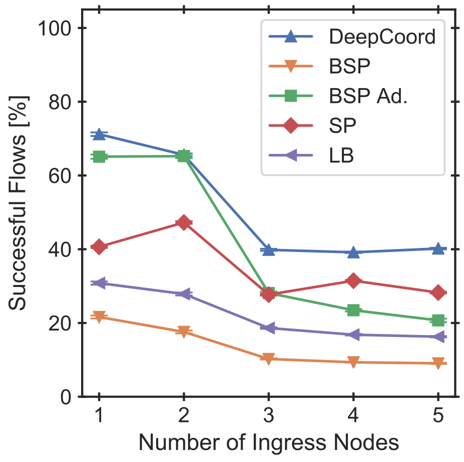
(a) Fixed arrival



(b) Poisson arrival



(c) MMPP arrival



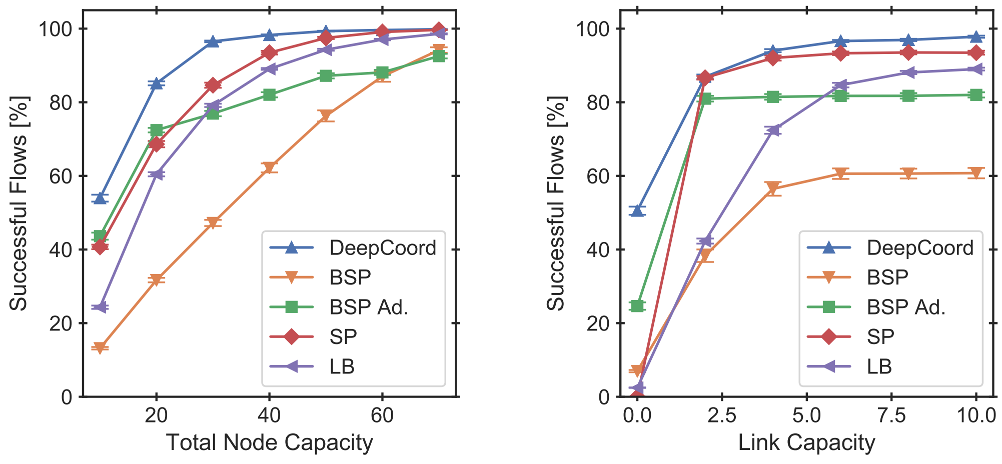
(d) Real-world traffic trace

Figure 8.6.: Percentage of successful flows for an increasing number of ingress nodes, i.e., increasing load, and increasingly realistic flow arrival patterns. Compared to other approaches, DeepCoord processes most flows successfully. (Figure adapted from [S22]; © 2021 IEEE.)

Varying Ingress Nodes and Traffic Patterns

First, I vary the number of ingress nodes from 1 to 5 and choose increasingly complex flow arrival patterns. With more ingress nodes, total traffic increases and the network's capacities become saturated such that flows have to be dropped (Figure 8.6).

The simplest traffic pattern I consider is *fixed flow arrival*, where flows arrive in fixed intervals (10 time steps) at each ingress. Figure 8.6a shows the percentage of successful flows for the different algorithms. As described in Section 8.5.1, default BSP performs poorly, but the manually adapted BSP ("BSP Ad.") does much better and only drops flows with more than 3 ingress nodes. SP avoids dropped flows up to 2 ingress nodes and LB always drops many flows. DeepCoord outperforms all other algorithms, processing much more flows successfully in the highly saturated network with 4 and 5 ingress nodes (up to 76 % more successful flows than adapted BSP). Figure 8.6b shows the results for *Poisson flow arrival* (mean inter-arrival time 10 time steps). Due to the randomness in flow arrival, multiple flows may arrive directly after another in bursts, which can easily lead to dropped flows. Again, adapted BSP is slightly better than SP and much better than default BSP. LB still performs worse than the other algorithms and DeepCoord still outperforms all algorithms. Compared to adapted BSP, DeepCoord processes up to 43 % more flows. In particular, it learns to deal with Poisson flow arrival by not fully utilizing all resources of a node but leaving some resources free for handling small bursts.



(a) Varying node capacity (b) Varying link capacity

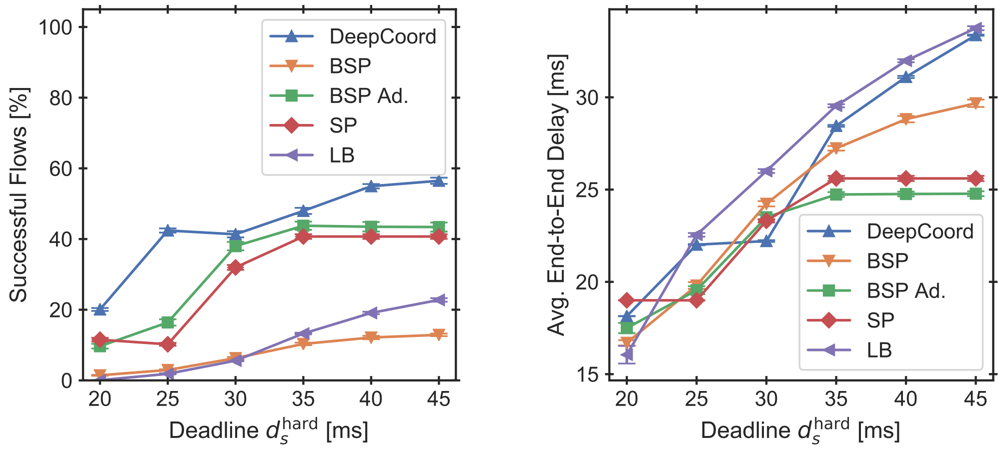
Figure 8.7.: DeepCoord leverages increasing node and link capacities to process more flows successfully, outperforming other approaches. (Figure adapted from [S22]; © 2021 IEEE.)

Next, I consider more realistic flow arrival following a *Markov-Modulated Poisson Process (MMPP)* [80]. The two-state Markov process randomly switches between flow arrival with mean inter-arrival time 12 and 8 (50 % higher rate) every 100 time steps with 5 % probability. Figure 8.6c shows that DeepCoord handles MMPP flow arrival well and, again, outperforms the other algorithms (up to 41 % better than adapted BSP). Finally, Figure 8.6d shows the results for flows following *real-world traffic traces* that were recorded for the Abilene network [194]. To simulate increasing load, I enable an increasing number of ingress nodes where the trace-driven traffic arrives. Here, adapted BSP is better with low load (1–2 ingress nodes) and SP is better with high load (4–5 ingress nodes). DeepCoord handles this real-world traffic well and again outperforms all other algorithms (up to 43 % better than adapted BSP and SP).

Varying Node and Link Capacities

I investigate self-adaptation of DeepCoord to scenarios with varying node and link capacity, again, training from scratch for each scenario. Specifically, I consider MMPP traffic with 4 ingress nodes on the Abilene network, starting with a total node capacity of 10 compute units (as before) and then evenly increasing node capacity. Figure 8.7a shows that DeepCoord adapts well to different node capacities and clearly outperforms all other algorithms. In comparison, the other algorithms need at least 67 % more resources to reach high success rates of above 95 %.

Next, I consider scenarios with limited link capacity, varying cap_l from 0 to 10 for each link l. In contrast to all other scenarios, where nodes' compute resources were the bottleneck, here, communication between nodes is the bottleneck. Such communication-constraint scenarios are not the main focus of DeepCoord, which uses fixed (shortest) paths when scheduling flows between nodes rather than routing them dynamically. Still, DeepCoord achieves more successful flows than the other algorithms (Figure 8.7b) and is the only approach to achieve success rates of more than 95 %. This indicates that, even without dynamic routing, it successfully adapts to varying link capacities by distributing load across nodes and corresponding paths.



(a) Successful flows (b) End-to-end delay

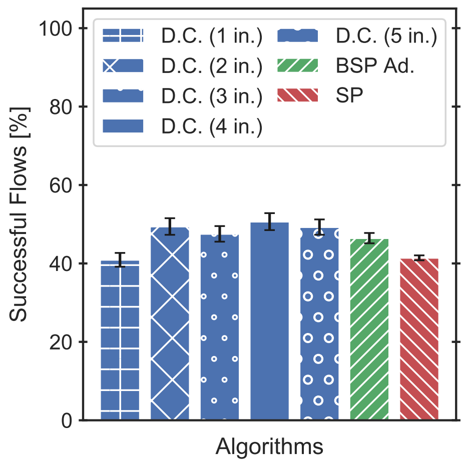
Figure 8.8.: DeepCoord self-adapts to hard deadlines. It leverages higher deadlines to better distribute load and achieve higher flow success rates. (Figure adapted from [S22]; © 2021 IEEE.)

Varying QoS Requirements (Hard Deadlines)

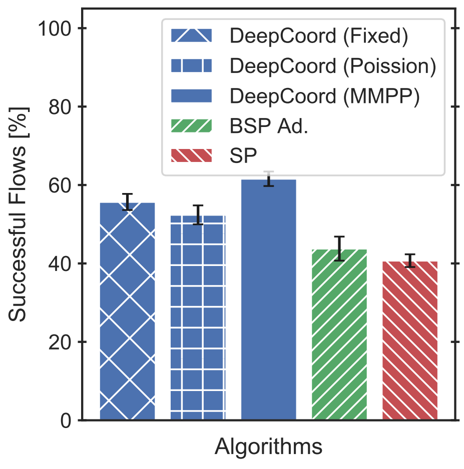
Here, I explore scenarios with MMPP traffic, 4 ingress nodes, and varying QoS requirements in terms of hard deadlines d_s^{hard}. As flows are dropped automatically if a hard deadline is not met, completing flows that exceed the deadline is not possible. DeepCoord implicitly learns that flows are dropped when their end-to-end delay is too high—without requiring any explicit knowledge about these deadlines. Figure 8.8 shows that DeepCoord exploits increasing deadlines by distributing flows to nodes farther away. This leads to an increasing success rate (Figure 8.8a) and results in higher end-to-end delay within the allowed deadline (Figure 8.8b). Similarly, LB exploits increasing deadlines and tries to balance load across all nodes as far as possible with the given deadlines. Still, it achieves much lower success rates than DeepCoord. In contrast, adapted BSP and SP do not exploit deadlines beyond 35 ms to increase success rates, leading to an increasing gap compared to DeepCoord.

8.5.3. Generalization to Unseen Scenarios

For the different scenarios of Section 8.5.2, I always train DeepCoord from scratch but reuse the same hyperparameter settings. This allows to fully automate training and applying DeepCoord to different scenarios. In practice, a trained agent still needs to perform reasonably well when facing a new scenario, e.g., due to changes in load or traffic. Training a new agent optimized for the new scenario can take hours, during which the old agent is still being used. To support such generalization, I define observations and reward based on generally-available information and normalize observations, actions, and rewards to be in a similar range (Section 8.4.2).



(a) Unseen ingress nodes



(b) Unseen traffic pattern

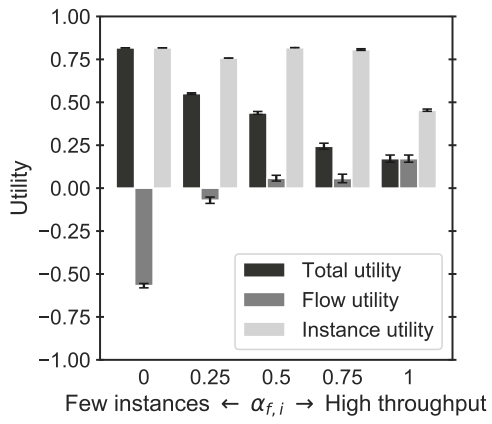
Figure 8.9.: DeepCoord generalizes to new scenarios with previously unseen a) ingress nodes and b) traffic patterns. (Figure adapted from [S22]; © 2021 IEEE.)

I investigate generalization of DeepCoord to scenarios with unseen load. In particular, I train five different DeepCoord agents on MMPP traffic with 1–5 ingress nodes, respectively, representing scenarios with low to high load. Without any additional training, I then test all five DeepCoord agents on MMPP traffic with 4 ingress nodes. Figure 8.9a shows that, as expected, the DeepCoord agent trained and tested on 4 ingress nodes (abbreviated as "D.C. (4 in.)") performs best. However, also the agents trained on 2, 3, and 5 ingress nodes generalize well to unseen traffic from 4 ingress nodes. In fact, they achieve a similar percentage of successful flows as the agent trained and tested on 4 ingress nodes and still outperform adapted BSP and SP. Only the agent trained on a single ingress node leads to slightly worse results when generalizing to much higher load with 4 ingress nodes. Still, its performance is comparable with SP's.

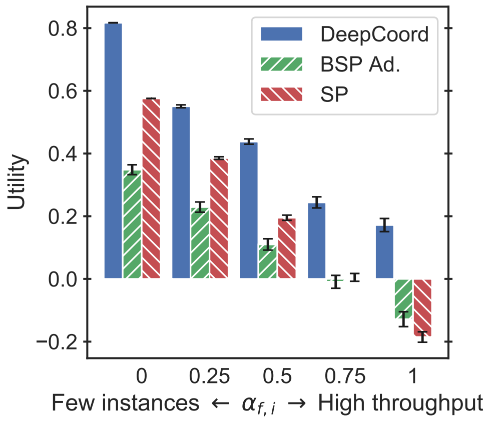
I also study generalization of DeepCoord to scenarios with new traffic patterns. Specifically, I train one agent on fixed flow arrival and another on Poisson flow arrival and confront both with previously unseen MMPP flow arrival. Figure 8.9b shows the successful flows for both cases in the Abilene network with 4 ingress nodes. For comparison, I also show results of DeepCoord trained on MMPP traffic and of adapted BSP and SP. The figure indicates that DeepCoord generalizes well to new traffic patterns without significantly reduced successful flows. The generalized agents still outperform adapted BSP and SP.

8.5.4. Optimizing Multiple Objectives

In Sections 8.5.2 and 8.5.3, I focus on maximizing successful flows (o_f) as the only optimization objective. In practice, operators often want to optimize multiple objectives. Here, in Section 8.5.4, I evaluate network and service coordination with multiple objectives, investigating the trade-off between maximizing successful flows and objectives o_i, o_n, and o_d, defined in Section 8.3. I also consider a more complex, custom objective function for supporting soft deadlines in addition to hard deadlines in Section 8.5.4. In all cases, I consider MMPP traffic with 4 ingress nodes on the Abilene network.



(a) DeepCoord's total, flow, and instance utility for trade-off $\alpha_{f,i}$

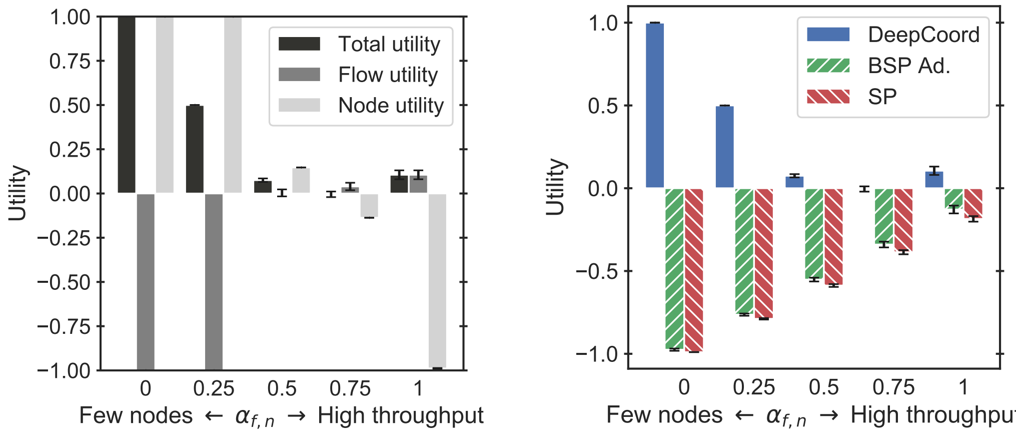


(b) Algorithms' total utility

Figure 8.10.: DeepCoord effectively navigates trade-off $\alpha_{f,i}$ between placing fewer instances and higher throughput. (Figure adapted from [S22]; © 2021 IEEE.)

Weighted Sum of Objectives

First, I consider utility $U_T = w_f o_f + w_i o_i$ as weighted sum of maximizing successful flows (objective o_f) and minimizing the number of placed instances (objective o_i). The two objectives are often conflicting as maximizing successful flows may require balancing the load across more instances. I explore this trade-off by systematically varying $\alpha_{f,i} = w_f = 1 - w_i \in [0,1]$ and training DeepCoord for each setting. Figure 8.10a shows the results in terms of flow utility o_f, instance utility o_i, and total, weighted utility U_T. Clearly, $\alpha_{f,i}$ affects how DeepCoord coordinates services. As desired, higher $\alpha_{f,i}$ leads to better flow utility. At the same time, the agent manages to maintain high instance utility, i.e., placing few instances. Only for $\alpha_{f,i} = 1$, the agent places more instances (i.e., lower instance utility) for even higher flow utility. In the given scenario, there are only enough resources to process some but not all flows successfully ($o_f < 1$) such that the total utility decreases with increasing $\alpha_{f,i}$. Compared to the other algorithms, DeepCoord achieves significantly better total utility for all $\alpha_{f,i}$ values (Figure 8.10b), indicating that it learns to navigate this trade-off well.

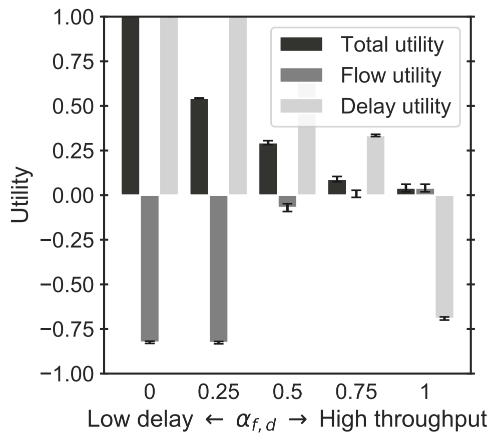


(a) DeepCoord's total, flow, and node utility for trade-off $\alpha_{f,n}$

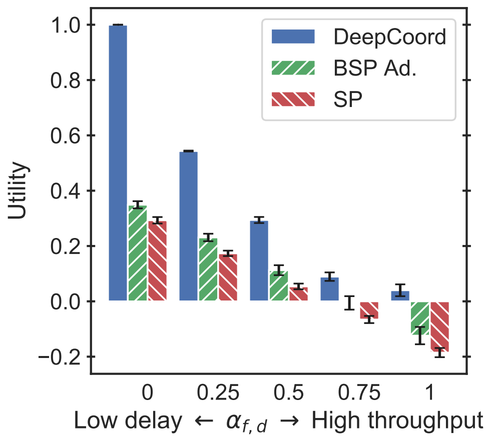
(b) Algorithms' total utility

Figure 8.11.: DeepCoord effectively navigates trade-off $\alpha_{f,n}$ between utilizing fewer nodes and achieving higher throughput. (Figure adapted from [S22]; © 2021 IEEE.)

Second, I evaluate the trade-off between maximizing successful flows and minimizing the number of used compute nodes with $U_T = w_f o_f + w_n o_n$ and $\alpha_{f,n} = w_f = 1 - w_n \in [0,1]$. A compute node can be turned off if no instances are placed there, saving costs and energy. Hence, to turn off a node v and to optimize o_n, DeepCoord has to learn to select actions that do not schedule any traffic (below threshold p_{thres}) to any instance at v. Compared to minimizing the number of instances (objective o_i), optimizing o_n is more challenging since DeepCoord is not rewarded for removing a single instance as long as there are still other instances placed at the same node. Still, Figure 8.11a shows that DeepCoord explores actions effectively and does learn different coordination schemes depending on $\alpha_{f,n}$. With low $\alpha_{f,n}$ (0 or 0.25), DeepCoord drops all flows ($o_f = -1$) in favor of optimal node utility ($o_n = 1$). As desired, with higher $\alpha_{f,n}$, it learns to process more flows successfully at the cost of utilizing more nodes. Compared to the other algorithms, its overall utility is much higher for all values of $\alpha_{f,n}$ (Figure 8.11b). Hence, while $\alpha_{f,n}$ has to be chosen carefully, DeepCoord can successfully optimize both objectives and navigate the trade-off as controlled by $\alpha_{f,n}$.



(a) DeepCoord's total, flow, and delay utility for trade-off $\alpha_{f,d}$



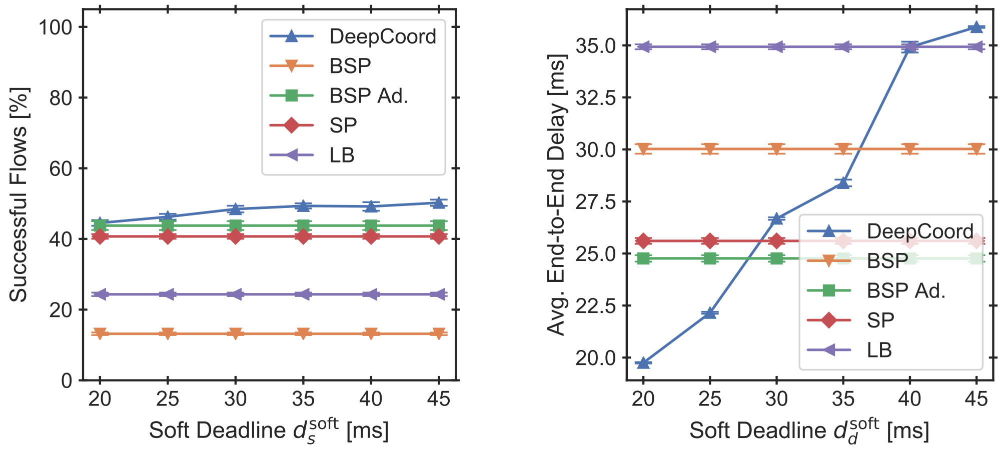
(b) Algorithms' total utility

Figure 8.12.: DeepCoord effectively navigates trade-off $\alpha_{f,d}$ between lower delay (thus, better QoS) and higher throughput. (Figure adapted from [S22]; © 2021 IEEE.)

Finally, I consider the trade-off between maximizing successful flows and minimizing average end-to-end delay with $U_T = w_f o_f + w_d o_d$ and $\alpha_{f,d} = w_f = 1 - w_d \in [0, 1]$. Again, o_f and o_d are often opposing objectives as processing more flows successfully may require scheduling them to nodes farther away, i.e., with higher path and end-to-end delay. Again, Figure 8.12a shows that DeepCoord learns different coordination schemes corresponding to $\alpha_{f,d}$. As desired, higher $\alpha_{f,d}$ leads to better flow but worse delay utility. With decreasing $\alpha_{f,d}$, it favors shorter delays at the cost of more dropped flows. The agents trained with $\alpha_{f,d} = 0$ and $\alpha_{f,d} = 0.25$ drop most flows but process the remaining successful flows with optimal delay. Again, DeepCoord achieves significantly better total utility for all $\alpha_{f,d}$ values than the other algorithms (Figure 8.12b). Overall, DeepCoord learns to effectively optimize multiple, even competing objectives, where the trade-off between these objectives can be controlled conveniently via weights $\alpha_{f,i}$, $\alpha_{f,n}$, and $\alpha_{f,d}$.

Custom Utility Function (Soft Deadlines)

In addition to optimizing individual objectives or weighted sums of multiple objectives, DeepCoord also supports optimizing custom utility functions. These functions may specify any complex relationship between available monitoring information and resulting utility. As an example, I here consider the custom utility function defined in Section 8.3. There, the total utility depends on the successful flows and their delay in terms of meeting QoS requirements. The utility is high as long as flows' end-to-end delay is below a given soft deadline and then gradually drops off until a hard deadline is reached (Figure 8.1). Unlike the scenario with hard deadlines in Section 8.5.2, flows are not dropped automatically if they exceed their soft deadline. Hence, when only optimizing objective o_f, DeepCoord would be unaware of and not adapt to these soft deadlines.



(a) Successful flows (b) End-to-end delay

Figure 8.13.: DeepCoord self-adapts to custom utility functions, here the utility function in Figure 8.1 for soft deadlines. As a result, it learns to respect soft deadlines and leverages increasing deadlines to improve the flow success rate. (Figure adapted from [S22]; © 2021 IEEE.)

Instead, when optimizing the custom utility function, Figure 8.13 shows that DeepCoord does successfully learn to take these soft deadlines into account. It not only outperforms the other algorithms in terms of successful flows (Figure 8.13a) but also adapts to ensure the average end-to-end delay stays below the given soft deadline to maximize utility and QoS. In turn, it exploits higher soft deadlines to process more flows successfully (15 % more with $d_s^{\text{soft}} = 45\,\text{ms}$ compared to $d_s^{\text{soft}} = 20\,\text{ms}$). The other algorithms do not support QoS optimization with custom utility functions and are unaware of the soft deadlines. Hence, they do not adapt to varying soft deadlines and, overall, process fewer flows successfully than DeepCoord.

8.5.5. Scalability

Finally, I evaluate DeepCoord's scalability to large-scale networks with many nodes. In addition to Abilene (11 nodes), I consider real-world network topologies BT Europe (24 nodes), China Telecom (42 nodes), and TiNet (53 nodes) [142], each with 4 ingress nodes and MMPP flow arrival (as defined in Section 8.5.2).

Figure 8.14 shows DeepCoord's learning curves, i.e., the average reward per training episode when training DeepCoord offline. As action noise enforces exploration, the reward is much lower during training than when testing the trained agent. Still, the rapid growth of episode reward within the first 100 episodes indicates that DeepCoord quickly learns a good coordination policy.

The figure also shows that more training may still increase performance significantly, e.g., the reward for 42 nodes leaps around episodes 300 and 600. With much more training, I expect further leaps in performance. Especially large networks require excessive training to explore the large action space and to find an optimal policy. The need for excessive training is not specific

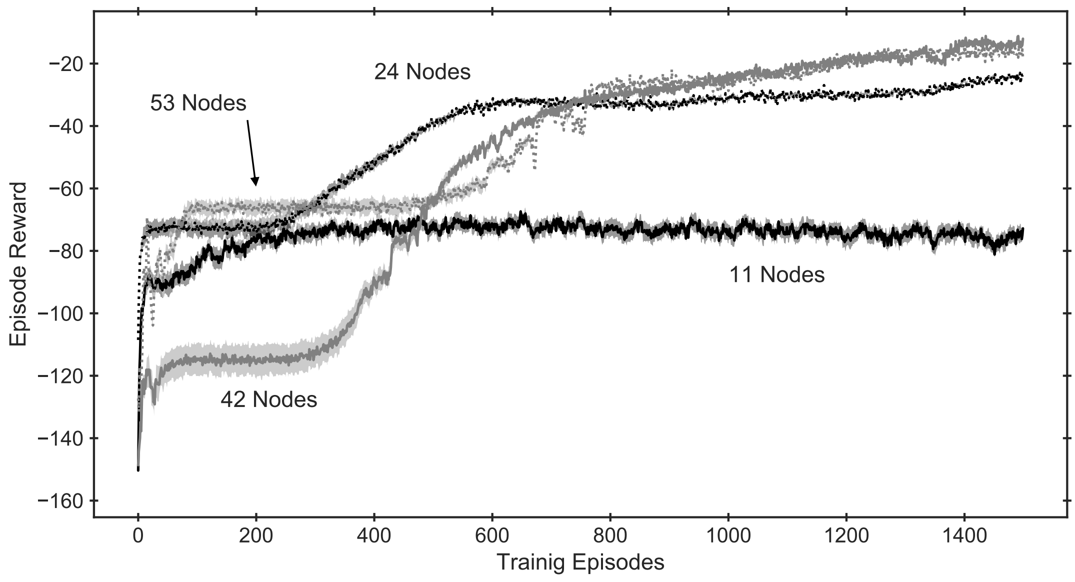
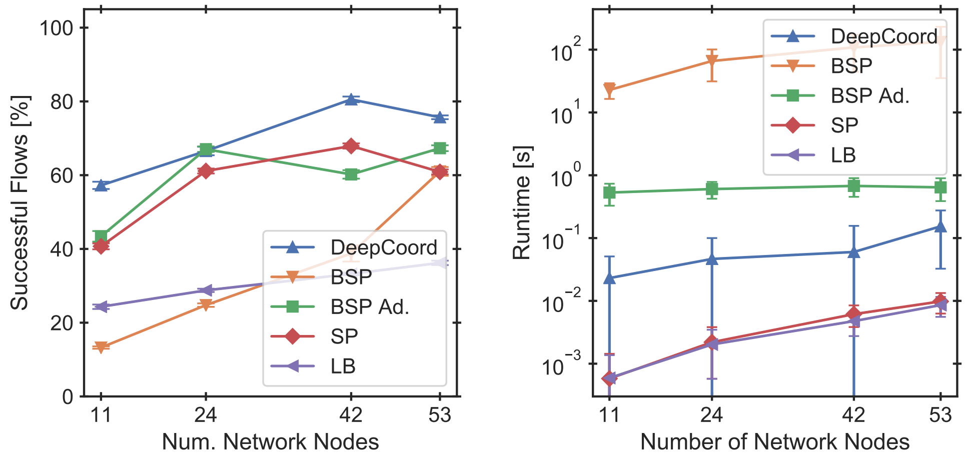


Figure 8.14.: Learning curves of DeepCoord when training on networks of varying sizes. (Figure adapted from [S22]; © 2021 IEEE.)

to my approach but a well-known problem in deep learning [241]. E.g., DeepMind's famous AlphaGo Zero was trained over almost 5 million games [236]. Due to limited time and resources, I had to restrict training to 1500 episodes (each with $T = 20000$). One option to improve training efficiency is with distributed DRL, which I explore in the following Chapter 9.

Despite limited offline training, DeepCoord can compete with or even outperform all baseline algorithms. Figure 8.15a compares the algorithms' percentage of successful flows (optimizing only objective o_f). As before, adapted BSP performs comparable to SP and considerably better than the default BSP version. LB performs worse on small networks but processes an increasing number of flows successfully with increasing network size. This is because LB balances traffic equally among all nodes with resources, leading to lower load per node and more successful flows for larger networks. Still, DeepCoord processes as many or even more flows successfully. The difference is especially large for 42 nodes, where there was a particularly large leap in performance during training (Figure 8.14). I believe that considerably more training (e.g., in a commercial setting) could result in similar performance leaps and even better results for 24 and 53 nodes.

In addition to quality, the runtime of online coordination decisions is crucial to quickly adapt to changes. Figure 8.15b shows the algorithms' average runtime per coordination decision on a logarithmic scale. While offline training is slow, applying the trained DRL agent for online inference only requires tens of milliseconds and is much faster than the (adapted) BSP heuristic. Compared to DeepCoord, the simple SP and LB baselines are even faster but at the cost of reduced coordination performance. Overall, DeepCoord scales well to large networks while maintaining reasonable runtimes.



(a) Successful flows (b) Algorithm runtime

Figure 8.15.: Even in large networks, DeepCoord processes a) more flows successfully than existing approaches and b) maintains short inference runtimes (logarithmic scale). (Figure adapted from [S22]; © 2021 IEEE.)

8.6. Conclusion

The proposed DRL approach, DeepCoord, goes beyond the machine learning approach of Chapter 7, which requires integration with an existing coordination approach (e.g., from Part I). Instead, DeepCoord is a standalone coordination approach and learns autonomous network and service coordination entirely from interaction with the network environment. In contrast to existing approaches using a priori knowledge for planning, it relies on realistically available, partial, and delayed observations with uncertain future traffic and without knowledge of network topology or service structure. It learns without human intervention or expertise and flexibly adapts to different scenarios or optimization objectives, even supporting multiple opposing objectives such as throughput, costs, energy, and QoS. Hence, I believe the approach is an important step towards truly driver-less, self-learning networks and thus towards higher efficiency, more flexibility, and improved reliability.

The following Chapter 9 further extends the idea of self-learning network and service coordination and proposes a distributed coordination approach using DRL. Its distributed architecture allows more fine-grained control and even better scalability and applicability to large, real-world networks.

9. Self-Learning Distributed Coordination

Conventional approaches for network and service coordination (e.g., in Part I) typically build on rigid models and assumptions and require expert knowledge or even a priori knowledge. In contrast, self-learning approaches using Deep Reinforcement Learning (DRL) such as DeepCoord from Chapter 8 learn without expert or a priori knowledge and self-adapt to varying scenarios, making these approaches more flexible and better applicable in practice. Existing self-learning approaches are promising but still have limitations as they often only address simplified versions of the network and service coordination problem and are typically centralized and thus may not scale to practical, large-scale networks. To address these issues, I propose a distributed and self-learning coordination approach using DRL. After centralized training, a distributed DRL agent is deployed at each node in the network, making fast coordination decisions locally in parallel with agents at other nodes. Each agent only observes its direct neighbors and does not need global knowledge. Hence, the approach scales independently from the size of the network. In the extensive evaluation using real-world network topologies and traffic traces, I show that the proposed approach outperforms a fully distributed conventional heuristic from Chapter 6 as well as the centralized DRL approach from Chapter 8 (60 % higher throughput on average) while requiring less time per online decision (1 ms).

This chapter is based on my paper [S34], which relies on concepts, code, and evaluation results created by Haydar Qarawlus for his master's thesis [207]. I proposed the initial topic and idea and closely advised the thesis through written feedback and weekly discussions. Based on the good outcomes of the thesis, we further improved the approach and extended the evaluation as part of the RealVNF project [S20]. I led the RealVNF project, guided the improvements and extensions, and wrote the published paper. This chapter contains verbatim copies of this paper [S34]: Stefan Schneider, Haydar Qarawlus, and Holger Karl. "Distributed Online Service Coordination Using Deep Reinforcement Learning." In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2021, © 2021 IEEE. In this chapter and throughout my dissertation, I consistently write in the first-person singular form ("I" rather than "we") for ease of reading. The source code corresponding to this chapter is publicly available on GitHub [S11].

9.1. Introduction

Existing work on network and service coordination typically has some major limitations as discussed in Section 8.1: Existing approaches mostly focus on long-/medium-term planning based on estimated user demand for an upcoming time interval. Most existing work also proposes conventional coordination approaches (such as in Part I), which are tailored to specific scenarios by experts and built on rigid models and assumptions. Operational reality easily diverges from such constructed scenarios and from estimated user demand, e.g., flows typically arrive stochastically without following any rigid pattern or with unexpected distributions.

Then, assumptions may no longer hold and the approaches could perform much worse than anticipated or even break completely.

Hence, self-learning approaches have been proposed recently for network and service coordination. These approaches mainly use DRL or contextual bandits to learn coordination directly from experience rather than following hand-written rules. The hope is that, in this way, they can self-adapt to new scenarios or optimization objectives without human intervention or expertise. Most existing DRL approaches, however, consider either complementary or simplified subproblems of the full network and service coordination problem. As described in Chapter 3, network and service coordination involves joint scaling, placement, scheduling, and routing. Furthermore, most DRL approaches are centralized, where a single global DRL agent observes and controls the entire network. Up-to-date global knowledge of fast-changing information such as demand or resource utilization as well as fine-grained central control of individual, rapidly arriving flows are unrealistic in practical large-scale networks.

To address these issues, I propose a self-learning approach that solves the full network and service coordination problem in a distributed manner. The approach combines ideas from the DeepCoord DRL approach of Chapter 8 and the fully distributed heuristics of Chapter 6. Specifically, it deploys separate DRL agents at each node in the network, deciding coordination individually in parallel. These agents are trained offline in a centralized fashion, leveraging experience from all agents, and then coordinate network and services independently online in a fast, fully distributed fashion. Each agent only observes itself and its direct neighbors and only has local control of how incoming flows are processed and forwarded. Hence, the approach requires neither global knowledge nor centralized control. In contrast to typical centralized approaches, the size of observation and action spaces is invariant in the network size and only depends on the network degree, i.e., the maximum number of neighbors per node. Since the network degree is typically much smaller than the total number of nodes in the network [142], the approach scales better to realistic, large-scale networks and allows fast fine-grained control of individual flows. Furthermore, the approach generalizes to new, unseen scenarios and is also more robust to failures than typical centralized approaches as there is no single point of failure. Hence, the presented approach combines the flexibility and self-adaption of DRL (Chapter 8) with scalability and fast per-flow control of distributed coordination (Chapter 6). The evaluation shows that it consistently and significantly outperforms both the fully distributed heuristic from Chapter 6 and the centralized DRL approach from Chapter 8. Overall, the contributions of this chapter are:

- In Section 9.4, I formalize a Partially Observable Markov Decision Process (POMDP) and propose a novel self-learning approach using distributed DRL agents for scalable network and service coordination in practice.
- Section 9.5 evaluates the approach on real-world network topologies and traffic traces, showing its adaptability, generalization capabilities, and scalability.
- For reproducibility, the code is publicly available online [S11].

9.2. Related Work

Existing work mostly proposes conventional optimization approaches without DRL for solving the network and service coordination problem [166, 114, 110]. Many authors [182, 179, 28, 144] consider offline coordination with full a priori knowledge of user demand. Related work that does consider online coordination [85, 91, S2] focuses on scenarios with few and long flows

(Case I in Section 3.1.3), assuming traffic to arrive in fixed intervals and to be known a priori for each interval. Instead, this chapter focuses on scenarios with many, short flows (Case II), which requires scheduling these rapidly incoming flows quickly at runtime. Blöcher et al. [29] do schedule flows dynamically at runtime but assume a given fixed placement and do not consider routing. In contrast, this chapter proposes joint scaling, placement, scheduling, and routing at runtime in a distributed fashion without any a priori knowledge.

In Chapter 6, I propose two algorithms for fully distributed network and service coordination at runtime, including scaling, placement, scheduling, and routing. Similar to the fully distributed DRL approach proposed in this chapter, these algorithms only have local observations and control, making individual coordination decisions at each node in parallel. In contrast to the approaches in Chapter 6, the approach here explicitly considers potential deadlines that bound flows' maximum acceptable delay to achieve good Quality of Service (QoS). More importantly, the algorithms of Chapter 6, as well as all other aforementioned approaches, are hand-crafted and include built-in assumptions, which may not hold in practice. In contrast, the proposed model-free DRL approach in this chapter does not follow a hand-crafted algorithm and has very few built-in assumptions (e.g., about incoming traffic). Instead, it learns from experience how to effectively deal with different traffic patterns. I evaluate and compare the proposed DRL approach against a fully distributed conventional heuristic from Chapter 6 in Section 9.5.

Recently, DRL for self-learning coordination has been proposed [279, 199, 255, 209, 261, 265, 98, S21]. Zhang et al. [279] focus only on placement and use tabular Q-learning, which does not support large and continuous observations or generalization between observations, limiting practical applicability. Pei et al. [199] rely on a simulator to test actions and select the best one in each time step, which would be too slow for online coordination at runtime with rapidly arriving flows—especially when done in a centralized fashion. Wang et al. [255] schedule flows equally to all placed instances, which could lead to high end-to-end delays and bad service quality. More similar to the approach proposed in this chapter, Quang et al. [209] and Xiao et al. [261] dynamically control individual flows and adjust component placement accordingly. However, these authors do not consider scaling or routing. Furthermore, they propose centralized approaches, assuming global up-to-date knowledge and control of the entire network. In practice, global knowledge can be achieved through monitoring but only with some monitoring delay (e.g., 1 min in Prometheus by default [204]), making it unsuitable for fast per-flow decisions at runtime. Such expensive centralized decisions per flow do not scale to large networks and would be too slow for efficient online control of many rapidly arriving, time-sensitive flows as considered here (Case II).

DeepCoord proposed in Chapter 8 as well as related approaches [265, 98] consider centralized DRL but avoid scalability issues by installing scheduling rules at all nodes, which are then applied to incoming flows in a distributed fashion at runtime. These rules are updated periodically by the centralized DRL agent. Furthermore, DeepCoord only relies on realistically available, infrequent, partial, and delayed observations. Compared to the distributed DRL approach proposed in this chapter, these approaches still have a number of drawbacks: Xu et al. [265] only consider traffic engineering but not scaling and placement of service components. DeepCoord (Chapter 8) and the approach by Gu et al. [98] do not support dynamic routing but rely on routing along predefined paths. Xu et al. and Gu et al. further require support from a hand-written heuristic. Moreover, all three approaches optimize coarse-grained rules that are applied to all incoming flows and do not have fine-grained control over individual flows. As shown in the evaluation (Section 9.5), such fine-grained control is important for precise load balancing and proper dynamic routing. The distributed DRL approach proposed in this chapter jointly optimizes scaling, placement, scheduling, and routing, thus explicitly controlling all aspects of network and service coordination described in Section 3.2. It does not require support from a heuristic and it controls individual flows efficiently by distributing decisions over different DRL agents for each node.

To the best of my knowledge, this is the first distributed self-learning approach for online network and service coordination. Compared to existing work, the approach is more powerful as it considers dynamic scaling and placement as well as per-flow scheduling and routing and it is more efficient due to its fully distributed architecture.

9.3. Problem Statement

I consider the problem of network and service coordination as described in Chapter 3. Similar to DeepCoord from Chapter 8, the distributed DRL approach of this chapter does not require explicit knowledge of the full network state and all network parameters (Section 3.1). Instead, it self-adapts to any given scenario through partial observations and feedback from its actions. In contrast to the centralized DeepCoord approach (Chapter 8), the fully distributed agents here observe up-to-date local information rather than delayed global information. For example, they observe the currently utilized resources $r_v(t)$ and $r_l(t)$ of neighboring nodes and links and information about the currently requested component $c_f \in C_{s_f} \cup \{\varnothing\}$ of an incoming flow f (detailed in Section 9.4.2). The size of these observations is bounded not by the network size (i.e., $|V|$ and $|L|$) but by the network degree Δ_G, i.e., the maximum number of neighbors in the network.

Similar to Chapter 8, this chapter mainly focuses on scenarios with many, short flows (Case II in Section 3.1.3) but is not strictly limited to such scenarios. These flows may have service-specific QoS requirements $\Theta_s = \{d_s\}$ in the form of hard deadlines d_s that are defined relative to flow arrival time t_f^{in}. The remaining time of a flow f from time t until its deadline is denoted by $d_f(t) = d_{s_f} - (t - t_f^{\text{in}})$. Once $d_f(t) = 0$, the deadline is reached, the flow expires and is dropped automatically, freeing any currently blocked resources.

The goal here is to maximize the percentage of successful flows over all T time steps:

$$\max o_f = \frac{|F_{\text{succ}}|}{|F_{\text{succ}}| + |F_{\text{drop}}|} \in [0,1] \tag{9.1}$$

A flow f is successful after routing from ingress v_f^{in} to egress v_f^{eg} while traversing instances of all components of the requested service s_f and completing within deadline d_{s_f}. Hence, maximizing objective o_f requires dynamic service scaling and placement as well as flow scheduling and routing (Section 3.2), taking limited resources and QoS requirements into account.

9.4. Distributed DRL Approach

I propose a DRL approach based on centralized training and fully distributed inference to coordinate network and services online, maximizing objective o_f. The approach consists of separate DRL agents deployed at each node in the network, where each agent only has local observations and control. In Section 9.4.1, I explain how these distributed agents jointly scale and place services as well as schedule and route incoming flows at runtime. Section 9.4.2 formalizes the POMDP, specifying the observation and action spaces as well the reward function for the DRL approach. Finally, the overall framework for training and inference is discussed in Section 9.4.3.

9.4.1. Joint Scaling, Placement, Scheduling, and Routing

At each node v, a separate DRL agent controls incoming flows independently from agents located at other nodes. Whenever a flow f arrives at a node v, the DRL agent at v needs to decide whether to process the flow locally at an instance of requested component c_f hosted at v (i.e., $y_{f,c_f}(t) = v$) or to forward f to a neighbor v' (i.e., $z_{f,v}(t) = v'$). By setting $y_{f,c_f}(t)$ and $z_{f,v}(t)$, the agents directly control flow scheduling and routing (Section 3.2).

Scaling and placement is jointly derived from $y_{f,c_f}(t)$, by setting $x_{c_f,v}(t) = 1$ if $y_{f,c_f}(t) = v$. Hence, if the agent decides to process f locally at node v, this implies that an instance of c_f is either already available at v or a new instance is automatically started. Unused instances of a component c_f are removed after a manually configured timeout δ_{c_f}. If idle instances of component c_f consume a lot of resources (i.e., $r_{c_f}(0)$ is large), setting a short timeout is useful to remove unused instances quickly and avoid wasting resources. Conversely, setting a long timeout, e.g., for components with negligible idle resource consumption, reduces overhead of frequently terminating and restarting instances due to fluctuating demand. In future work, the DRL approach could be extended to also dynamically control timeout δ_c.

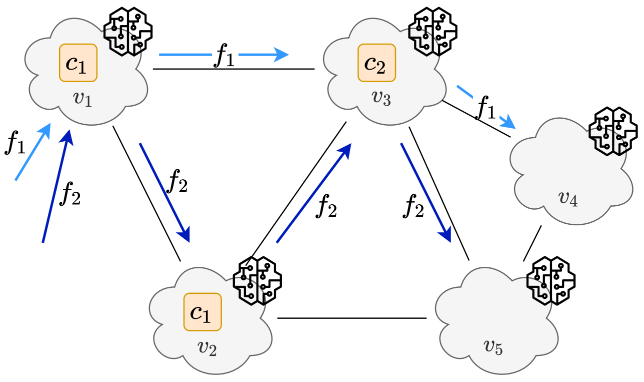


Figure 9.1.: Example with separate DRL agents at each node coordinating two incoming flows f_1, f_2. (Figure from [S34]; © 2021 IEEE.)

Figure 9.1 illustrates the overall process in an example. Here, a flow f_1 and, shortly after, another flow f_2 arrive at ingress node v_1 and both request service s with $C_s = \langle c_1, c_2 \rangle$. Flow f_1 has egress $v^{\text{eg}}_{f_1} = v_4$ and f_2 has egress $v^{\text{eg}}_{f_2} = v_5$. When f_1 arrives at v_1 at time t_1, the node still has enough free resources to host an instance of $c_f = c_1$, such that v_1's DRL agent decides to process f_1 locally, setting $x_{c_1,v_1}(t_1) = 1$, $y_{f_1,c_1}(t_1) = v_1$, and $z_{f,v_1}(t_1) = \emptyset$. When f_2 arrives shortly after at time t_2, v_1's resources are already fully utilized such that v_1's DRL agent decides to forward f_2 to its neighbor v_2, setting $z_{f_2,v_1}(t_2) = v_2$. At v_2, the corresponding DRL agent decides to process f_2 locally, i.e., $x_{c_1,v_2}(t_3) = 1$ and $y_{f_2,c_1}(t_3) = v_2$.

In the following, f_1 finishes processing c_1 at v_1, then requests the next component c_2, and is forwarded to neighbor v_3. At v_3, the corresponding DRL agent decides to instantiate c_2 and process f_1 locally. Similarly, f_2 is sent to process c_2 at v_3. After processing, the DRL agent at v_3 sends each flow to its egress node, where the flows depart successfully.

This example illustrates how the distributed DRL agents coordinate incoming flows individually and in parallel to the other nodes' agents. Depending on their own resource utilization as well as the utilization and location of their neighbors, they decide for each flow whether to process it locally or to forward it to a suitable neighbor, setting decision variables $x_{c,v}(t)$, $y_{f,c}(t)$, and $z_{f,v}(t)$ jointly at runtime.

9.4.2. Partially Observable Markov Decision Process (POMDP)

The complete network state depends on a multitude of parameters (Section 3.1) and cannot be fully observed by the DRL approach. Instead, each agent only has local and partial observations that are realistically available in practice. To solve the problem using DRL, I formalize the corresponding POMDP as tuple $(\mathcal{O}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, consisting of observations $\mathcal{O}$, actions $\mathcal{A}$, typically unknown environment dynamics $\mathcal{P}$, and reward function $\mathcal{R}$. In contrast to DeepCoord (Chapter 8), which is called periodically and independently from flow arrival, the fully distributed DRL agents in the approach proposed here obtain observations, take actions, and receive a reward whenever a flow arrives at their corresponding node. I define $\mathcal{O}, \mathcal{A}$, and $\mathcal{R}$ accordingly as follows.

Observations $\mathcal{O}$

An agent's observations are restricted to local information about the incoming flow f, current node v itself, and its neighbors. Specifically, $\mathcal{O} = \langle F_f, R_v^L, R_v^V, D_{v,f}, X_v \rangle$ consists of flow attributes F_f, link utilization R_v^L, node utilization R_v^V, delays to egress $D_{v,f}$, and available instances X_v. I normalize all elements within the vectorized observations to be in range $[0,1]$ or $[-1,1]$ as detailed below. Ensuring that all observations are in a roughly similar range is important for effective training and generalization of deep neural networks [118]. Otherwise, the DRL agent may become "blind" to the weak signals of observations with a small range, which are drowned out by stronger signals of other observations with much larger ranges. Next, I define observations $\mathcal{O} = \langle F_f, R_v^L, R_v^V, D_{v,f}, X_v \rangle$ in more detail.

Flow Attributes Vector $F_f = \langle \hat{p}_f, \hat{d}_f \rangle$ consists of two relevant flow attributes. The progress inside the service chain of a flow f is indicated by $\hat{p}_f \in [0,1]$ (cf. service index in Service Function Chaining (SFC) [210]). It starts at $\hat{p}_f = 0$ when the flow arrives and progresses towards $\hat{p}_f = 1$ with every traversed component. With respect to deadlines, the agent observes $\hat{d}_f = \frac{d_f(t)}{d_{s_f}} \in [0,1]$, which is the remaining time $d_f(t)$ to the flow's deadline normalized by deadline d_{s_f} itself (defined relative to flow arrival t_f^{in}). Hence, it starts at $\hat{d}_f = 1$ and gradually decreases towards 0 over time.

Link Utilization Vector $R_v^L = \langle \frac{\text{cap}_l - r_l(t) - \lambda_f}{\max_{l' \in L_v} \text{cap}_{l'}} | l \in L_v \rangle \in [-1,1]^{\Delta_G}$ contains the free resources on all outgoing links L_v of node v, normalized by the maximum capacity of all outgoing links L_v. Subtracting the flow's data rate λ_f shifts the value to be ≥ 0 if and only if a link can forward the flow.

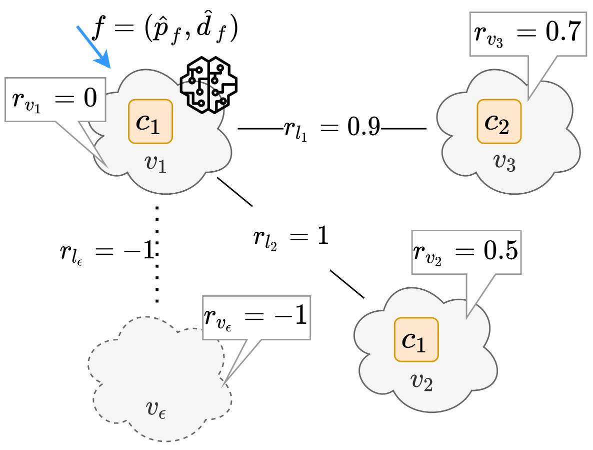


Figure 9.2.: Example observations of the DRL agent at node v_1 in Figure 9.1. Node v_ϵ is a dummy node to ensure a consistent size of observations across all agents. Time t is omitted for simplicity. (Figure adapted from [S34]; © 2021 IEEE.)

To allow combining experiences from all agents (details in Section 9.4.3), the size of the observation and action spaces need to be identical for all agents. Hence, I define $|R_v^L| = \Delta_G$ according to network degree Δ_G. If a node has less than Δ_G neighbors, I add dummy neighbors v_ϵ to V_v and set their link utilization to -1, indicating that they do not exist. Figure 9.2 shows the observations of the DRL agent at v_1 in the example network of Figure 9.1. Here, a dummy node v_ϵ is added to v_1's two neighbors to match the network degree $\Delta_G = 3$.

Node Utilization Similar to R_v^L, vector $R_v^V = \langle \frac{\text{cap}_{v'} - r_{v'}(t | x_{c_f,v'}(t)=1 \wedge y_{f,c_f}(t)=v')}{\max_{v'' \in V} \text{cap}_{v''}} | v' \in V_v \cup \{v\} \rangle \in [-1,1]^{\Delta_G+1}$ contains the free compute resources at v and its neighbors if they were to process the incoming flow f. Hence, elements in R_v^V are ≥ 0 for nodes with enough free resources to process f and < 0 for nodes with insufficient resources. This helps the agent decide whether to process flow f locally and to which neighbor to send f next, such that f can be processed successfully without overloading nodes. As in Section 6.4.1, $r_{v'}(t | x_{c_f,v'}(t) = 1 \wedge y_{f,c_f}(t) = v')$ denotes the total resources required at node v' at time t if f were to be processed there. This can be calculated locally, even before making the corresponding scheduling decision $y_{f,c}(t)$, based on the monitored data rate processed at v' and knowledge of λ_f and $r_c(\lambda)$. Chapter 7 presents how to automatically derive function $r_c(\lambda)$ based on real-world data. In case flow f is already fully processed ($c_f = \varnothing$), it does not require further resources for processing, i.e., $r_{v'}(t | x_{c_f,v'}(t) = 1 \wedge y_{f,c_f}(t) = v') = r_{v'}(t)$.

As before, I ensure $|R_v^V| = \Delta_G + 1$ (including v itself) by adding observations of $r_{v_\epsilon}(t) = -1$ for dummy nodes v_ϵ if necessary. Similar to R_v^L, the observation for all non-dummy nodes is normalized by the maximum node capacity such that it is in $[0,1]$ if there are enough resources for processing the flow and in $[-1, 0)$ otherwise. Here, there is a small difference in how values are normalized compared to R_v^L. Values in R_v^L are normalized by the maximum capacity of outgoing links L_v (not all links L) because flow f must necessarily be routed via one of these outgoing links L_v. In contrast, f can be processed at v and its neighbors V_v but also at any other node in V. Hence, I normalize values in R_v^V by the maximum capacity over all nodes V (not just neighbors V_v). This normalization helps the DRL agent to identify nodes with high available absolute resources, not just relative to the neighborhood.

Delays to Egress Vector $D_{v,f} = \langle \max \left\{ -1, \frac{d_f(t) - d_{v,v',v_f^{\text{eg}}}}{d_f(t)} \right\} | v' \in V_v \rangle \in [-1, 1]^{\Delta_G}$ defines the shortest path delays (denoted by $d_{v,v',v_f^{\text{eg}}}$) from current node v to flow f's egress via each neighbor v' in relationship to the remaining time $d_f(t)$ to f's deadline. This information helps the agent forward f to neighbors that are in the direction of its egress node. If the observation is below 0 for a neighbor v', there is no chance that forwarding via v' will be successful. Assuming a fixed network topology and link delays, the shortest paths and their path delays $d_{v,v',v_f^{\text{eg}}}$ can be precomputed and accessed in constant time during runtime. Again, $D_{v,f}$ is padded with -1 to ensure length $|D_{v,f}| = \Delta_G$.

Available Instances Binary vector $X_v = \langle x_{c_f,v'}(t) | v' \in V_v \cup \{v\} \rangle \in \{-1, 0, 1\}^{\Delta_G + 1}$ indicates whether an instance of requested component c_f is currently available at v and its neighbors based on variable $x_{c_f,v'}(t)$ (Section 3.2.1). For fully processed flows (i.e., $c_f = \varnothing$), the corresponding observation $x_{\varnothing,v'}(t)$ is always zero. Again, I pad X_v with -1 to ensure length $|X_v| = \Delta_G + 1$.

Actions $\mathcal{A}$

The DRL agents take actions whenever a flow f arrives at their node. The action space $\{0, 1, ..., \Delta_G\}$ is the same for all agents depending on network degree Δ_G. An action $a \in \{0, 1, ..., \Delta_G\}$ by a DRL agent at node v specifies how to set $x_{c_f,v}(t)$, $y_{f,c_f}(t)$, and $z_{f,v}(t)$ as described in Section 9.4.1. If $a = 0$, the flow is processed locally, i.e., $y_{f,c_f}(t) = v$. An action $a > 0$ sends the flow to v's a-th neighbor $v_a \in V_v$, i.e., $z_{f,v}(t) = v_a$. An action is only valid for $a \leq |V_v|$. If a node has fewer neighbors than Δ_G, an action $|V_v| < a \leq \Delta_G$ points to a non-existing neighbor and is invalid. In this case, flow f is dropped and the agent receives a high penalty. Based on the observed -1 values for non-existing dummy neighbors, agents should be aware of which neighbors really exist and only forward flows there.

If a DRL agent selects $a = 0$ even though flow f is already fully processed ($c_f = \varnothing$), f stays at the node for one time step and the agent is queried again. Doing so reduces the remaining time $d_f(t)$ to f's deadline and incurs a small penalty. Based on this penalty and on observation $\hat{p}_f$, agents should learn to forward processed flows directly to their egress without keeping them unnecessarily.

Reward $\mathcal{R}$

Ultimately, the DRL agents should learn a coordination scheme that processes as many flows successfully as possible. Hence, I give a large positive reward of $+10$ when a flow completes. Conversely, I penalize the agent with a reward of -10 when dropping a flow.

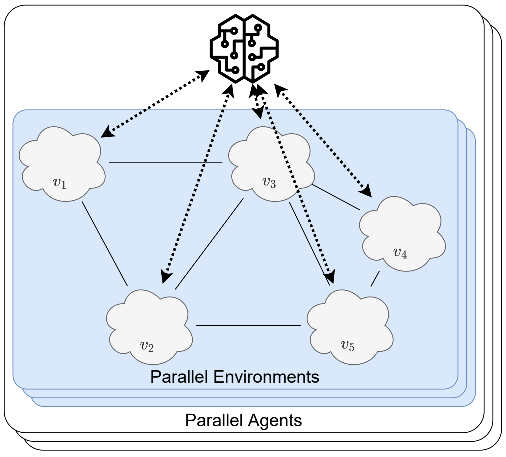
Successful completion of a flow requires proper coordination, taking available neighbors, compute and link resources, link delays, and flow deadlines into account. Hence, when starting training with a random policy, it is very unlikely that a series of random actions leads to a flow completing successfully. Consequently, rewards of $+10$ will initially be very sparse, preventing effective training.

One way to address the challenge of sparse rewards is through reward shaping [249]. To this end, I add additional weaker reward signals that indicate whether an action seemed useful or not—even before a flow is completed or dropped. Specifically, the DRL agents receive a small positive reward of $+\frac{1}{|C_{s_f}|}$ whenever a flow successfully traverses an instance, where $|C_{s_f}|$ is the length of the requested service chain. This encourages the DRL agents to host component instances and process flows locally when possible. Furthermore, I give a small penalty of $-\frac{d_l}{D_G}$ whenever a DRL agent sends a flow along a link l. This penalty corresponds to the link's propagation delay normalized by the network's diameter D_G (in terms of path delay). It encourages the DRL agents to forward flows along fewer links and links with shorter delays when possible. I give a similar penalty of $-\frac{1}{D_G}$ when a DRL agent keeps a flow ($y_{f,c_f}(t) = v$) that is already fully processed.

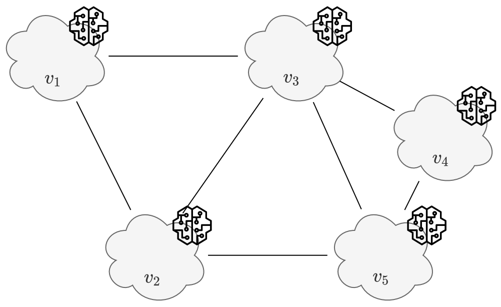
These additional rewards help nudge the DRL agents towards a useful policy and can improve training. Yet, it is important that these auxiliary rewards are significantly smaller than the rewards/penalties for successful and dropped flows. Otherwise, they limit the agents' ability to find "creative" but successful policies. Too strong auxiliary rewards can even encourage unwanted behavior, e.g., if processing two flows half-way is more rewarding than fully completing one flow. Hence, the auxiliary rewards are intentionally small compared to the $+/-10$ rewards for completed/dropped flows.

9.4.3. DRL Coordination Framework

As defined in Section 9.4.2, this chapter approaches network and service coordination through local observations and actions that control incoming flows for each node. I propose a DRL framework with offline training, combining experience from all nodes in a logically centralized neural network as illustrated in Figure 9.3a. After training converges, this single, central neural network can make decisions for any node in the network. It distinguishes different nodes based on different observations and takes suitable actions accordingly. Now, separate DRL agents can be deployed at each network node, copying the neural network to each of these agents (Figure 9.3b). This allows each agent to locally control incoming flows highly efficiently based only on local observations. While each DRL agent has a copy of the same neural network and thus follows the same policy, this policy is trained to distinguish flows at different nodes and handle them correspondingly.



(a) Centralized training



(b) Distributed inference

Figure 9.3.: a) Centralized training procedure with multiple copies of the network environment and the DRL agent. b) Fully distributed inference with trained DRL agents at each node. (Figure from [S34]; © 2021 IEEE.)

Design Choices

Two natural alternatives to this approach are either centralized observations and control for all network nodes at once or distributed training and inference using different neural networks for each node. A centralized approach observing and controlling all nodes at once is more in line with current related work (Section 9.2 and Chapter 8). Yet this would require a significantly larger observation and action space, e.g., a concatenation of the observations and actions for each node, which typically requires more training until convergence. Furthermore, centralized inference decisions per flow would not scale to large networks with rapidly incoming flows, where thousands of decisions may be necessary per millisecond for the network as a whole. Besides, it would require up-to-date global knowledge, which is not realistically available, but at best partial and delayed through monitoring (as assumed in Chapter 8). Distributed training and inference using separate neural networks is a promising alternative approach, yet makes training challenging. E.g., when training distributed neural networks, agents at nodes that are seldom traversed by flows would barely be trained at all, possibly leading to bad policies for these nodes. Hence, the proposed centralized training with distributed inference tries to combine the benefits of both approaches, e.g., by leveraging experience from all agents, providing more data for effective training. To support continuous online training during inference, DRL agents could update their neural network locally and then synchronize the gradient updates with all other nodes (cf. federated learning [163]). Lest online inference is blocked, training and sharing of updates should happen asynchronously.

However, the approach also comes with the challenge of properly designing the POMDP such that the trained neural network can effectively generalize between similar situations but still distinguish semantically different situations. For example, to keep the action space small, action *i* means selecting neighbor *i*, not necessarily node *i*. Hence, the same action *i* can lead to very different results when executed by DRL agents at different nodes. To help the DRL agents distinguish between different neighbors and selecting a suitable one, the observations include all relevant information about neighbors' resources, available instances, and distance to the egress node (in terms of shortest path delay). This should help the DRL agents learn to select neighbors with sufficient resources and towards the egress node—independent from the exact node IDs and thus generalizing across DRL agents at different nodes.

Algorithm

For training the DRL agents, I leverage the Actor-Critic using Kronecker-factored Trust Region (ACKTR) algorithm [258]. ACKTR is an extension to the well-known Asynchronous Advantage Actor-Critic (A3C) [180], which leverages multiple parallel environment copies during training for more diverse training data. Similar to Trust Region Policy Optimization (TRPO) [227] and Proximal Policy Optimization (PPO) [228], ACKTR ensures that the learned policy is updated gradually during training, avoiding abrupt and potentially destructive changes in the learned behavior. In contrast to Deep Deterministic Policy Gradient (DDPG) [155] used for continuous actions in Chapter 8, ACKTR is better suited for discrete actions required here.

Algorithm 7 DRL With Centralized Training and Fully Distributed Inference

```
 1: Initialize $\pi_\theta, V_\phi, b$                                   ▷ Centralized Training
 2: $l \leftarrow$ number of parallel training environments
 3: for $l$ environments in parallel do
 4:     while $t \leq T$ do
 5:         if flow $f$ arrives at node $v$ then
 6:             $o_t, r_t \leftarrow$ adapter.process($f, v, V_v, L_v, G$)
 7:             $b \xleftarrow{\text{add}} (o_{t-1}, a_{t-1}, r_t, o_t)$
 8:             $a_t \leftarrow \pi_\theta(o_t)$
 9:             $x_{c_f,v}(t), y_{f,c_f}(t), z_{f,v}(t) \leftarrow$ adapter.process($a_t$)
10:         if $b$ is full then
11:             Train $V_\phi$ using temporal difference updates [180]
12:             Train $\pi_\theta$ maximizing $\mathbb{E}[\sum_i \gamma^i r(o_{t+i}, a_{t+i})]$
13: Select best agent $(\pi_\theta, V_\phi)$                              ▷ Distributed Inference
14: Deploy a copy $\pi_\theta^v$ of $\pi_\theta$ at each node $v \in V$
15: while $t \leq T$ do
16:     if flow $f$ arrives at node $v$ then
17:         $o_t, r_t \leftarrow$ adapter.process($f, v, V_v, L_v, G$)
18:         $a_t \leftarrow \pi_\theta^v(o_t)$
19:         $x_{c_f,v}(t), y_{f,c_f}(t), z_{f,v}(t) \leftarrow$ adapter.process($a_t$)
```

Algorithm 7 shows the high-level algorithm for centralized offline training (lines 1–12) and then for distributed online inference (lines 13–19). ACKTR trains two neural networks for the actor π_θ and the critic V_ϕ using mini-batches b. The two neural networks are initialized randomly and trained over l parallel copies of the network environment (lines 1–3). Clearly, $l = 1$ if the network environment cannot be duplicated. Whenever a flow arrives at a node (lines 5–9), the DRL agent obtains current observation o_t and previous reward r_t from the network through an adapter. It adds the experience to mini-batch b and selects the next action by passing o_t to its actor π_θ. Based on selected action a_t, the corresponding coordination decisions are taken as described in Sections 9.4.1 and 9.4.2.

Once mini-batch b is full, i.e., contains a predefined number of experiences, critic and actor are trained (lines 10–12). Critic V_ϕ estimates the long-term value $V_\phi(o)$ of observing o and following the current policy. It is trained using bootstrapping and temporal difference as detailed in [258], which is a standard technique for reinforcement learning [242]. The value estimate $V_\phi(o)$ is necessary to calculate the advantage, i.e., the relative value of each action a after observing o, which is, in turn, needed to train the actor π_θ. The actor is trained to maximize the long-term return, i.e., the discounted (by γ) sum of future rewards, using the natural gradient method [258]. This training procedure is repeated for a configurable number of training episodes until the agent converges.

As the random training seed can significantly impact convergence [108], I train multiple agents (in parallel) using k different random seeds. After training, I automatically select the best agent with the highest reward for online inference (line 13). The neural network of this agent is copied to each network node to facilitate fast, distributed inference by a local DRL agent at each node (lines 14–19). While offline training can be time-intensive, depending on the number of training episodes, online inference is very fast [107]. With a fixed number of hidden units, time and space complexity for inference is in $O(\Delta_G)$, i.e., linear in the network degree.

Implementation and Target Deployment

Figure 9.4 illustrates the implemented prototype of the DRL approach, which is built on Tensorflow [4] and the `stable-baselines` framework [112] using Python. The source code is publicly available on GitHub [S11]. As light-weight network simulator, I used `coord-sim` [S26]. Similar to DeepCoord (Section 8.4.3), each DRL agent here also interacts with the network through adapters, using the OpenAI Gym interface [32]. These adapters interface the network environment to retrieve relevant observations (e.g., from monitoring data), calculate the reward during training, and apply the selected actions. In a possible Network Function Virtualization (NFV) target deployment [110], the DRL agents could be deployed with and interface a distributed Management and Orchestration (MANO) [215] for Virtual Network Function (VNF) orchestration and traffic steering.

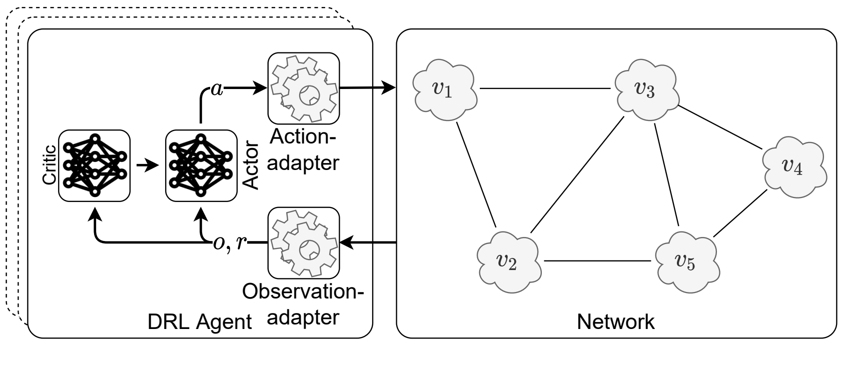


Figure 9.4.: Implementation consisting of the DRL agent interacting with the network through adapters. (Figure adapted from [S34]; © 2021 IEEE.)

9.5. Evaluation

I evaluate the proposed fully distributed DRL approach and compare its performance against state-of-the-art approaches in a variety of different scenarios. In Section 9.5.2, I evaluate how well the DRL approach adapts to varying ingress nodes and traffic patterns, comparing the percentage of successful flows (as defined in Section 9.3) over $T = 20000$ time steps. Similarly, I evaluate its adaptability to varying flow deadlines in Section 9.5.3, just by retraining the DRL agent for each scenario but without changing any hyperparameters or making manual adjustments. In Section 9.5.4, I investigate how well the DRL approach generalizes to previously unseen scenarios without any retraining. Finally, Section 9.5.5 evaluates the scalability of the approach on large real-world network topologies.

9.5.1. Evaluation Setup

Base Scenario

I consider different variations of a base scenario using real-world network topology Abilene [142], which connects 11 cities in the United States. In Section 9.5.5, I also consider larger real-world topologies to evaluate scalability. Compute resources are assigned to nodes uniformly at random between 0 and 2, link capacities between 1 and 5, and link delay is derived from the distance between connected nodes. I consider a video streaming service s with $C_s = \langle c_{\text{FW}}, c_{\text{IDS}}, c_{\text{video}} \rangle$ and deadline $d_s = 100\,\text{ms}$. While I successfully tested the approach with multiple services, this evaluation focuses on a single service for simplicity. All components $c \in C_s$ have a processing delay of $d_c = 5\,\text{ms}$ and require resources linear to their load. Flows requesting s have unit data rate ($\lambda_f = 1$) and length ($\delta_f = 1$), but I consider scenarios with increasingly complex flow arrival patterns (Section 9.5.2) and varying deadlines (Section 9.5.3). I consider between 1 and 5 ingress nodes (v_1–v_5) and a single egress v_8.

DRL Hyperparameters

I use the following hyperparameter settings for the DRL approach:

- Fully connected neural networks for both actor and critic. Each neural network is configured with two hidden layers, each with 64 hidden units and tanh activation [126], trained with the RMSprop optimizer [217].
- Discount factor $\gamma = 0.99$.
- Initial learning rate $\alpha = 0.25$.
- Batch size $|b| = 80$.
- $k = 10$ training seeds and $l = 4$ parallel training environments (Section 9.4.3).
- ACKTR-specific parameters: Entropy loss 0.01, loss on V_ϕ 0.25, and Fisher coefficient 1.0, max. gradient 0.5, Kullback-Leibler clipping 0.001.
- Timeout and removal of unused instances after $\delta_c = 100\,\text{ms}$ for all components $c \in C$ (Section 9.4.1).

I selected these hyperparameter settings based on the ACKTR default settings [112] and manual tuning to achieve good performance across scenarios. Tuning hyperparameters automatically for each scenario is slow and resource-intensive but could potentially further improve performance.

Compared Algorithms

I compare the proposed fully distributed DRL approach against two coordination approaches from previous chapters, where the code is publicly available, and against a simple greedy baseline:

- *DeepCoord*: The centralized DRL approach from Chapter 8, which uses and periodically updates scheduling rules at all nodes that are applied to incoming flows at runtime. It handles partial and delayed observations of the global network state, which are available via periodic monitoring.
- *Greedy Coordination with Adaptive Shortest Paths (GCASP)*: The fully distributed heuristic algorithm from Chapter 6. In both GCASP and the proposed approach in this chapter, each node observes and controls incoming flows locally.
- *Shortest Path Coordination Approach (SP)*: A simple greedy baseline, which tries to process all flows along the shortest path from ingress to egress.

Execution & Figures

All experiments were executed on machines with an Intel Xeon W-2145 CPU and 128 GB RAM. Figures show the mean and 95 % confidence interval over 30 random seeds.

9.5.2. Varying Traffic Patterns

I evaluate the proposed distributed DRL approach on the base scenario (Section 9.5.1) with varying flow arrival patterns and an increasing number of ingress nodes (v_1–v_5) and thus increasing load. The simplest pattern I consider is *fixed flow arrival* with flows arriving every 10 time steps at each ingress node. Figure 9.5a shows the corresponding percentage of successful flows, which naturally decreases with increasing load and limited node and link capacities. While all approaches process almost all flows successfully with one ingress node, only the self-learning DRL approaches achieve around 100 % successful flows for up to three ingress nodes. For four and five ingress nodes, the proposed distributed DRL approach outperforms all other algorithms and successfully processes up to 35 % more flows than the centralized DeepCoord approach. While DeepCoord relies on shortest path routing, the distributed DRL approach in this chapter explicitly optimizes routing jointly with scaling, placement, and scheduling. In doing so, it takes link capacities into account and balances load across different paths to minimize dropped flows. The simple SP baseline relies on sufficient resources along the shortest path and thus easily drops flows. Ingress nodes v_1–v_3 are co-located in the given network such that their shortest paths to the egress overlap and compete for shared resources. Ingress v_4 and v_5 are farther away such that their shortest paths do not overlap and, instead, allow SP to utilize more resources for more successful flows. Similar to SP, the fully distributed heuristic, GCASP, favors processing flows along the shortest paths but dynamically reroutes flows when necessary, avoiding bottlenecks and searching for compute resources.

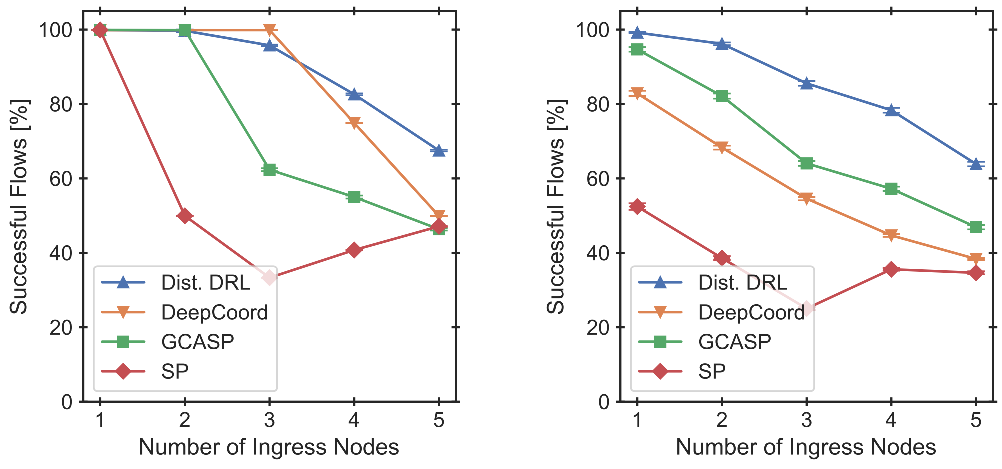
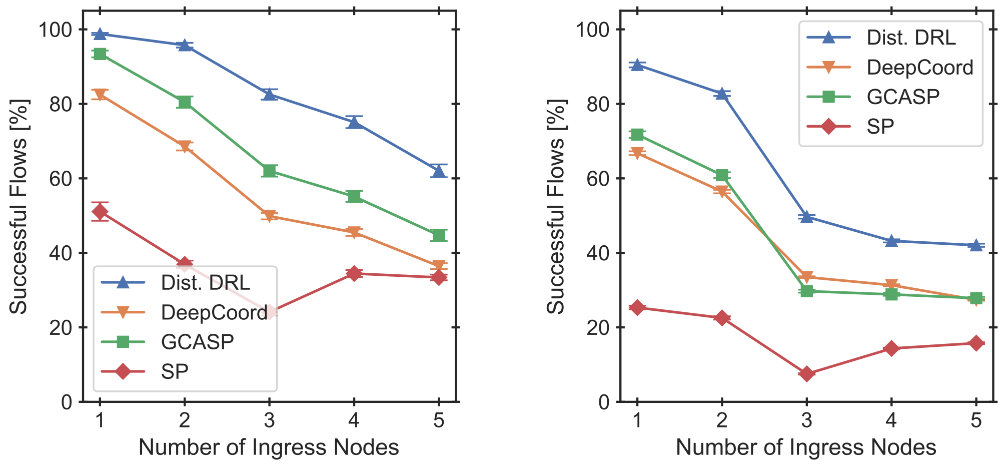


Figure 9.5.: Percentage of successful flows for an increasing number of ingress nodes, i.e., increasing load, and increasingly realistic traffic patterns (here, fixed and Poisson flow arrival; extended in Figure 9.6). Compared to other approaches, the distributed DRL approach processes most flows successfully. (Figure adapted from [S34]; © 2021 IEEE.)

Next, I consider stochastic flow arrival following a *Poisson process* with exponentially distributed inter-arrival times (mean 10 time steps). Figure 9.5b shows the corresponding results. Here, DeepCoord performs worse because its centralized observations are always slightly outdated—as they would be for any centralized approach in practice! Hence, it cannot effectively react to small bursts where multiple flows arrive directly after another. Instead, the same scheduling rules are applied to all incoming flows, which easily leads to dropped flows when considering stochastic flow arrival. In contrast, the fully distributed architecture based on local observations and control of both the proposed distributed DRL approach and GCASP lets them control individual flows quickly at runtime. This allows them to react to short bursts, distributing individual flows among available resources. Still, the distributed DRL clearly and consistently outperforms all other algorithms and is 37 % better than GCASP on average.

Figure 9.6a shows very similar results for yet more realistic flow arrival following a *Markov-Modulated Poisson Process (MMPP)* [80]. The corresponding two-state Markov process randomly switches between flow arrival with mean inter-arrival time 12 and 8 (50 % higher rate) every 100 time steps with 5 % probability. The distributed DRL approach successfully adapts to this traffic pattern and outperforms all other approaches (GCASP by 39 % on average).

Finally, Figure 9.6b shows the results for flows following real-world traffic traces that are publicly available for the Abilene network [194]. Here, DeepCoord and GCASP lead to comparable results. Again, the distributed DRL is considerably better and even increases its lead over both approaches with 60 % higher success rates on average.



(a) MMPP arrival (b) Real-world traffic trace

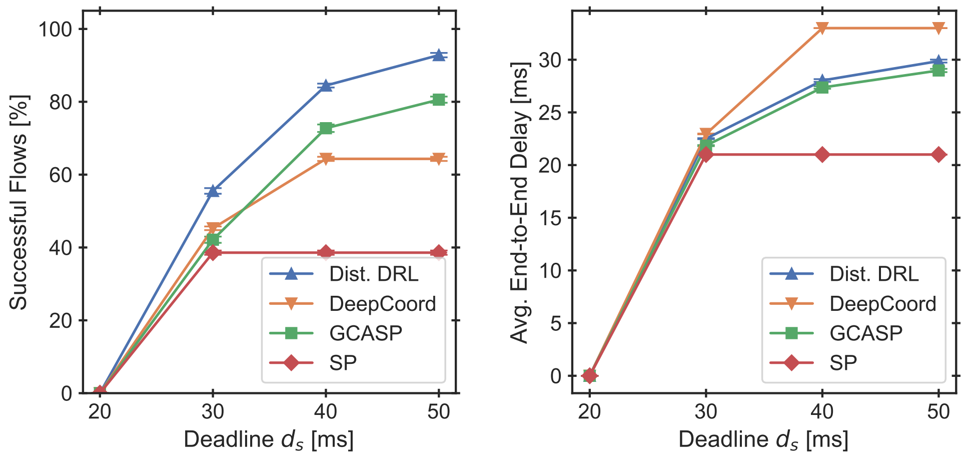
Figure 9.6.: Percentage of successful flows, extending Figure 9.5 with even more realistic traffic patterns (MMPP and real-world trace). Again, compared to other approaches, the distributed DRL approach processes most flows successfully. (Figure adapted from [S34]; © 2021 IEEE.)

9.5.3. Varying Deadlines

I now consider the base scenario (Section 9.5.1) with fixed ingress nodes (v_1, v_2), Poisson flow arrival and instead systematically vary the deadline for service s ($d_s \in \{20, 30, 40, 50\}$ ms). Figure 9.7 shows the resulting percentage of successful flows and corresponding average end-to-end delay of completed flows. With deadline $d_s = 20$ ms, all flows are dropped since flows need more than 20 ms to complete, even in the best case where flows are processed along the shortest paths. Indeed, the SP heuristic attempts to process all flows along the shortest paths, leading to a fixed average end-to-end delay of 21 ms for deadlines $d_s = 30$ ms and above. Further increasing the deadline does not increase the percentage of successful flows for SP. In contrast, the other algorithms exploit increasing deadlines and then also use longer paths to balance load. Overall, the proposed distributed DRL approach balances the load effectively, taking the given deadlines into account and processing more flows successfully than any other algorithm (21 % more than GCASP on average).

9.5.4. Generalization to Unseen Scenarios

In the previous sections, I evaluate how well the distributed DRL approach adapts to changing scenarios (traffic load, traffic patterns, deadlines) just by retraining but without human interaction. Here, I go one step further and investigate how well the trained DRL agent generalizes to previously unseen scenarios without any retraining. In practice, it is important that the DRL agent continues to coordinate services reasonably even if the scenario suddenly changes. A new DRL agent may be retrained periodically (or even continuously) to optimize for the current scenario. The frequency of periodic retraining depends on the available training resources and on how frequently the scenario changes, e.g., in terms of flows' ingress nodes or arrival



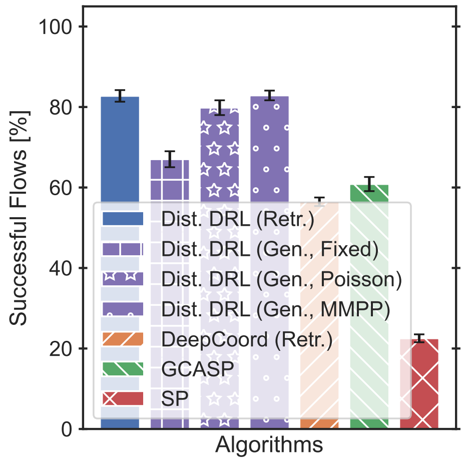
(a) Successful flows (b) Average end-to-end delay

Figure 9.7.: Percentage of successful flows and average end-to-end delay for varying deadlines. The distributed DRL approach adapts to the given deadlines, outperforming the other approaches. (Figure adapted from [S34]; © 2021 IEEE.)

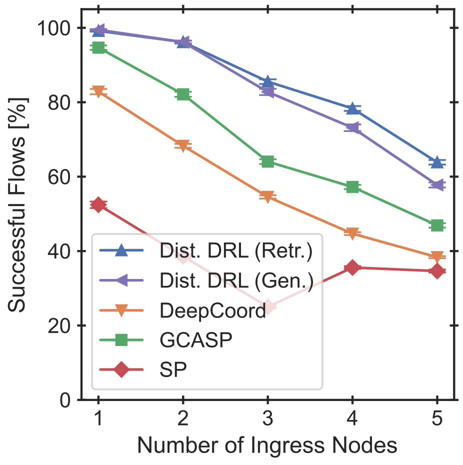
pattern. Either way, the incumbent DRL agent still has to coordinate decently until the new, retrained agent converges. To facilitate generalization, I designed the observation space to include generally available and useful information and normalized all values to be in a similar range (Section 9.4.2).

Here, I explore generalization to previously unseen traffic patterns with two ingress nodes in the base scenario (Section 9.5.1). To this end, I train the distributed DRL approach on fixed flow arrival and test it without retraining on unseen real-world traces. Similarly, I test generalization of the DRL agent trained on Poisson and MMPP traffic. Figure 9.8a shows the percentage of successful flows when generalizing these different DRL agents to unseen trace-driven traffic. For comparison, the figure also shows the success ratio of DeepCoord trained and tested on these traces and of the other algorithms. For the proposed distributed DRL approach, the success rates of the generalizing DRL agents ("Gen." in Figure 9.8a) are very close to the performance of the retrained agent ("Retr.") and still clearly outperform the other algorithms. This indicates that the proposed distributed DRL generalizes well to unseen traffic patterns.

Next, I investigate generalization to unseen network loads by training the distributed DRL approach on the base scenario with two ingress nodes (and Poisson traffic) and testing it with 1–5 ingress nodes, representing increasing load. Figure 9.8b shows the resulting successful flows in comparison with a DRL agent that is retrained for each scenario and with the other algorithms. Again, the generalizing DRL agent ("Gen.") processes almost as many flows successfully as the DRL agent that is retrained for each scenario ("Retr.") and still clearly outperforms all other algorithms.



(a) Unseen traffic patterns



(b) Unseen network load

Figure 9.8.: The distributed DRL approach generalizes to scenarios with traffic patterns and network load that were not previously seen during training. (Figure adapted from [S34]; © 2021 IEEE.)

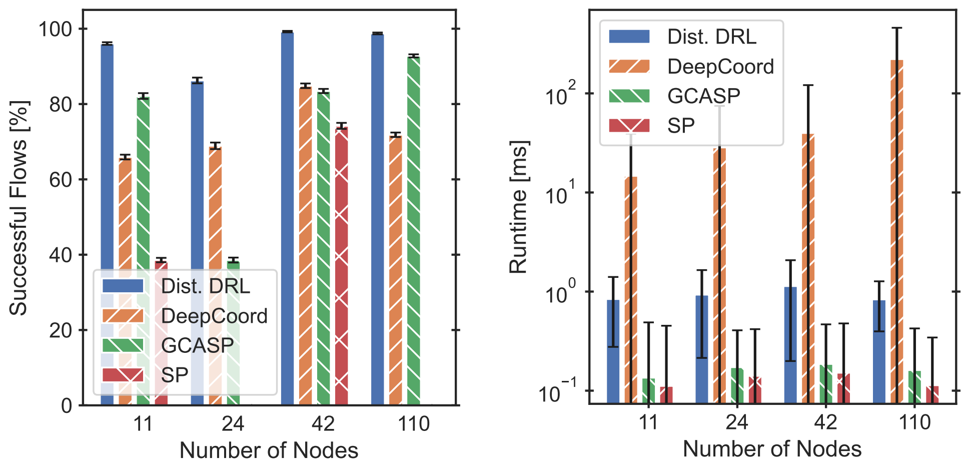
Table 9.1.: Real-world network topologies [142]

Network G	Nodes $\lvert V\rvert$	Links $\lvert L\rvert$	Degree (Min./Max. Δ_G/Avg.)
Abilene	11	14	2 / 3 / 2.55
BT Europe	24	37	1 / 13 / 3.08
China Telecom	42	66	1 / 20 / 3.14
Interroute	110	158	1 / 7 / 2.87

9.5.5. Scalability

Finally, I evaluate the scalability of the distributed DRL approach on large real-world network topologies. Specifically, I consider the BT Europe, China Telecom, and Interroute networks in addition to Abilene [142] (Table 9.1). Particularly the China Telecom network is highly skewed in terms of node degree, which influences the size of the observation and action spaces (Section 9.4.2). As before, I consider Poisson traffic at two ingress nodes (with node IDs v_1 and v_2), one egress (v_8) and randomly uniform node capacities (between 0 and 2) and link capacities (between 1 and 5).

Figure 9.9a shows the resulting percentage of successful flows on these networks. Despite the large size and node degree skewness of these networks, the distributed DRL approach achieves almost perfect results with close to 100 % successful flows. In the BT Europe network, link capacities between ingress and egress are scarce leading to more dropped flows for all algorithms. SP fails completely on BT Europe (24 nodes) and Interroute (110 nodes) since there are insufficient resources on the shortest paths. The distributed DRL approach outperforms DeepCoord by 31 % and GCASP by 42 % on average. At the same time, Figure 9.9b (logarithmic scale) shows that the inference time of the distributed DRL approach is on the order of 1 ms. In particular, it is magnitudes faster than the central DeepCoord approach and, unlike DeepCoord, invariant in the network size.



(a) Successful flows (b) Inference runtime

Figure 9.9.: Percentage of successful flows and inference runtime (logarithmic scale) with increasing network size. The distributed DRL approach scales well to large networks resulting in high success rates yet short runtime. (Figure adapted from [S34]; © 2021 IEEE.)

9.6. Conclusion

The presented distributed DRL approach combines ideas and strengths from Chapters 6 and 8: Similar to Chapter 8, the self-learning coordination approach using DRL self-adapts to varying scenarios and requires neither expert knowledge nor human intervention. As such, it is more flexible and better applicable in practice than conventional coordination approaches (Part I). Inspired by the fully distributed heuristics in Chapter 6, the approach combines centralized training with fully distributed inference. This allows fast, fine-grained, per-flow control (1 ms per decision), relying only on local observations and scaling to large real-world networks. As a result, the evaluation shows that the presented distributed DRL approach is more flexible, better scalable, and generally more successful than existing solutions. Overall, this makes it a better approach for autonomous network and service coordination in practice, particularly in large networks with many, short flows.

The following Chapter 10 explores how self-learning coordination using DRL can be extended and applied to wireless mobile networks. In comparison to wired networks, considered so far, there are similarities (e.g., competition for limited resources) but also new challenges in such wireless scenarios (e.g., user mobility). In these scenarios, Chapter 10 further investigates benefits and drawbacks of centralized vs. distributed DRL approaches.

10. Self-Learning Coordination for Mobile Networks

In the chapters so far, I consider network and service coordination in wired networks. In this chapter, I focus on coordination in wireless mobile networks where multiple moving users compete for limited radio resources. In such mobile networks, macrodiversity is a key technique to increase the capacity. Macrodiversity can be realized using Coordinated Multipoint (CoMP), simultaneously connecting users to multiple overlapping cells. Selecting which users to serve by how many and which cells is NP-hard but needs to happen continuously, in real time as users move and channel state changes. Existing approaches often require strict assumptions about or perfect knowledge of the underlying radio system, its resource allocation scheme, or user movements, none of which is readily available in practice. Instead, I propose three novel self-learning and self-adapting approaches using model-free Deep Reinforcement Learning (DRL): DeepCoMP, DD-CoMP, and D3-CoMP. DeepCoMP leverages central observations and control of all users to select cells almost optimally. DD-CoMP and D3-CoMP use multi-agent DRL, which allows distributed, robust, and highly scalable coordination. All three approaches learn from experience and self-adapt to varying scenarios, reaching 2x higher Quality of Experience (QoE) than other approaches. They have very few built-in assumptions and do not need prior system knowledge, making them more robust to change and better applicable in practice than existing approaches.

This chapter is based on my paper [S19]. I proposed the idea and concepts, discussed them with project partners, developed and evaluated all prototypes, and wrote the paper. This chapter contains verbatim copies of this paper [S19]: Stefan Schneider et al. "DeepCoMP: Coordinated Multipoint Using Multi-Agent Deep Reinforcement Learning." In: *IEEE Transactions on Network and Service Management (TNSM)* (2022). **Under Review** **Invited Talk at Ray Summit 2021** In this chapter and throughout my dissertation, I consistently write in the first-person singular form ("I" rather than "we") for ease of reading. The source code corresponding to this chapter is publicly available on GitHub [S13].

10.1. Introduction

Autonomous coordination is not just relevant for wired networks, discussed in previous chapters, but also has real-world use cases in wireless mobile networks (Section 2.3). Modern cellular networks coordinate resources across multiple terminals, base stations, cells, and access network entities. An example is CoMP, a kind of cooperative Multiple Input and Multiple Output Radio (MIMO) that leverages macrodiversity by allowing User Equipment (UE) to connect to and receive data from multiple cells simultaneously. CoMP was introduced in 3GPP Long-Term Evolution (LTE) Release 11 [1] but will play an even more important role in 5G and 6G with many small and partially overlapping cells [69].

As UEs move around, their channel state continuously changes due to path loss, shadowing, reflections, etc. Especially UEs at the cell edge experience strong path loss and benefit from connecting to multiple cells. At the same time, connections must be balanced across different cells as all UEs served by a cell compete for limited resources, e.g., Physical Resource Blocks (PRBs) in LTE. With more UEs connected to a cell, its available PRBs per UE decrease. Hence, a UE may connect to more cells to increase its effective data rate but thereby increases competition at these additional cells, possibly reducing the available PRBs and consequently the data rate of other connected UEs. To navigate this trade-off and adapt to UE movement, changing load and channel state, it is crucial to dynamically select how many and which cells should serve which UEs.

I focus on dynamic multi-cell selection in downlink CoMP with joint transmission and coordinated scheduling [1]. Instead of maximizing mere data rate, my goal is to also support maximizing QoE for all UEs. QoE depends on the considered scenario and may, for example, require reaching a certain data rate threshold or show diminishing returns with increasing data rate [77, 134]. Multi-cell selection is significantly more complex than single-cell selection and the resulting problem is NP-hard and cannot be reasonably approximated [15]. Despite this complexity, it must happen quickly online as user movement affects channel state and achievable data rates.

Existing approaches for multi-cell selection typically use heuristic algorithms or solve Mixed-Integer Linear Programs (MILPs) (Section 10.2). Often, these approaches build on rigid models or rely on perfect knowledge of the underlying system, including radio model, resource allocation scheme (i.e., intra-cell allocation of PRBs to connected UEs), or even user movement [15, 253, 169]. Such detailed knowledge may only be approximated or even be completely unavailable in practice (e.g., vendor-specific, unknown PRB allocation), limiting applicability of these approaches. Simpler approaches based on channel measurements and simple rules, e.g., [169, 27], may require different manual configurations for different scenarios and often lead to suboptimal results (Section 10.7).

Instead, I propose three novel self-learning and self-adaptive approaches, DeepCoMP, DD-CoMP, and D3-CoMP, using model-free DRL. DRL excels at sequential decisions to optimize long-term rewards [242] and is thus particularly useful here, where I want to optimize UEs' long-term QoE over multiple sequential cell assignments. Moreover, the proposed model-free approaches require very few built-in assumptions and learn from experience of previous actions. They do not need explicit and detailed system information but adapt to scenarios with varying and even heterogeneous resource allocation schemes and cell density. Unlike existing work [39, 270, 281], they support multiple moving UEs and autonomously adapt to varying UE numbers and movement. The approaches can continuously adapt to ongoing changes in the scenario through online transfer learning. Similarly, they support transfer learning for fine-tuning pretrained models to new, unseen situations without requiring expensive retraining from scratch (cf. Google AutoML [246]). Overall, the DRL approaches are not explicitly aware of the underlying wireless system details, which they abstract away, and instead self-adapt to any given scenario through indirect feedback.

In more detail, DeepCoMP is a centralized DRL approach that jointly coordinates all UEs. By central observation and control, DeepCoMP achieves close-to-optimal solutions but requires long training for large scenarios, depending on the number of active UEs. To address scenarios without global observations or with many UEs, I propose two variants: DD-CoMP and D3-CoMP. Both are multi-agent DRL approaches that observe and control UEs individually, in a distributed fashion. DD-CoMP trains a single neural network that is later replicated for distributed inference. D3-CoMP distributedly trains separate neural networks. While the proposed DRL approaches learn without human intervention, designing the underlying Partially Observable Markov Decision Process (POMDP) is known to be challenging for practical problems [242, 11, 63], which I solved by applying domain knowledge to the POMDP design. For example, I carefully design DD-CoMP's and D3-CoMP's observation and action spaces to be invariant in the number of UEs, allowing fast training even in large scenarios. For the observations and reward of each agent, I also take locally available information of surrounding UEs into account to encourage cooperative behavior, which is an open challenge in multi-agent DRL [274, 101]. Overall, the contributions of this chapter are:

- Section 10.4 proposes DeepCoMP as a centralized DRL approach, jointly selecting multiple cells for all UEs.
- Section 10.5 proposes DD-CoMP and D3-CoMP as distributed multi-agent DRL approaches that coordinate UEs individually and converge more quickly.
- Section 10.6 discusses architecture and options for practical deployment of DeepCoMP, DD-CoMP, and D3-CoMP.
- In Section 10.7, I evaluate adaptability, robustness, and scalability of the three DRL approaches and show that they consistently and significantly outperform existing approaches (2x higher QoE).
- All code is publicly available to encourage reproduction and reuse [S13].

10.2. Related Work

I first discuss related work proposing conventional approaches such as heuristic algorithms or MILPs in Section 10.2.1. I then compare related self-learning DRL approaches in Section 10.2.2.

10.2.1. Conventional Approaches Without DRL

Xenakis et al. [259] and Giust et al. [92] survey approaches for mobility management and dynamic cell selection, which are often addressed together with problems like resource allocation or power control. While many authors consider selection of a single cell [183, 167, 247], I focus on approaches that select multiple serving cells for CoMP. Qamar et al. [206] review different CoMP modes and survey corresponding literature. One of their identified open research issues is fairness in CoMP, which I encourage by optimizing a logarithmic utility function (representing QoE) [135]. I focus on CoMP joint transmission (CoMP-JT) with coordinated scheduling (CoMP-CS) but expect the proposed approaches to also learn to effectively select cells for other CoMP modes, e.g., CoMP coordinated beamforming (CoMP-CB), if trained in a corresponding environment.

Liang et al. [154] propose a heuristic algorithm selecting a fixed number of cells for CoMP-CB. For their heuristic, they assume knowledge of each cell's resource allocation scheme and model UE requirements in terms of fixed amounts of radio resources. In contrast, the DRL approaches proposed in this chapter are agnostic and unaware of cells' resource allocation and can adapt to various, even heterogeneous resource allocation schemes. My approaches optimize a generic utility function, which supports UEs with fixed resource requirements but also allows maximizing UEs' QoE, e.g., with diminishing returns for increasing data rate. Finally, Liang et al. always select exactly three cells for CoMP-CB, whereas my approaches dynamically vary the number of selected cells according to the current situation.

Marsch and Fettweis [169] statically cluster cells into fixed groups of given size, optimizing CoMP for a priori known UE positions. Vijayarani and Nithyanandan [253] dynamically optimize the number of serving cells for each UE without selecting the cells as such, but evaluating the expected throughput for all possible numbers of cooperating cells. All these examples require detailed knowledge of the underlying system and environment dynamics (e.g., UE positions, data rates, resource allocation). Similarly, using such detailed knowledge, Nigam et al. [190] analyze UE coverage for a given set of selected cells.

In contrast, I do neither assume detailed system knowledge or a priori knowledge of UE positions nor given and fixed group sizes of cooperating cells. The proposed DRL approaches, without such knowledge, dynamically decide both how many and which cells to select and consistently outperform algorithms with a fixed number of cells (Section 10.7).

Similar to the approaches proposed here, Amzallag et al. [15] optimize how many and which cells to select for CoMP and also consider fairness. They use an offline approach that requires a priori knowledge of UEs, cells, and system details over all time steps. You and Yuan [270] also dynamically select multiple cells for CoMP and additionally optimize resource allocation. While their problem is very similar to the one in this chapter, their approach is not. Unlike their conventional approach, the DRL approaches presented here neither require detailed, possibly unavailable system knowledge nor are they tied to any specific resource allocation scheme.

Beylerian and Ohtsuki [27] propose a simpler, rule-based approach for multi-cell selection where users connect to all cells above a Signal-to-Interference-and-Noise Ratio (SINR) threshold. As I show in Section 10.7, the threshold parameter strongly influences performance and the best value varies between scenarios, again requiring human expertise for proper configuration. In contrast, the proposed DRL approaches adapt to different and even heterogeneous schemes without explicit knowledge of these schemes through self-learning and consistently outperform this approach.

10.2.2. Self-Learning DRL Approaches

Self-learning approaches, particularly reinforcement learning [162], have become popular as they can be applied without detailed system knowledge and should adapt to different scenarios. Nasir and Guo [186] propose multi-agent DRL for dynamic power control. Their approach is comparable to DD-CoMP (Section 10.5) as it also relies on local observations to make distributed decisions after centralized training. I also propose D3-CoMP, which supports distributed training. They assume each UE to be connected to a single fixed cell and then control transmission power. I do not consider power control but focus on cell selection. The approaches are hence complementary. Similarly, Ozturk et al. [195] use supervised learning to predict UE movement, which could be combined with my approaches to focus training on expected UE positions and movement. Ayala-Romero et al. [19] jointly allocate radio and compute resources using autoencoders and DRL without requiring a priori system knowledge but adapting to different and even heterogeneous systems. Their approach could be combined with mine for self-adaptive multi-cell selection and resource allocation.

More related to the problem in this chapter, Chen et al. [39] use DRL to select base stations for Multi-Access Edge Computing (MEC) but only consider a single UE and connections to a single cell. Elsayed et al. [66] use tabular Q-learning for cell selection and interference mitigation in 5G by connecting UEs to a single cell, ignoring CoMP. Tabular Q-learning does not scale to large or continuous observations such that their approach is limited to very small and simple threshold-based observations, possibly leading to suboptimal results and restricting applicability to small scenarios. Niyato and Hossain [191] dynamically select a single network in HetNets (comparable to single-cell selection), rely on tabular Q-learning, and do not consider user mobility.

The work by Zhou et al. [281] is most related to this chapter. They also consider self-learning approaches for dynamic cell selection with UEs moving across many small, partially overlapping cells. Zhou et al. address the non-stationary nature of the problem by proposing a piece-wise stationary approach using bandits for regret minimization. To this end, they assume that UEs only move abruptly for short durations and otherwise stand still. In contrast, I support constantly moving UEs, making a piecewise stationary approach inapplicable. My proposed on-policy DRL approaches can learn continuously online and adjust to changes in the environment (unlike typical off-policy approaches [263]). Moreover, the authors focus on a single UE, only consider connections to one cell at a time, and maximize throughput directly. I support multiple moving UEs and simultaneous connections using CoMP. My proposed approaches support not only optimizing throughput but focus on deriving and optimizing UEs' QoE.

In general, I go beyond existing work by proposing three different DRL approaches geared towards different, realistic scenarios. In particular, the proposed approaches are among the first in the area to successfully use cooperative multi-agent DRL, which is known to be challenging [101], as well as online transfer learning. My DRL approaches make fewer limiting assumptions, e.g., about the number or movement of UEs, and optimize a more generic and complex utility (supporting QoE in addition to plain throughput), which I believe makes my approaches more meaningful and generally applicable.

10.3. Problem Statement

The self-learning DRL approaches proposed in this chapter require very few assumptions and only readily available information about UEs and cells in the area (current connections, SINR, and QoE per UE as detailed in Sections 10.4.1 and 10.5.2). I intentionally omit a formalization of radio models or other system details that are irrelevant for the DRL approaches. The example model used for evaluation is described in Section 10.7.1. Unlike previous chapters, this chapter focuses on coordinating multi-cell selection in wireless mobile networks rather than coordination in wired networks. Hence, this section explicitly states the coordination problem in wireless mobile networks and does not build on the problem statement in Chapter 3.

10.3.1. Parameters

I consider M UEs $u_j \in U$ and N cells $c_i \in C$, transmitting in discrete time steps $t = 1, 2, ..., T$, synchronized among cells. I focus on the downlink and assume each cell $c_i \in C$ to schedule its PRBs (or other units of radio resources) to connected UEs autonomously and transparently using CoMP coordinated scheduling (CoMP-CS) [168]. Inside one cell, a PRB is assigned to at most one UE, but one UE can be assigned multiple PRBs from one or several cells. I make no other assumptions about cells' resource allocation scheme, i.e., how or how many PRBs they assign to each connected UE. Considering multi-cell selection for uplink communication (using CoMP joint processing [119]) is interesting future work, where I expect similar problems and solutions as described here for downlink.

UEs $u_j \in U$ move around freely with varying direction and velocity, unknown to the system. Movement affects channels between UEs and connected cells, fluctuating the measured SINR over time. Here, in the context of cell selection, I focus on fluctuations on the order of seconds and assume that fast-fading effects on the order of sub-milliseconds are handled and averaged out in the physical layer. I denote by $\text{SINR}_{i,j}(t)$ the measured SINR from cell c_i to UE u_j at time t.

UE u_j can only connect to and receive data from cell c_i if the SINR is above a given threshold γ_{SINR}. Assuming it can connect, the effective downlink data rate $r_{i,j}(t)$ from cell c_i to UE u_j at time t depends on cell-internal resource allocation, e.g., power, PRBs, antennas [146]. The proposed DRL approaches are neither explicitly aware of $r_{i,j}(t)$ nor of the cells' resource allocation but learn to adapt to a given scenario through feedback of UEs' QoE.

10.3.2. Decision Variables

I express cell selection by binary decision variable $x_{i,j}(t) \in \{0, 1\}$, indicating whether cell c_i serves UE u_j at time t. To allow $x_{i,j}(t) = 1$, the corresponding SINR must be above threshold γ_{SINR}. A UE's total rate is simply the sum of its cell rates (considering CoMP-CS [146]):

$$r_j(t) = \sum_{i=1}^{N} x_{i,j}(t) r_{i,j}(t) \tag{10.1}$$

Deciding which cells serve which UEs may be driven completely by the network or UEs may trigger or assist cell connections themselves (Section 10.6). To limit signaling overhead, UEs can connect to or disconnect from at most one cell per time step. Frequent connection changes are still possible as time steps are typically quite short, depending on the acceptable signaling overhead in a given scenario.

10.3.3. Objective: Optimize Quality of Experience (QoE)

Rather than simply maximizing the data rate per UE, as commonly done in related work [253, 281, 66], I am mostly interested in optimizing UEs' QoE, which better reflects users' satisfaction and strongly affects operators' financial success [122]. UEs' QoE typically increases roughly logarithmically with increasing data rate, i.e., with diminishing returns [77, 134]. I quantify QoE of UE u_j by its *utility* $U_j(t)$. Using a logarithmic function as example, $U_j(t)$ could be formalized as

$$U_j(t) = \min\left\{U_j^{\max}, \max\left\{U_j^{\min}, w_1 log_{w_2}(w_3 + r_j(t))\right\}\right\} \tag{10.2}$$

where w_1, w_2, w_3 are configurable weights and limits $U_j^{\min}$, $U_j^{\max}$ ensure that the utility is finite and well defined. An advantage of using logarithmic utility is that maximizing the sum of logarithms is equivalent to ensuring a natural concept of fairness, namely proportional fairness [135]. Nonetheless, logarithmic utility is just an example based on previous studies [77, 134]. The proposed approaches are not tied to this particular utility function but can also maximize plain data rate or adapt to other utility functions as shown in Section 10.7.2. In fact, they do not even require the utility function to be known. Instead, it suffices to approximate the instantaneous QoE of each UE (i.e., their utility), which is possible from the network side without relying on UEs reporting their QoE truthfully [173]. The overall goal is, therefore, to maximize the long-term utility, averaged over all UEs and time steps:

$$\text{Average QoE:} \qquad \max \lim_{T\to\infty} \frac{1}{T}\frac{1}{M} \sum_{t\in T, j\in\{1,\dots,M\}} U_j(t) \tag{10.3}$$

The proposed self-learning DRL approaches approximate and optimize this long-term utility through typical discounted rewards. I present the details of these DRL approaches next: DeepCoMP in Section 10.4, DD-CoMP and D3-CoMP both in Section 10.5, and a discussion of architecture and deployment options in Section 10.6.

10.4. DeepCoMP: Centralized DRL

DeepCoMP is a centralized DRL approach that simultaneously observes and controls all UEs in an area with a single DRL agent. While this requires centrally collecting observations of all UEs as well as centralized control, it enables DeepCoMP to learn close-to-optimal policies. These observations contain SINR and QoE estimates, but no detailed system information, which may be unavailable. Section 10.4.1 details DeepCoMP's observations, actions, and reward and Section 10.4.2 presents the overall algorithm. In Section 10.6, I discuss how DeepCoMP could be deployed in practice.

10.4.1. Partially Observable Markov Decision Process (POMDP)

Due to the complexity of the mobile scenario, it is unrealistic to capture the complete environment state, even for a centralized agent. Instead, I focus on partial observations that could be available in practice. The POMDP consists of tuple $(\mathcal{O}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ with observations $\mathcal{O}$, actions $\mathcal{A}$, unknown environment dynamics $\mathcal{P}$, and reward function $\mathcal{R}$, defined next. Figure 10.1 shows an example of DeepCoMP's observations, actions, and reward.

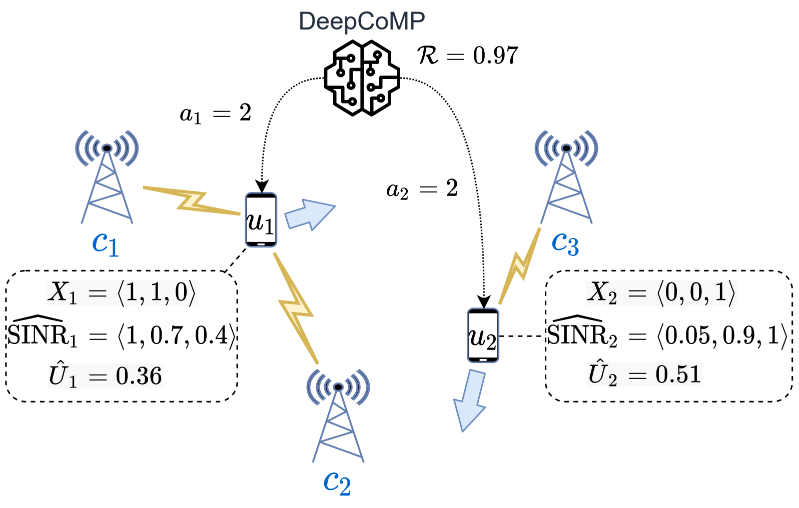

DeepCoMP
$\mathcal{R} = 0.97$
$a_1 = 2$
$a_2 = 2$
c_1
c_2
c_3
u_1
u_2
$X_1 = \langle 1,1,0 \rangle$
$\widehat{\text{SINR}}_1 = \langle 1, 0.7, 0.4 \rangle$
$\hat{U}_1 = 0.36$
$X_2 = \langle 0,0,1 \rangle$
$\widehat{\text{SINR}}_2 = \langle 0.05, 0.9, 1 \rangle$
$\hat{U}_2 = 0.51$

Figure 10.1.: Illustration of DeepCoMP's POMDP. (Figure from [S19].)

Observations $\mathcal{O}$

In each time step, DeepCoMP only observes the UEs' current connections, their SINR, and their utility, which are readily available (Section 10.6). Specifically, the DeepCoMP agent observes $O = \langle\langle X_j, \widehat{\text{SINR}_j}, \hat{U}_j\rangle | \forall j \in \{1, ..., M\}\rangle$ for each cell c_i and UE u_j. Observations are grouped into vectors X_j, $\widehat{\text{SINR}_j}$, and $\hat{U}_j$ for each UE u_j, containing elements for each cell c_i as detailed below. I normalize all observations to be in range $[-1, 1]$ (or $[0, 1]$), which is important for effective training and generalization of neural networks [118]. Otherwise, stronger observation signals with a larger range could drown out weaker signals with a smaller range, making the DRL agent "blind" to such observations.

Current Connections $X_j = \langle x_{i,j}(t) | \forall i \in \{1, ..., N\}\rangle \in \{0, 1\}^N$ indicates to which cells c_i a UE u_j is currently connected and directly corresponds $x_{i,j}(t)$ (Section 10.3.2). UEs hold their connections unless they are told explicitly to connect or disconnect or until they move away from a connected cell.

SINR $\widehat{\text{SINR}_j} = \langle \frac{\text{SINR}_{i,j}(t)}{\max_{i'} \text{SINR}_{i',j}(t)} | \forall i \in \{1, ..., N\}\rangle \in [0, 1]^N$ is the normalized $\text{SINR}_{i,j}(t)$ between each cell c_i and UE u_j. Specifically, $\text{SINR}_{i,j}(t)$ is normalized by the maximum SINR of all cells for UE u_j at time t. This maps the observed SINR of the strongest cell to 1 and highlights the differences in SINR between the available cells, simplifying cell selection. In the special case that a UE is far off any cells (with zero SINR), I set the vector to all zeros to avoid division by zero.

Utility The normalized utility of each UE u_j is $\hat{U}_j = 2 \cdot \frac{U_j(t) - U_j^{\min}}{U_j^{\max} - U_j^{\min}} - 1 \in [-1, 1]$, i.e., $U_j(t)$ scaled to range $[-1, 1]$ based on bounds $U_j^{\min}$, $U_j^{\max}$ (Section 10.3.3). If these bounds are not known explicitly, they can be approximated by keeping track of the lowest and highest $U_j(t)$ over all UEs and time steps so far. Observation $\hat{U}_j$ provides valuable information about each UE's current QoE, which is affected by the observed SINR but also by other factors that are not directly observable, e.g., cells' resource allocation and UE movement. Again, note that the current utility $U_j(t)$ of each UE can be approximated locally without relying on UEs to report their QoE truthfully [173].

Actions $\mathcal{A}$

To limit protocol overhead, each UE can either connect to or disconnect from at most one cell per time step. This also simplifies and shrinks the action space considerably compared to alternatives like choice of arbitrary subsets of cells. I design the corresponding action space as $\mathcal{A} = \langle a_j | \forall j \in \{1, ..., M\}\rangle \in \{0, 1, ..., N\}^M$. Hence, DeepCoMP selects for each UE u_j either a specific cell c_i, where it toggles the connection status, or a no-op. Action $a_j = i \in \{1, ..., N\}$ means that u_j's connection to cell c_i should be toggled, i.e., establishing a connection if u_j and c_i are not yet connected or disconnecting otherwise. Alternatively, $a_j = 0$ indicates a no-op, where all of u_j's connections remain unchanged.

Reward $\mathcal{R}$

DeepCoMP's reward in time step t is the average of current utilities over all UEs, using $\hat{U}_j \in [-1, 1]$ as defined in Section 10.4.1 for the reward: $\mathcal{R} = \frac{1}{M} \sum_{j \in \{1,...,M\}} \hat{U}_j$. This corresponds to the goal of maximizing the average QoE over all UEs (Section 10.3.3). Internally, the DRL agent maximizes the sum of discounted rewards to optimize long-term utility.

Varying Number of UEs

Since DeepCoMP observes and controls all UEs, the size of its observation and action spaces depends on the number of active UEs in the area, which may change over time. Observations and actions, on the other hand, are used as inputs and outputs of a neural network and, thus, need to be of fixed size. Even if the number M of active UEs in the area varies over time, the size of these observations and actions must not change. To this end, I define $M^{\max}$ as the maximum number of supported active UEs in the area and set the size of the observations and actions to $M^{\max}$. If $M < M^{\max}$, related observations are padded with zeros to ensure consistent size of observations and to indicate which UEs are currently missing. Resulting actions for $j \in \{M+1, ..., M^{\max}\}$, i.e., for non-existing UEs, are simply ignored. Overall, the observation and action spaces of DeepCoMP grow linearly with $M^{\max}$. The distributed DRL approaches (Section 10.5) are invariant in $M^{\max}$ and therefore suitable for larger scenarios.

10.4.2. DeepCoMP Algorithm

Algorithm 8 DeepCoMP: Centralized Training and Inference

```
1: Initialize $\pi_\theta, V_\phi, b$
2: for $t \in \{1, ..., T\}$ do
3:     for $u_j \in U$ do
4:         $o_j, r_j \leftarrow$ get_obs_and_r($u_j$)
5:     $o_t \leftarrow \langle o_j | \forall j \in \{1, ..., M\}\rangle \cup \langle 0 | \forall j \in \{M+1, ..., M^{\max}\}\rangle$
6:     $r_t \leftarrow \frac{1}{M} \sum_{j \in \{1,...,M\}} r_j$
7:     $b \overset{\text{add}}{\longleftarrow} (o_{t-1}, a_{t-1}, r_t, o_t)$
8:     $a_t \leftarrow \pi_\theta(o_t)$
9:     for $a_j \in a_t$ with $j \in \{1, ..., M\}$ do
10:        $x_{i,j}(t) \leftarrow$ set_conn($u_j, a_j$)
11:    if training and $b$ is full then
12:        Train $V_\phi$ using temporal difference updates [242]
13:        Train $\pi_\theta$ maximizing $\mathbb{E}[\sum_i \gamma^i r(o_{t+i}, a_{t+i})]$
```

I leverage Proximal Policy Optimization (PPO) [228], which is a state-of-the-art actor-critic DRL algorithm, to train the DeepCoMP DRL agent. Algorithm 8 shows the centralized training and inference procedure for DeepCoMP. DeepCoMP starts training by initializing stochastic policy π_θ (actor), value function V_ϕ (critic), and mini-batch b (line 1). DeepCoMP then observes and acts in each time step for all UEs simultaneously. First, it collects observations and rewards and constructs the observation vector and total reward as defined in Section 10.4.1 with function `get_obs_and_r`(u_j) for each UE u_j (lines 3–6). It then adds the experience (observation, action, reward, next observation) to mini-batch b (line 7) and selects an action a_t for the observation vector o_t according to policy π_θ (line 8). Action a_t is a vector that specifies the cell selection for all UEs and is applied to the environment by setting decision variable $x_{i,j}(t)$ accordingly (lines 9–10). Function `set_conn`(u_j, a_j) applies action a_j for UE u_j by dis-/connecting from/to the selected cell c_j if $a_j > 0$ and the selected cell is in range (a_j *toggles* connectivity).

During training, critic V_ϕ's weights are updated whenever mini-batch b is full (line 12). Using standard temporal difference updates [242], critic V_ϕ is trained to approximate long-term value $V_\phi(o)$ of receiving an observation o and then following policy π_θ. In turn, $V_\phi(o)$ is used to calculate advantage $A(o, a)$, which is the relative value of taking an action a after receiving observation o compared to all other actions. Finally, advantage A is used inside the policy update when training actor π_θ to optimize the long-term discounted reward (line 13), which corresponds to maximizing the average QoE, i.e., the objective defined in Section 10.3.3 [116].

I repeat this procedure over many episodes to train DeepCoMP offline and then switch to quick online inference upon training convergence. DeepCoMP also supports continuous online training, which I evaluate in Section 10.7.3. Here, training (lines 11–13) happens asynchronously to avoid blocking online inference. With a fixed number of hidden units, time and space complexity for inference with DeepCoMP is in $O(M^{\max} \cdot N)$ based on the size of the observation and action spaces (Sections 10.4.1 and 10.4.1).

10.5. DD-CoMP & D3-CoMP: Distributed DRL

DeepCoMP leverages global knowledge and simultaneous control of all UEs to learn highly optimized policies. In practice, however, observations from all UEs may not be available centrally or only with significant overhead, preventing fast centralized inference. Furthermore, its observation and action spaces grow with the (maximum) number of active UEs in the network. I hence propose distributed DeepCoMP (DD-CoMP), which only requires local observations and control per UE. DD-CoMP is a multi-agent DRL approach that trains a *logically centralized* neural network, which is *replicated* for distributed inference. I also propose D3-CoMP, which *trains separate* neural networks for each UE. Table 10.1 summarizes these differences in training and inference between the three DRL approaches as well as deployment options, which I discuss in Section 10.6.

Table 10.1.: Proposed DRL approaches

DRL Approach	Training	Inference	Deployment
DeepCoMP (Section 10.4)	Centralized	Centralized	Core network
DD-CoMP (Section 10.5)	Centralized	Distributed	Network or UEs
D3-CoMP (Section 10.5)	Distributed	Distributed	Network or UEs

10.5.1. Trade-Offs and Design Choices

Decisions per Cell vs. per UE

Using multi-agent DRL, an obvious idea would be to assign one agent per cell, controlling its connections. However, this leads to the same complications as discussed in Section 10.4.1, where the number of UEs fluctuates but the size of the observation and action spaces are fixed and thus need to be padded. The alternative of assigning one agent per UE appears more suitable as the number of cells is rather fixed. As UEs arrive and depart over time, DRL agents can be spawned/terminated on demand. Using separate agents per UE does not predicate the networking architecture/location of control (Section 10.6).

Using separate DRL agents for each UE, I can limit each agent's observations to only observe a single UE instead of all UEs, where relevant observations are available locally at the UE (Section 10.5.2). Similarly, the agent's actions determine the UE's cell selection, which can be applied locally without much communication overhead. In large areas with many cells, the observation and action spaces could be reduced by limiting agents' observations and actions to each UE's k closest cells, which are most relevant for cell selection. A remaining challenge with this approach is that each agent only observes and controls a single UE. This easily leads to greedy and unfair behavior and ultimately lower average QoE if agents do not consider or observe other UEs' utility. To overcome this challenge, I slightly adjust DeepCoMP's POMDP for DD-CoMP and D3-CoMP so that it takes other, nearby UEs into account (Section 10.5.2).

DD-CoMP vs. D3-CoMP

DD-CoMP and D3-CoMP only differ in their training architecture. DD-CoMP trains *a single, logically centralized* neural network for actor π_θ and critic V_ϕ, respectively. It collects experience from all DRL agents in joint mini-batches. While DD-CoMP's training is logically centralized, it could be implemented in a distributed fashion with minimal adjustments. For example, each agent may update its neural network locally and only share the computed gradient updates to synchronize the neural network weights across all DRL agents (cf. federated learning [267]).

D3-CoMP trains *separate, logically decentralized* neural networks for each agent/UE. D3-CoMP can hence learn individual cell selection policies per UE, allowing DRL agents to adapt to heterogeneous UE characteristics that affect QoE but are not explicitly observed, e.g., UE velocity or movement patterns. Furthermore, D3-CoMP does not exchange agents' experiences or gradients during training, lowering overhead. Training decentralized DRL agents independently easily leads to greedy or adversarial policies, which I address by coupling the reward of nearby, competing UEs (cf. best response strategy [109]). In comparison, DD-CoMP's central neural network is trained with more diverse data (from all UEs) than each of D3-CoMP's distributed DRL agents. Hence, DD-CoMP often learns more robust policies (Section 10.7).

10.5.2. Partially Observable Markov Decision Process (POMDP)

For DD-CoMP and D3-CoMP, I adjust DeepCoMP's POMDP from Section 10.4.1. Instead of vectors containing observations and actions for all UEs, agents only observe and control cell selection for a single UE u_j. To allow some awareness of nearby UEs and improve fairness, I extend the observation space and adjust the reward function.

Observations $\mathcal{O}$

For DD-CoMP and D3-CoMP, $O = \langle X_j, \widehat{\text{SINR}_j}, \hat{U}_j, \hat{U}^{\text{avg}}, \hat{M} \rangle$ includes $X_j, \widehat{\text{SINR}_j}, \hat{U}_j$ as defined in Section 10.4.1. While DeepCoMP receives these observations as a vector for all UEs, DD-CoMP and D3-CoMP just observe a single UE u_j. Additionally, I assume that cells collect and broadcast aggregated information $\hat{U}^{\text{avg}}$ and $\hat{M}$ about their connected UEs. Specifically, cells approximate the instantaneous QoE of their connected UEs [173] (or rely on QoE reported by the UEs) and calculate their corresponding utility $\hat{U}_j$ as defined in Section 10.4.1. Each cell c_i then broadcasts the average utility of all connected UEs as $\hat{U}_i^{\text{avg}} = \frac{1}{M_i} \sum_{j'} x_{i,j'}(t) \cdot \hat{U}_{j'}$, where $M_i = \sum_j x_{i,j}(t)$ denotes the number of currently connected UEs at c_i with $j, j' \in \{1, ..., M\}$. If no UEs are currently connected ($M_i = 0$), I set $\hat{U}_i^{\text{avg}}$ to zero. Similarly, if the observing UE u_j is too far away from a cell c_i to receive its broadcast (i.e., $\text{SINR}_{i,j}(t) < \gamma_{\text{SINR}}$), the UE assumes $M_i = 0$ and $\hat{U}_i^{\text{avg}} = 0$ as observations. Finally, vector $\hat{U}^{\text{avg}} = \langle \hat{U}_i^{\text{avg}} | \forall i \in \{1, ..., N\} \rangle \in [-1, 1]^N$ contains the observed average utility of all cells. Hence, vector $\hat{U}^{\text{avg}}$ can be observed locally by each DRL agent and helps the agent to select suitable cells with high $\hat{U}_i^{\text{avg}}$ to avoid competing for radio resources at cells with already low average utility.

To give the DRL agent a better sense of the load at different cells, cells also broadcast the number of currently connected UEs M_i as defined above. Based on the received M_i (if not received, $M_i = 0$), the DRL agent calculates the normalized load per cell $\hat{M}_i = \frac{M_i}{\sum_{i' \in \{1,...,N\}} M_{i'}}$, setting $\hat{M}_i = 0$ if $M_i = 0$. It then observes vector $\hat{M} = \langle \hat{M}_i | \forall i \in \{1, ..., N\} \rangle \in [0, 1]^N$ containing the normalized amount of connected UEs at each cell. This helps to avoid congested cells and identify free cells without any UEs, i.e., without competition for radio resources. The signaling overhead for $\hat{U}^{\text{avg}}$ and $\hat{M}$ is small and constant (Section 10.6.2). Centralized DeepCoMP does not require additional observations $\hat{U}^{\text{avg}}$ and $\hat{M}$ as it observes all UEs' connections and their QoE individually.

Actions $\mathcal{A}$

Actions of DD-CoMP and D3-CoMP are similar to DeepCoMP but refer to a single UE. Particularly, each DRL agent makes an action $\mathcal{A} = a \in \{0, 1, ..., N\}$, controlling cell selection for its UE u_j as defined in Section 10.4.1.

Reward $\mathcal{R}$

An obvious choice for the reward function of a DRL agent controlling UE u_j would be $\mathcal{R} = \hat{U}_j$, i.e., rewarding high utility of the corresponding UE. However, this reward would encourage agents to greedily optimize their UE's utility: Agents would tend to increase their UE's data rate even if it only marginally increases their utility. As a consequence, this could significantly harm other UEs' utilities and overall average utility, which is the main objective (Section 10.3.3).

Instead, I define $\mathcal{R} = \sum_{i \in \{1,...,N\}} \hat{U}_i^{\text{avg}} \cdot \hat{M}_i \in [-1, 1]$ as the average utility at all cells in range (i.e., with received broadcasts), weighted by their normalized amount of connected UEs $\hat{M}_i$. If UE u_j is connected to some cell(s), its utility is already included in the sum above. Otherwise, if u_j is not connected to any cells, I explicitly include its weighted utility in the reward and define

$\mathcal{R} = \sum_{i \in \{1,\dots,N\}} \hat{U}_i^{\text{avg}} \cdot \hat{M}_i + \frac{\hat{U}_j}{\sum_{i' \in \{1,\dots,N\}} M_{i'}}$. The UE's own utility is therefore always considered but does not dominate the reward. Hence, this reward function encourages DRL agents to maximize the average utility over all UEs and not just the utility of their own UE.

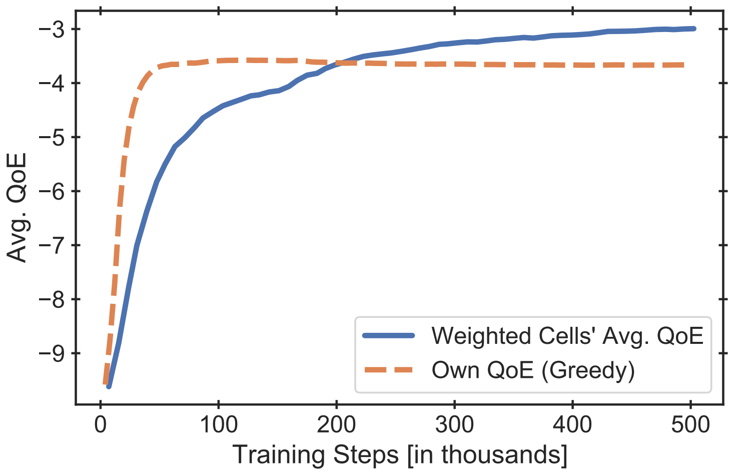


Figure 10.2.: DD-CoMP learning curves with different reward (figure from [S19]).

Figure 10.2 illustrates the average QoE (Section 10.3.3) for the two reward formulations in an example scenario. Using just the own UE's QoE as reward leads to greedy behavior, which is easier to learn and leads to faster yet clearly suboptimal convergence. The more sophisticated reward based on weighted average QoE at surrounding cells requires negligible signaling overhead (Section 10.6.2) and still leads to quick convergence and ultimately to a significantly better policy (20 % higher QoE).

10.5.3. DD-CoMP Algorithm

DD-CoMP agents observe and control each UE's connections separately in parallel (Algorithm 9, lines 3–7) rather than jointly for all UEs like DeepCoMP. Still, experiences from all UEs are added to the same mini-batch b and train a central neural network for actor π_θ and critic V_ϕ (lines 8–10). After logically centralized training, DD-CoMP performs distributed inference with separate DRL agents, each using a copy of the trained actor network (line 11) for local inference (lines 12–16).

To support continuous training, all DRL agents need to continue sending their UEs' experiences to central batch b to train the the logically centralized neural network. Again, sharing and training happens asynchronously to avoid blocking inference. For quick inference, the distributed DRL agents could regularly update their local copies π_{θ_j}. Assuming a fixed number of hidden units, space and time complexity of inference itself is in $O(N)$, i.e., it is invariant in the number of UEs M and linear only in the number of cells N. If N is large, space and time complexity could further be reduced to be constant by only considering the k closest cells per UE (Section 10.5.1).

If the number of UEs in the area varies over time, the number of DRL agents is adjusted accordingly. If a new UE arrives, a new DRL agent is instantiated, and removed again when

Algorithm 9 DD-CoMP: Centralized Training, Distributed Inference

```
 1: Initialize $\pi_\theta, V_\phi, b$                                   ▷ Training
 2: for $t \in \{1, ..., T\}$ do
 3:     for $u_j \in U$ in parallel do
 4:         $o_j^t, r_j^t \leftarrow$ get_obs_and_r($u_j$)
 5:         $b \xleftarrow{\text{add}} (o_j^{t-1}, a_j^{t-1}, r_j^t, o_j^t)$
 6:         $a_j^t \leftarrow \pi_\theta(o_j^t)$
 7:         $x_{i,j}(t) \leftarrow$ set_conn($u_j, a_j^t$)
 8:     if $b$ is full then
 9:         Train $V_\phi$ using temporal difference updates [242]
10:         Train $\pi_\theta$ maximizing $\mathbb{E}[\sum_{j,k} \gamma^k r(o_j^{t+k}, a_j^{t+k})]$
11: Deploy a copy $\pi_{\theta_j}$ of $\pi_\theta$ for each UE $u_j \in U$        ▷ Inference
12: for $t \in \{1, ..., T\}$ do
13:     for $u_j \in U$ in parallel do
14:         $o_j^t, r_j^t \leftarrow$ get_obs_and_r($u_j$)
15:         $a_j^t \leftarrow \pi_{\theta_j}(o_j^t)$
16:         $x_{i,j}(t) \leftarrow$ set_conn($u_j, a_j^t$)
```

the UE leaves the area. All DRL agents access the logically centralized neural network and thus directly start with a good policy, benefiting from experience of other UEs. Note that, while all UEs share the same neural network weights, their experiences are never shared with each other directly.

10.5.4. D3-CoMP Algorithm

Algorithm 10 shows the distributed D3-CoMP training and inference procedure. While the main procedure is similar to DeepCoMP and DD-CoMP, D3-CoMP directly initializes M different actor π_{θ_j} and critic V_{ϕ_j} networks for each UE u_j (line 1). Experiences from different UEs are added to different mini-batches b_j (line 5). Consequently, actor and critic of each DRL agent are trained only on the corresponding UE's experiences (lines 8–10). Similar to DD-CoMP, inference time and space complexity is in $O(N)$.

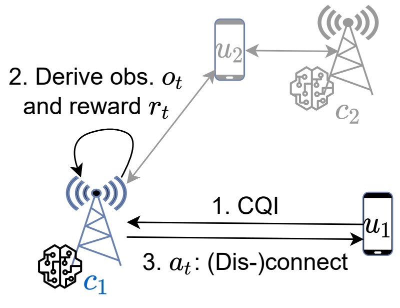
D3-CoMP also automatically adjusts the number of DRL agents to the number of UEs in the area, starting new DRL agents when UEs arrive and terminating them once UEs leave. As D3-CoMP trains separate policies per UE, it typically initializes new neural networks for newly spawned DRL agents. However, trained neural network weights for a UE could be stored when the UE leaves the network and loaded again (e.g., based on a UE identifier) when it returns to avoid retraining from scratch. New UEs (with a previously unseen ID) could start with a trained and stored policy of another UE (e.g., with similar velocity or movement pattern). A hybrid solution between DD-CoMP and D3-CoMP is also conceivable, where a shared default policy is trained centrally up front by DD-CoMP and then used as starting point for new DRL agents in D3-CoMP when new UEs arrive in the network.

Algorithm 10 D3-CoMP: Distributed Training and Inference

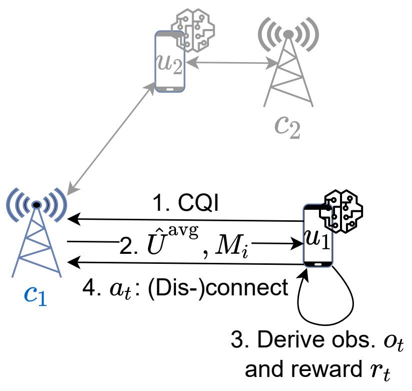
```
1: Initialize $\pi_{\theta_j}, V_{\phi_j}, b_j \quad \forall j \in \{1, ..., M\}$
2: for $t \in \{1, ..., T\}$ do
3:     for $u_j \in U$ in parallel do
4:         $o_j^t, r_j^t \leftarrow$ get_obs_and_r($u_j$)
5:         $b_j \xleftarrow{\text{add}} (o_j^{t-1}, a_j^{t-1}, r_j^t, o_j^t)$
6:         $a_j^t \leftarrow \pi_{\theta_j}(o_j^t)$
7:         $x_{i,j}(t) \leftarrow$ set_conn($u_j, a_j^t$)
8:     if training and $b_j$ is full then
9:         Train $V_{\phi_j}$ using temporal difference updates [242]
10:        Train $\pi_{\theta_j}$ maximizing $\mathbb{E}[\sum_k \gamma^k r(o_j^{t+k}, a_j^{t+k})]$
```

10.6. Architecture and Deployment Options

There are different architectural options for deploying DeepCoMP, DD-CoMP, and D3-CoMP, summarized in Table 10.1. All three approaches can be deployed in the network for network-initiated cell selection. Depending on available resources and latency requirements, deployment could be in the core, edge, or access network [243]. Alternatively, DD-CoMP and D3-CoMP also support distributed deployment of DRL agents at the UEs, enabling UE-initiated cell selection. Both network-controlled and UE-based cell selection are supported and relevant in 5G [187]. In both cases, the DRL approaches can leverage already available data with low overhead. In the following, I detail the two architectural options (Sections 10.6.1 and 10.6.2), which are illustrated in Figure 10.3, and discuss training and inference in practice (Section 10.6.3).



(a) Network-based deployment


u_2
c_2
1. CQI
2. $\hat{U}^{\text{avg}}, M_i$
u_1
4. a_t: (Dis-)connect
c_1
3. Derive obs. o_t and reward r_t

(b) UE-based deployment

Figure 10.3.: Architecture and deployment options: a) DRL agents deployed in the network for network-initiated cell selection. b) Deployed at each UE for UE-initiated cell selection. (Figure from [S19].)

10.6.1. Network-Based Deployment

The centralized DeepCoMP approach is designed to be deployed on the network side for network-initiated cell selection. Here, the agent collects available data and exchanges control messages between cells for centralized observations and actions. The entire network could be split into different areas (e.g., based on cell coverage or business aspects) that are controlled independently by separate DeepCoMP agents. Within each area, a single DeepCoMP agent is deployed, which should be pretrained (e.g., in simulation) as further discussed in Section 10.6.3. Inside an area, the responsible DeepCoMP agent observes and controls all active UEs.

For the agent's observations, the network could keep track of each UE's current connections and collect the UEs' SINR and QoE (step 1 in Figure 10.3a). I emphasize, again, that this approach does not assume any predefined, known utility function (e.g., a logarithmic function) but only requires instantaneous approximations of UEs' QoE. To do so, UEs could either report their SINR and QoE directly or cells could use *already available information*, e.g., UEs' Channel Quality Indicators (CQIs) and Quality of Service (QoS) measurements, to approximate SINR and QoE locally [173, 35]. In doing so, cells can locally process available data to derive the necessary observations (Section 10.4.1) and reward (Section 10.4.1) in step 2. In step 3, the processed observations are passed through DeepCoMP's actor neural network to obtain the next action, which selects suitable cells for all UEs and triggers necessary connections or disconnections.

DD-CoMP and D3-CoMP can also be deployed in the network for network-initiated cell selection. This works similarly as with DeepCoMP but, here, with multiple DRL agents being deployed for the different active UEs in the area, spawning and terminating DRL agents on demand (as discussed in Sections 10.5.3 and 10.5.4). These agents could be deployed at different cells, where each one only requires observations and reward of its corresponding UE rather than data from all UEs. If DD-CoMP and D3-CoMP agents are deployed in the network, cells can locally compute and exchange their average utility $\hat{U}^{\text{avg}}$ and load $\hat{M}$, which are required as observations and for the reward (Section 10.5.2), without the need to broadcast them.

10.6.2. UE-Based Deployment

Alternatively, each DRL agent could be deployed directly at a UE for user-initiated cell selection. Deploying DeepCoMP at the UEs is theoretically feasible but impractical since it requires each UE to collect observations from all other UEs. Instead, DD-CoMP and D3-CoMP are suitable for UE-based deployment because they rely on local observations and control. Each cell c_i still needs the QoE of its connected UEs, which the cells can approximate locally based on UEs' reported CQIs [173, 35] (step 1 in Figure 10.3b). In step 2, cells calculate and broadcast their average utility $\hat{U}_i^{\text{avg}}$ and number M_i of connected UEs (from which $\hat{M}$ can be derived). In step 3, the DRL agents at each UE derive the current observations and reward (Section 10.5.2) based on the received broadcasts and locally observed X_j, $\widehat{\text{SINR}}_j$, and $\hat{U}_j$. Finally, each DRL agent uses its actor neural network to choose an action for its UE and initiates the corresponding connections or disconnections (step 4).

Cells' broadcasts of $\hat{U}_i^{\text{avg}}$ and M_i have small, constant size and could be included in existing System Information Broadcasts (SIBs) [2] without requiring extra signaling messages. Hence, DD-CoMP and D3-CoMP require negligible signaling overhead; much smaller than existing approaches that require detailed system knowledge (e.g., [15, 270]). Furthermore, UE-based deployment allows DRL agents to collect all relevant observations locally at the UEs without requiring cells to exchange information with each other directly, e.g., for scenarios with cells of different operators. While UE-based cell selection enables UEs to indicate which cells they want to connect to, the final connection decision can still be controlled by the network (cf. UEs' connection requests in 5G [187]).

10.6.3. Training and Inference in Practice

In practice, I suggest to train a reasonable policy in simulations up front and then copy and deploy the trained neural networks at the network or UE side. The pretrained neural networks can then be used for fast online inference with low resource requirements. The trained neural networks are lightweight (less than 5 MB in size) and can be further optimized using TensorFlow Lite for efficient inference with minimal space, compute, memory, and power consumption [49].

As I show in Section 10.7.3, it is important to train on randomized simulation scenarios for learning robust policies. To further support bridging the gap from simulation to real deployment ("sim2real gap" [125]), the proposed DRL approaches support online transfer learning, where they continuously fine-tune their pretrained policy in the actual deployment scenario. I explore and evaluate online transfer learning in Section 10.7.3. This online training could happen at edge computing centers close to the cells as proposed in 5G [243]. To facilitate online learning in practice, the reward signal for DeepCoMP, DD-CoMP, and D3-CoMP can be directly and locally derived from the collected observations without requiring additional information or overhead (Sections 10.4.1 and 10.5.2). During training, DD-CoMP agents periodically share their experiences and update their copy of the joint neural network, but D3-CoMP agents train locally without extra communication. To reduce communication overhead for DD-CoMP during training, agents could locally calculate their gradient updates and only share those using federated learning [267]. The computed gradients are smaller and therefore require less communication overhead than sharing the agents' full experiences, including observations, actions, and reward. Federated learning is practically feasible on mobile devices and already being used for commercial applications, e.g., Google Gboard [104]. During inference, no communication between DRL agents (or UEs) is necessary for either of the approaches.

10.7. Evaluation

I implemented prototypes of DeepCoMP, DD-CoMP, and D3-CoMP using Python 3.8, TensorFlow 2, and RLlib [153]. To encourage reproduction and reuse, the source code is publicly available [S13]. It also contains the light-weight simulation environment I used for training and evaluation, which others can use to study and improve their approaches, too.

10.7.1. Evaluation Setup

I evaluate DeepCoMP, DD-CoMP, and D3-CoMP in different scenarios, comparing their average QoE over $T = 100$ time steps (Section 10.3.3). In Section 10.7.2, I train the approaches from scratch in different mobile scenarios and evaluate how well they adapt to each scenario through self-learning without changing their hyperparameters or making any manual adjustments. In Section 10.7.3, I investigate how well the pretrained DRL approaches generalize to new scenarios without any further training and how continuous online transfer learning can boost performance and training efficiency. Finally, Section 10.7.4 compares the scalability of the different approaches.

Base Scenario

I vary a base scenario with three partially overlapping cells and three moving UEs, using typical LTE parameters [248] and the Okumura-Hata model [171] for path loss in urban areas, detailed in Table 10.2. Since I focus on downlink, there is no intra-cell interference. The approaches control cell selection but not resource allocation, e.g., scheduling of PRBs, which the cells coordinate transparently themselves using CoMP coordinated scheduling, avoiding inter-cell interference [168, 206].

Table 10.2.: Evaluation parameter settings

Parameter	Value
Carrier frequency	2.5 GHz
Bandwidth	9 MHz
Thermal noise density	-90 dBm/Hz
Path loss model	Okumura-Hata [171]
Cell transmit power	30 dBm
SINR connection threshold γ_{SINR}	-77 dB
Cell tower height	50 m
UE height	1.5 m
Cell-to-cell distance	100 m

UEs move according to the improved random waypoint model [269] with uniformly distributed velocity of 1 m/s–3 m/s and pause 2 time steps at waypoints. Each UE u_j has utility $U_j(t) = 10\log_{10}(r_j(t))$ with $U_j^{\min} = -10$ and $U_j^{\max} = 10$, quantifying its QoE [77, 134]. As evaluation metric, I consider the average QoE over all time steps and UEs as defined in Section 10.3.3, where positive values close to $U_j^{\max} = 10$ indicate excellent QoE, values around 0 indicate satisfactory QoE, and negative values close to $U_j^{\min} = -10$ indicate bad QoE. The chosen utility function and limits are examples and could also be chosen differently. In Section 10.7.2, I show that the DRL approaches also successfully adapt to other objectives and utility functions.

DRL Hyperparameters

To avoid time-intensive and resource-intensive hyperparameter tuning, I used the PPO default settings [153]:

- Fully connected neural networks for both actor and critic. Each neural network is configured with two hidden layers, each with 256 hidden units and tanh activation [126], and trained with the Adam optimizer [138].
- Discount factor $\gamma = 0.99$.
- Learning rate $\alpha = 5 \cdot 10^{-5}$.

- Mini-batch size $|b| = 4000$ with 30 training epochs, each using a subset of 128 experiences sampled from b.
- PPO clipping parameter of 0.3.
- Kullback-Leibler coefficient initialization of 0.2 and target of 0.01.
- Generalized advantage estimation parameter $\lambda = 1$.
- Value function parameter 1 and entropy coefficient 0.

These settings are largely in line with general recommendations for practical DRL [16]. Tuning hyperparameters automatically for each scenario could further improve performance.

Compared Algorithms

I compare DeepCoMP, DD-CoMP, and D3-CoMP against three other approaches:

- *Dynamic:* A dynamic UE-centric multi-cell selection heuristic by Beylerian and Ohtsuki [27], which connects a UE u_j to all cells c_i with $\text{SINR}_{i,j}(t) \geq \epsilon \cdot \text{SINR}_j^{\max}(t)$, where $\text{SINR}_j^{\max}(t)$ is the SINR of the strongest cell. I consider settings $\epsilon \in \{0, 0.5, 1\}$. With $\epsilon = 1$, the heuristic only connects to the strongest cell, similar to common 3GPP-based approaches for single-cell selection [281].
- *Static:* A static clustering approach similar to the approach by Marsch and Fettweis [169]. It groups all cells into fixed clusters of size C and then connects UEs to all cells of the strongest cluster. I consider $C \in \{1, 2, 3\}$.
- *Per-Step Optimal:* A brute-force approach that has full knowledge and checks all actions to select the best one in each time step. The selected action is optimal *per step* but not necessarily in the long term. This approach is clearly impractical and only serves as a comparison.

All approaches have the same action space, but only the proposed DRL approaches learn through feedback from their actions.

Execution & Figures

I repeated all experiments with 10 different seeds for training and testing, running on machines with an Intel Xeon W-2145 CPU and 100 GB RAM. Figures show the mean and 90 % confidence interval of the average QoE as defined in Section 10.3.3.

10.7.2. Self-Adaption to Varying Scenarios

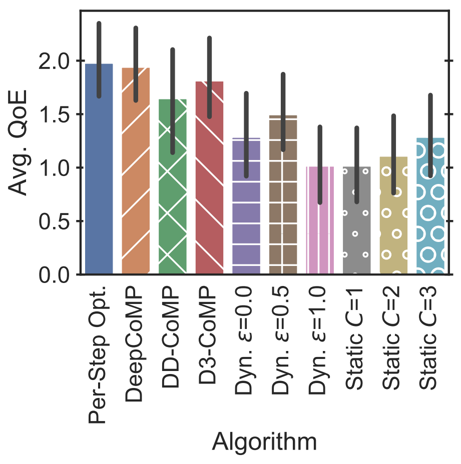
Varying Resource Allocation

I first consider the base scenario (Section 10.7.1) with varying resource allocation schemes applied by the cells to allocate their available PRBs to connected UEs. In particular, I consider scenarios with the following schemes:

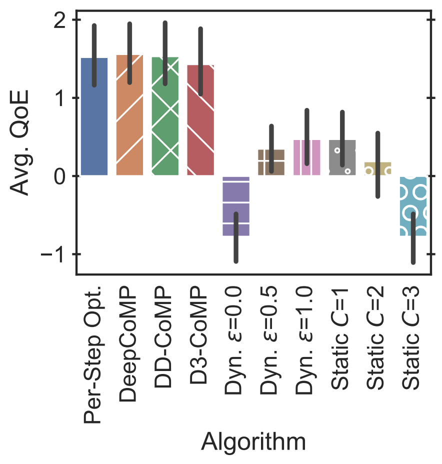
- Resource-fair: All cells allocate their available PRBs equally among connected UEs [224].
- Rate-fair: Each cell ensures the same downlink data rate for its connected UEs by assigning more PRBs to UEs that are farther away.

- Proportional-fair: Cells allocate PRBs in a proportional-fair way, based on UEs' instantaneous and historical data rates [266].

In addition, I also consider a *heterogeneous* scenario with all three schemes used in parallel, each by one of the three cells.



(a) Resource-fair

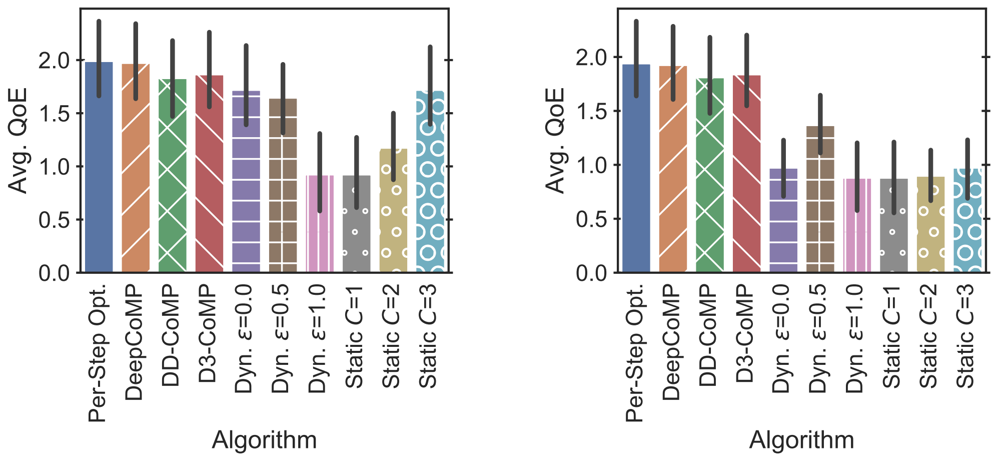


(b) Rate-fair

Figure 10.4.: The DRL approaches adapt to varying resource allocation schemes, without explicit knowledge of these schemes (here, resource-fair and rate-fair allocation; extended in Figure 10.5). Compared to other approaches, the proposed DRL approaches lead to significantly higher average QoE. (Figure from [S19].)

Figures 10.4 and 10.5 show the resulting average QoE under the different schemes. While the average QoE is mostly satisfactory (around 0 or above) in these scenarios, there are significant differences between the algorithms. DeepCoMP's results are almost identical (within 0.4 % on average) to the brute-force approach ("Per-Step Opt." in the figures) and much better than the heuristics. On average, DeepCoMP is 2x better than the dynamic heuristic (abbreviated as "Dyn." in the figures) and 2.3x better than the static heuristic over all parameter settings. In fact, DeepCoMP is even slightly better (2.5 %) than the brute-force approach for the rate-fair scheme (Figure 10.4b). Recall that brute force is only optimal *per step* but not necessarily long-term. Particularly, DeepCoMP learns to favor actions that may be suboptimal in short-term (e.g., connecting to a far-away cell) but pay off in long-term QoE. In contrast, the brute-force approach may get stuck at myopic optima since each UE is limited to at most one connection or disconnection per time step to limit overhead (Section 10.3.2). With rate-fair allocation, switching to a far-away cell is very resource-expensive but may pay off when reducing competition at a closer cell.

DD-CoMP and D3-CoMP only have local observations and control for each UE and are thus slightly worse than DeepCoMP but still much better than the dynamic and static heuristics. On average, DD-CoMP is 87 % better than the dynamic heuristic and 107 % better than the static heuristic. D3-CoMP is comparable to DD-CoMP but learns slightly better policies in some resource-fair scenarios where different UEs benefit from heterogeneous policies. Note that the performance of the two heuristics strongly varies with their parameter (ϵ and C) and



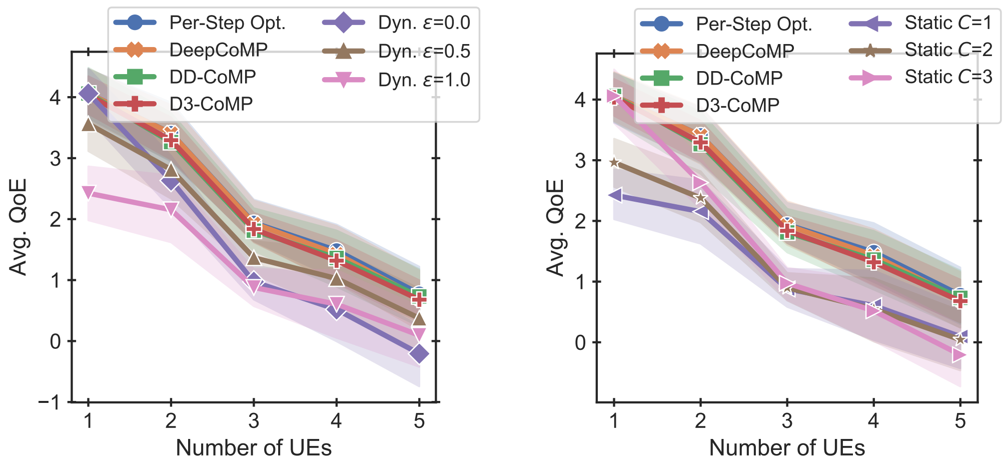
(a) Proportional-fair (b) Heterogeneous

Figure 10.5.: Average QoE under varying resource allocation schemes, extending Figure 10.4 (proportional-fair and heterogeneous). Again, the DRL approaches adapt to the different resource allocation schemes and even to heterogeneous schemes, outperforming all other approaches. (Figure from [S19].)

that the best parameter depends on the specific resource allocation scheme. For example, the dynamic heuristic works best with $\epsilon = 1$ in the rate-fair case, with $\epsilon = 0$ in the proportional-fair case, and with $\epsilon = 0.5$ in the other two cases. Similarly, the static heuristic works best with different parameter settings in different scenarios. Hence, for best results, these heuristics require knowledge of the resource allocation schemes and corresponding (possibly manual) configuration. In contrast, the three self-learning approaches autonomously and successfully adapt to the different resource allocation schemes and even the heterogeneous mix of schemes from training experience. I emphasize again that the agents have no explicit knowledge about which resource allocation scheme is employed inside the cells!

Varying Number of UEs

I again consider the base scenario (Section 10.7.1) with heterogeneous resource allocation and now vary the number of active UEs. Figure 10.6 shows the average QoE for five different scenarios with 1–5 UEs, respectively. Overall, average QoE decreases with more UEs as competition increases. With few UEs (1 or 2), there is little competition for radio resources. Here, the static heuristic with $C = 3$ works well as it connects UEs to many cells and uses available resources effectively. Conversely, selecting fewer cells ($C = 1$) leads to better results in scenarios with higher load (4 and 5 UEs) as it reduces competition among UEs by limiting a UE to a single cell. The dynamic heuristic works best with $\epsilon = 0.5$ across all scenarios (except for 1 UE) but is still clearly worse than the DRL approaches (e.g., DeepCoMP is 35 % better on average). The three proposed DRL approaches self-adapt to the number of UEs and learn cell selection policies that are very close to the per-step optimal brute-force approach. They consistently and considerably outperform both heuristics—in low-load as well as high-load scenarios and with any parameter setting.



(a) DRL vs. dynamic heuristic (b) DRL vs. static heuristic

Figure 10.6.: Average QoE for varying number of active UEs. The DRL approaches successfully adapt to scenarios with varying numbers of UEs and outperform all other approaches. (Figure from [S19].)

Varying Cell Density

I now vary the cell-to-cell distance (80 m–120 m) in the base scenario (three UEs, heterogeneous resource allocation), where cell size is generally small due to strong path loss in the considered urban scenario (Section 10.7.1). While the average QoE naturally decreases with sparser cells (from very good to satisfactory), Figure 10.7 shows that varying cell density affects the heuristics with different parameters differently: The static heuristic works best with $C = 1$ in denser cells (80 m), where a single serving cell can achieve good QoE, and with $C = 3$ in sparser cells (120 m), where multiple connections are required for cell-edge UEs. The dynamic heuristic is also affected by the cell density ($\epsilon = 0$ vs. $\epsilon = 1$) but works best with $\epsilon = 0.5$ across all scenarios and is comparable to DD-CoMP and D3-CoMP at 80 m. However, the DRL approaches adapt much better to other cell densities and overall outperform both the dynamic and the static heuristic by 6 %–73 % on average, depending on the parameter setting (ϵ and C).

Varying Objective and Utility Function

Now, I consider the base scenario (Section 10.7.1) with heterogeneous resource allocation and investigate the impact of using different utility functions. Instead of the logarithmic utility function, I first consider maximizing UEs' data rate directly rather than their QoE by setting $U_j(t) = r_j(t)$ with $U_j^{\min} = 0$ and $U_j^{\max} = 1000$. I choose $U_j^{\max} = 1000$, i.e., capping data rates above 1000 Mbit/s, as example supporting even upcoming highly demanding and immersive services [208]. Figure 10.8a shows UEs' resulting average data rate for each algorithm. Since UEs' data rates fluctuate heavily with their movement and distance to connected cells (due to strong path loss), the error bars are comparably large here. Still, the results clearly show that the proposed DRL approaches achieve similar average data rates as the per-step optimal

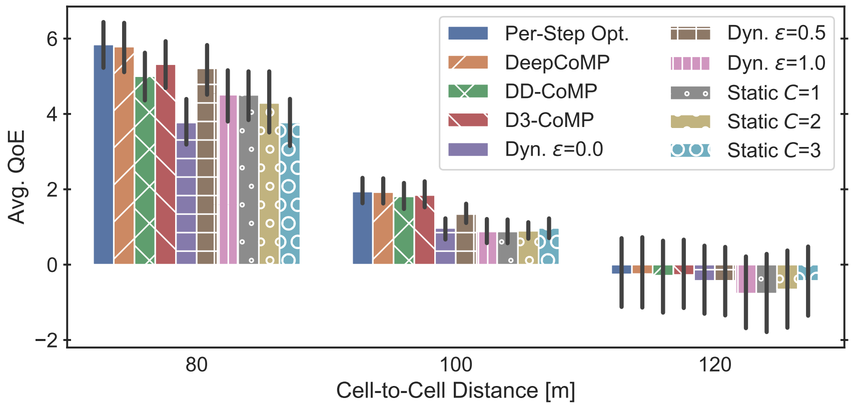
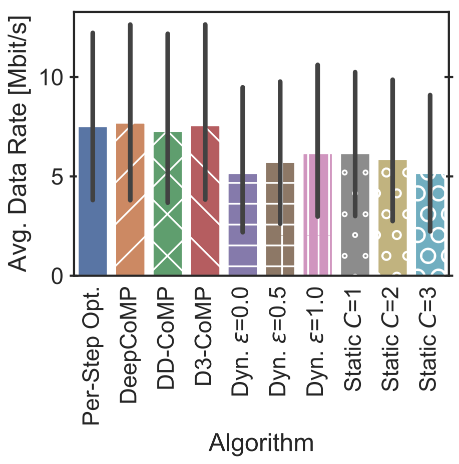


Figure 10.7.: DRL adapts to varying cell density. (Figure from [S19].)

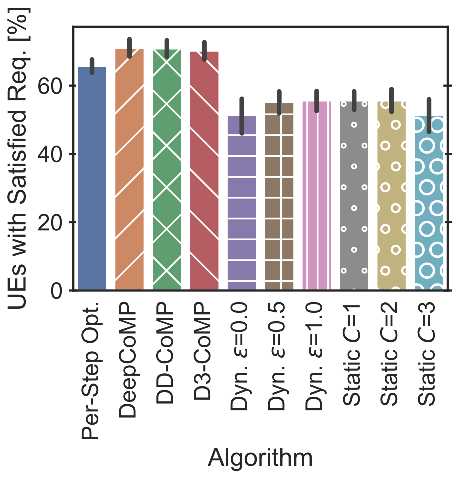
brute force approach and outperform all other algorithms (by 18 %–49 % on average, depending on ϵ and C). Particularly, they learn to only connect one UE per cell with the highest channel capacity. While this is unfair towards other UEs, e.g., at the cell edge, it maximizes the overall data rate and therefore the objective in this case. This illustrates why maximizing only data rate, as commonly done in related work, often does not reflect user satisfaction and easily leads to undesired behavior.

As another example, in Figure 10.8b, I consider the case that UEs have a hard data rate requirement (here 1 Mbit/s) that must be met. Hence, the utility follows a step function with $U_j(t) = -10$ if $r_j(t) < 1\,\text{Mbit/s}$ and $U_j(t) = 10$ otherwise. Figure 10.8b shows the resulting average percentage of UEs with satisfied data rate requirement, i.e., with $r_j(t) \geq 1\,\text{Mbit/s}$. The DRL approaches adapt to this scenario exceptionally well and outperform all other approaches, ensuring that 26 %–38 % more UEs reach their data rate requirements compared to the dynamic and static heuristic. They learn to select cells such that UEs' data rate is just high enough ($\geq 1\,\text{Mbit/s}$) and there are sufficient remaining radio resources to satisfy the data rate requirements of as many UEs as possible. Note that the DRL approaches are neither explicitly aware of UEs' required data rate nor the utility function but learn to adapt through feedback from UEs' instantaneous utility. The dynamic and static heuristic are also not aware of the required data rate but, unlike the DRL approaches, do not learn from experience. Instead, they tend to waste radio resources by either providing unnecessarily high data rates to some UEs or connecting cell edge users whose data rate still remains under the required threshold. Ultimately, this leads to many UEs with unfulfilled data rate requirements.

Figure 10.8b shows that the DRL approaches even outperform the brute force approach. Again, this is possible because the brute force approach is only optimal *per step* but not necessarily in the long term. Indeed, brute force leads to suboptimal cell selection in this setting where one cell uses proportional fair resource allocation, which depends on UEs' historic long-term data rate. While the DRL approaches are not explicitly aware of the resource allocation schemes at the different cells, they implicitly learn to select actions with possibly suboptimal instantaneous utility but that optimize the average QoE in the long term. Overall, the DRL approaches effectively self-adapt to different objectives and utility functions and outperform all other algorithms. Again, I used these utility functions merely as examples for evaluation. In practice, the approaches do not require defining a utility function at all and it suffices to approximate UEs' instantaneous QoE (e.g., with machine learning [173]).



(a) Maximizing data rate



(b) Hard data rate requirements

Figure 10.8.: DRL adapts to varying objectives and utility functions. (Figure from [S19].)

10.7.3. Generalization and Transfer Learning

Generalization

In Section 10.7.2, I train the DRL approaches from scratch in different scenarios and test how well they adapt to each scenario, using identical UE positions and movement during training and testing. Here, I consider pretrained DRL agents and evaluate their capability to generalize to unseen, randomized UE movement without further training. Note that the approaches are only indirectly aware of UEs' position and movement via observed SINR and QoE.

When trained on a single UE movement pattern, Figure 10.9 ("Fixed") shows that DeepCoMP and D3-CoMP generalize poorly to new UE movement patterns (bad average QoE indicated by negative values) while DD-CoMP generalizes much better. This is because DeepCoMP and D3-CoMP learn cell selection separately for each UE, whereas DD-CoMP combines UEs' experiences and learns a joint policy for all UEs. Based on the diverse experience from these UEs, the joint policy is naturally more robust to different UE locations and movement. Accordingly, Figure 10.9 ("Random") shows that training on randomly varying UE movement greatly improves generalization for DeepCoMP and D3-CoMP but also for DD-CoMP.

A challenge when training with randomized UE movement is the resulting high variance of experience collected during training, which is known to negatively affect performance of DRL approaches [97]. While randomized UE movement during training helps generalization, a simple option to reduce variance is to increase discounting of rewards by smaller γ (cf. Algorithm 8, line 13). This decreases the weight of subsequent rewards and, thus, also reduces impact of variance in later time steps. Figure 10.9 ("Discounted") shows that stronger discounting with

$\gamma = 0.5$ (instead of $\gamma = 0.99$) increases average QoE when training the DRL approaches on randomized UE movement, significantly outperforming the two heuristics by 40 %–143 % on average ("Baselines").

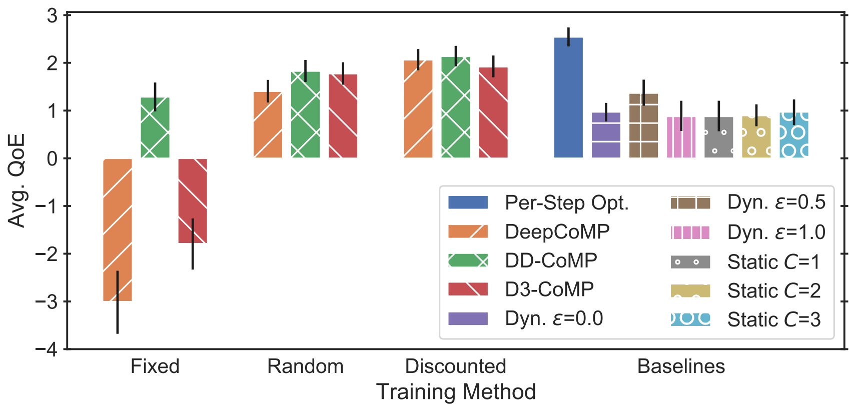


Figure 10.9.: Generalization to unseen UE movement. (Figure from [S19].)

Online Transfer Learning

As UE movement patterns may change over time, the pretrained DRL approaches can adapt online to the current pattern using transfer learning. In fact, fine-tuning a pretrained policy to new UE movement with transfer learning is much more efficient than training a new policy from scratch, i.e., starting with a randomly initialized neural network. For DeepCoMP, Figure 10.10 shows that such fine-tuning helps to quickly converge towards a very good policy (similar to the per-step optimal brute-force approach). Compared to training from scratch, it requires roughly half the amount of training steps until convergence—both for policies pretrained on fixed and randomized UE movement.

In practice, a safe and efficient approach would be to pretain a robust policy on randomized scenarios and then quickly and continuously fine-tune it for highly optimized QoE in the current scenario (comparable to "Fine-tuned Rand." in Figure 10.10). This approach quickly reaches and exceeds the quality of the best heuristic (dynamic with $\epsilon = 0.5$) within just 6 training iterations or 24 k training steps (batch size $|b| = 4000$; Section 10.7.1). With ongoing transfer learning, the approach exceeds the best heuristic by ultimately 13 %. DD-CoMP and D3-CoMP converge even faster than DeepCoMP as shown in following Section 10.7.4. For a production system, the efficiency of my prototype implementation could be further improved, e.g., through code optimizations or hyperparameter tuning, leading to even faster online transfer learning.

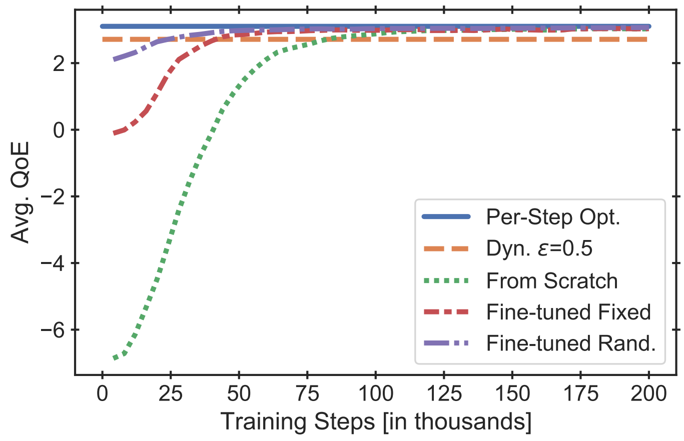


Figure 10.10.: The DRL approaches support online transfer learning for efficiently fine-tuning pretrained policies to new scenarios. (Figure adapted from [S19].)

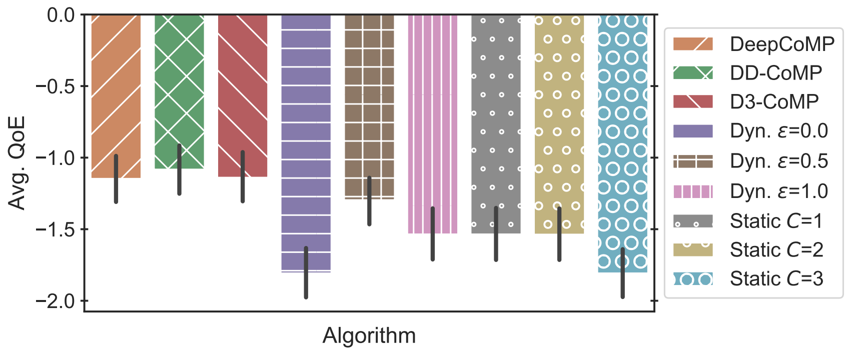


Figure 10.11.: DRL adapts to dynamic arrival of 15 UEs. (Figure from [S19].)

10.7.4. Scalability

Dynamic UE Arrival

I now explore a scenario similar to the base scenario (Section 10.7.1) but with up to 15 UEs that arrive dynamically over time. Due to the significantly higher number of UEs and the resulting combinatorial explosion, the brute-force approach becomes intractable and is therefore omitted. Overall, the average QoE is much lower than in previous scenarios due to the higher load and increased competition between UEs (Figure 10.11). With dynamically arriving UEs, DeepCoMP's observation and action spaces need to be configured for the maximum number of UEs, here $M^{\max} = 15$ (Section 10.4.1). In contrast, DD-CoMP and D3-CoMP have much smaller observation and action spaces that are invariant in the number of active UEs and can simply spawn new DRL agents whenever a new UE arrives (Sections 10.5.3 and 10.5.4). Consequently, DD-CoMP and D3-CoMP scale better to many UEs and converge much faster (investigated further in Section 10.7.4). Still, DeepCoMP also learns a good cell selection policy within the given training steps (here, 1 million). All three DRL approaches adapt successfully to the dynamically changing number of active UEs and outperform both heuristics with all parameter settings by 14 %–67 %.

Large Scenario

I further investigate the scalability of the proposed DRL approaches in a larger scenario with 7 cells and up to 20 moving UEs. Figure 10.12 shows the learning curves of the DRL approaches for 20 UEs, trained from scratch. DD-CoMP and D3-CoMP converge rapidly even in large scenarios with many UEs since their observation and action spaces are small and invariant in the number of UEs. In contrast, DeepCoMP's observation and action spaces grow linearly with more UEs, leading to higher complexity for larger scenarios, and indeed to much slower training convergence. At 1 million training steps, DeepCoMP is still significantly worse than DD-CoMP and D3-CoMP but keeps learning with more training and eventually catches up with DD-CoMP and D3-CoMP.

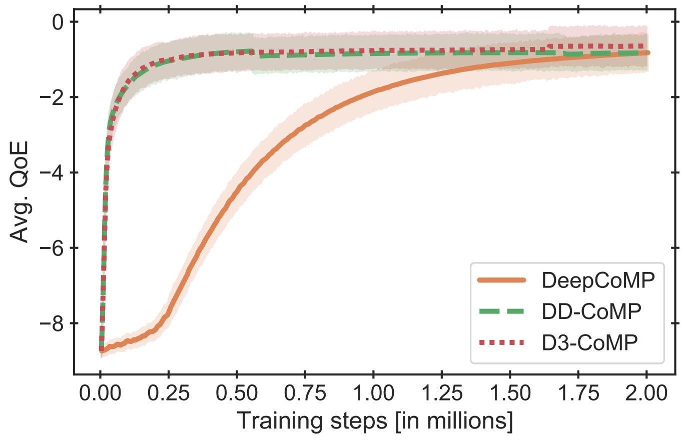


Figure 10.12.: DRL learning curves in a large scenario with 7 cells and 20 UEs. (Figure adapted from [S19].)

Figure 10.13 shows the final results of the DRL approaches trained for 2 million training steps, compared to the other algorithms. Again, average QoE decreases from rather good (> 0) to rather bad (< 0) with more UEs as competition increases. Due to the escalating complexity, the optimal brute-force approach is intractable and omitted for 10 or more UEs. Thanks to its global view and control of all UEs, DeepCoMP learns highly optimized cell selection and even exceeds DD-CoMP and D3-CoMP for 5–15 UEs. As shown in Figure 10.12 for 20 UEs, it reaches comparable performance within 2 million training steps but is not yet fully converged and would likely further improve with even more training. In contrast, DD-CoMP and D3-CoMP converge rapidly and still clearly outperform both heuristics with all parameter settings. For example, DD-CoMP is 69 % better on average than the best heuristic (Dyn. $\epsilon = 0.5$) at 20 UEs.

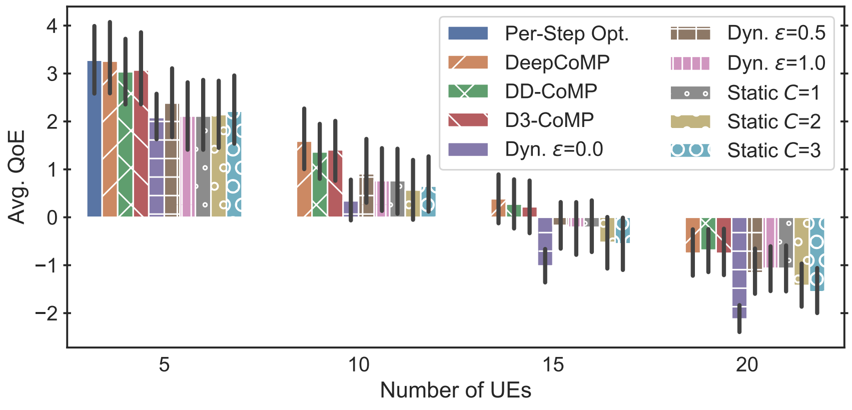


Figure 10.13.: Scalability to 7 cells and 5–20 UEs. (Figure from [S19].)

10.8. Conclusion

The three self-learning DRL approaches proposed in this chapter effectively self-adapt to various scenarios and objectives. They consistently and considerably outperform existing approaches without requiring detailed system knowledge or human instructions. DeepCoMP leverages its global view and control to learn highly optimized results and is useful for network-initiated cell selection when long training times are acceptable. Alternatively, DD-CoMP and D3-CoMP are suitable for either network-initiated or mobile-initiated cell selection, converge rapidly, and are particularly useful in practical large-scale scenarios. DD-CoMP learns a single, robust policy and D3-CoMP learns separate policies that can adapt to heterogeneous UEs. The source code is available online [S13] and can be used as a platform by others to study and evaluate their own solutions.

In future work, DeepCoMP, DD-CoMP, and D3-CoMP could be combined into a hybrid solution that dynamically switches to the most suitable approach. DD-CoMP and D3-CoMP could be further improved through recent advances in cooperative multi-agent DRL, e.g., curriculum learning [101]. Overall, I believe that the proposed DRL approaches are an important step towards self-adaptive, effective CoMP and higher QoE in practice. This chapter demonstrates that self-learning coordination techniques from wired networks (Chapters 8 and 9) can be adjusted and effectively applied to coordination in wireless mobile networks. Hence, I expect the proposed techniques to also carry over to other network control tasks, paving the way to zero-touch network and service management [25].

11. Future Research and Conclusion

This chapter discusses promising future work (Section 11.1) and finally concludes my dissertation (Section 11.2).

11.1. Future Research

While this PhD thesis covered a wide range of approaches for network and service coordination, there are many options for interesting future research building upon and complementing my contributions. My work focused on the underlying generic problem of network and service coordination (Chapter 3), which is relevant for a variety of real-world use cases. To use the proposed approaches in practice, they need to be tailored to the specific use case, integrating with available frameworks to obtain necessary inputs for the approaches and to apply coordination decisions. Chapter 2 discussed options for such integration, e.g., with Management and Orchestration (MANO) systems for coordination in Network Function Virtualization (NFV) scenarios.

Apart from these technical challenges, future research could focus on conceptual improvements or extensions of the proposed coordination approaches. For example, the hierarchical coordination approach (Chapter 5) repeatedly solves Mixed-Integer Linear Programs (MILPs) for the different domains on each hierarchical level. While this yields good solution quality, it becomes impractical in large, dynamic scenarios where quick decisions are required. Here, the proposed two-phase procedure could be combined with a fast heuristic algorithm instead of solving an MILP. This would maintain the benefit of hierarchical coordination, i.e., limited global knowledge and control as well as better scalability to large scenarios, but could speed up coordination significantly. Similarly, with recent progress in hierarchical Deep Reinforcement Learning (DRL) [197], hierarchical self-learning coordination approaches are an interesting research direction. Such approaches could strike a balance between centralized and distributed self-learning coordination approaches proposed in Chapters 8–10. Here, DRL agents on a lower hierarchical level could handle fast, time-critical coordination decisions locally (e.g., per-flow scheduling and routing). Medium-level agents could handle service scaling and placement on slightly longer time scales. Agents on higher hierarchical levels could focus on the bigger picture and continuously optimize network and service coordination in the long term by maintaining an overview and parametrizing and constraining lower-level agents. Alternatively, as discussed in Sections 6.6 and 10.8, the proposed centralized and distributed coordination approaches (conventional and/or learning-based) could be combined into a single hybrid approach. This approach could delegate time-critical coordination decisions to the fast, distributed coordination approaches (e.g., for small mice flows) and highly optimize other decisions with centralized coordination approaches (e.g., for elephant flows [42]).

To ensure meaningful progress in network and service coordination, the community needs more standardized procedures for evaluating and comparing proposed coordination approaches.

Similar to widely adopted benchmarks in machine learning (e.g., ImageNet [54] for computer vision or OpenAI Gym and Atari environments for reinforcement learning [32]), the networking community needs common and open datasets, simulation environments, baselines, and agreed-upon metrics for meaningful comparison. Consequently, all my coordination approaches as well as environments and datasets used for learning and evaluation are publicly available as open source to encourage reproduction and reuse [S25, S26, S11, S27, S28, S12, S17, S23, S14, S13, S29, S9]. Still, more effort is required to agree upon and adopt common processes for evaluation of both conventional and machine learning approaches.

Generally, I see self-learning coordination as an exciting and promising way forward for network and service coordination. As I showed in Chapters 8–10, these approaches self-adapt to varying scenarios and objectives without detailed knowledge or human instruction and can significantly outperform conventional coordination approaches. Nevertheless, there are a number of open challenges regarding self-learning coordination that must be solved before such approaches can be safely deployed in practice. Safe DRL is a field of active research and focuses on avoiding potentially harmful actions during online training and inference while still encouraging exploration [87, 223, 12, 41]. To avoid destructive coordination decisions (e.g., overloading available resources), safe DRL is also crucial for network and service coordination and should be considered in future research. Safe deployment of self-learning coordination approaches also requires bridging the gap from training on simulation to deployment on production infrastructure, i.e., addressing the "sim2real gap" [125, 111, 130]. Moreover, there are several other areas that are relevant for self-learning network and service coordination in practice: Leveraging available expert knowledge to accelerate learning [226, S38], few-shot transfer learning to quickly adapt to new scenarios [52], encoding and representation learning for more useful observations [53, 254], explainability of coordination decisions [205], and automated reward design and techniques for lifelong learning as coordination scenarios and goals evolve over time [7, 133, 81]. Since my work in this PhD thesis already showed the potential and benefits of self-learning coordination compared to conventional approaches, I suggest to next focus more on safe deployment of such self-learning coordination approaches than on further improving them. At the same time, explainability is important to build trust in these novel coordination approaches. Once self-learning coordination approaches are deployed successfully in practice, they can be improved iteratively (e.g., accelerating training, improving observations, automating reward design), taking insights from use in production into account. Overall, this PhD thesis significantly advances the state of the art in network and service coordination but also offers exciting opportunities for even more powerful and better applicable coordination approaches building upon my work.

11.2. Conclusion

My proposed approaches for network and service coordination each have different strengths and weaknesses and are therefore suitable for different scenarios: The conventional coordination approaches in Part I are useful for rather static and well-understood scenarios since they build on corresponding expert knowledge to coordinate network and services almost optimally. The centralized Bidirectional Scaling and Placement Coordination Approach (BSP) of Chapter 4 offers good coordination of few and long flows (Case I in Section 3.1.3) whereas the fully distributed heuristics of Chapter 6 focus on rapid coordination of many, short flows (Case II). The hierarchical coordination approach (Chapter 5) constitutes an intermediate solution, e.g., for medium to long flows in large networks.

To rely less on rigid rules and expert knowledge, a clear trend goes towards coordination approaches using machine learning (Part II). These approaches leverage available data to *learn* network and service coordination and can support existing conventional approaches (Chapter 7). While manually collecting data is cumbersome and expensive, data collection can often be automated, e.g., using automated service benchmarking or self-learning with DRL. DRL approaches can learn network and service coordination entirely by themselves, self-adapting to varying scenarios and objectives without detailed knowledge or human intervention (Chapters 8–10). Coordination approaches building on machine learning, particularly DRL, are therefore suitable for most practical scenarios, which are not perfectly understood and evolve over time. Again, there is a trade-off between centralized (highly optimized) vs. distributed (fast and scalable) coordination. This trade-off should be navigated depending on the considered problem size, pace of change, and available compute resources. If in doubt, I recommend using the distributed DRL approaches of Chapters 9 and 10 as sensible default approaches. These approaches incorporate my insights and lessons learned from all previous approaches and combine very good solution quality with fast, scalable, and self-adaptive coordination.

Overall, my PhD thesis presented approaches for automated and continuously optimized network and service coordination, significantly improving over existing work. Such improved coordination is critical to ensure great service quality with higher efficiency, leading to better user experience, lower costs, and reduced environmental impact. I therefore believe my contributions to be highly relevant for many real-world use cases and expect them to become even more important in the future!

Acknowledgments

First and foremost, I want to thank Prof. Dr. Holger Karl for his advice over all these years. I have truly enjoyed working with him and learning from him—be it through critical feedback for papers or through our numerous fruitful discussions. I also gratefully acknowledge Prof. Dr. Martina Zitterbart and Jun.-Prof. Dr. Sebastian Peitz for reviewing my dissertation.

Working in the computer networks group at Paderborn University has been great, thanks to the collaboration and interaction with my colleagues. Particularly, the joint work and discussions with Manuel, Sevil, Haitham, and Florian were fun and very helpful for my dissertation. I also want to thank Ramin and Artur at Huawei Munich with whom I collaborated closely and successfully over the last years. I really enjoyed our joint work and am proud of the outcomes. I am also very thankful for the fantastic opportunity the Software Campus program has offered me. Besides introducing the collaboration with Huawei Munich, the program's funding allowed me to lead my own research project (RealVNF) over two years. During these years, I grew significantly, gaining first-hand experience in leadership and project management, and built a network of friendly and useful connections. I am also thankful for the productive and pleasant collaboration with Sven, Adnan, Haydar, Rafael, and Stefan, who I recruited as student assistants for this and for later projects. Previously, I was involved in the 3-year European project 5GTANGO. Being my first larger project, I learned immensely both about collaboration in large projects and about good engineering practices. The project meetings all over Europe were certainly a highlight during my PhD. I am truly thankful for all these opportunities where I learned so much and where I was fortunate to meet and work with so many wonderful people. I hope we stay in touch!

Outside of work, I am deeply thankful for the constant support from my family. My parents motivated me to start a PhD in the first place and have been supporting me in all my decisions and all phases of life—thank you! Similarly, I want to thank my sister and grandparents for their constant support. Special thanks go to Elisa for her love, understanding, and support.

Lastly, I want to thank you, dear reader, for your interest and endurance, reading through the many pages of my dissertation. I hope you have found my research inspiring and I look forward to receiving feedback.

Acronyms

A3C Asynchronous Advantage Actor-Critic
ACKTR Actor-Critic using Kronecker-factored Trust Region
API Application Programming Interface
AR Augmented Reality
AWS Amazon Web Services
BFS Breadth-First Search
BNG Broadband Network Gateway
BSP Bidirectional Scaling and Placement Coordination Approach (Chapter 4)
CapEx Capital Expenditure
CQI Channel Quality Indicator
CoMP Coordinated Multipoint
CPU Central Processing Unit
DDPG Deep Deterministic Policy Gradient
DFS Depth-First Search
DPI Deep Packet Inspection
DRL Deep Reinforcement Learning
ETSI European Telecommunications Standards Institute
GCASP Greedy Coordination with Adaptive Shortest Paths (Section 6.4.1)
GCP Google Cloud Platform
GPS Global Positioning System
GPU Graphics Processing Unit
IDS Intrusion Detection System
IETF Internet Engineering Task Force
IRTF Internet Research Task Force
IoT Internet of Things
PRB Physical Resource Block
LB Load Balancing Coordination Approach (Baseline in Section 8.5.1)
LTE Long-Term Evolution
MANO Management and Orchestration
MDP Markov Decision Process
MEC Multi-Access Edge Computing
MILP Mixed-Integer Linear Program
MIMO Multiple Input and Multiple Output Radio
MLP Multi-Layer Perceptron
MMPP Markov-Modulated Poisson Process
MPLS Multiprotocol Label Switching
MSC Mobile-Services Switching Center
NFV Network Function Virtualization
NFVI NFV Infrastructure
NMRG Network Management Research Group
ONAP Open Network Automation Platform

OpEx	Operational Expenditure
OSM	Open Source MANO
OSPF	Open Shortest Path First
PCE	Path Computation Element
PGW	Packet Data Network Gateway
PPO	Proximal Policy Optimization
POMDP	Partially Observable Markov Decision Process
PRB	Physical Resource Block
QoE	Quality of Experience
QoS	Quality of Service
RAM	Random Access Memory
ReLU	Rectified Linear Unit
RMSE	Root Mean Squared Error
SBC	Score-Based Coordination (Section 6.4.2)
SDN	Software-Defined Networking
SFC	Service Function Chaining
SIB	System Information Broadcast
SINR	Signal-to-Interference-and-Noise Ratio
SLA	Service-Level Agreement
SP	Shortest Path Coordination Approach (Baseline in Sections 8.5.1 and 9.5.1)
SVR	Support Vector Regression
TCP	Transmission Control Protocol
TPU	Tensor Processing Unit
TRPO	Trust Region Policy Optimization
UE	User Equipment
VNE	Virtual Network Embedding
VNF	Virtual Network Function
VR	Virtual Reality
WAN	Wide Area Network

List of Algorithms

List of Figures

List of Tables

Bibliography: Own Work

[S1] Daniel Behnke, Marcel Müller, Patrick-Benjamin Bök, Stefan Schneider, Manuel Peuster, Holger Karl, Alberto Rocha, Miguel Mesquita, and José Bonnet. “NFV-Driven Intrusion Detection for Smart Manufacturing.” In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2019, pp. 1–6 (cit. on p. 5).

[S2] Sevil Dräxler, Stefan Schneider, and Holger Karl. “Scaling and Placing Bidirectional Services with Stateful Virtual and Physical Network Functions.” In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2018, pp. 123–131 (cit. on pp. 3, 7, 31, 32, 34, 39, 40, 61, 62, 80, 81, 98, 122).

[S3] Marcel Müller, Daniel Behnke, Patrick-Benjamin Bok, Manuel Peuster, Stefan Schneider, and Holger Karl. “5G as Key Technology for Networked Factories: Application of Vertical-specific Network Services for Enabling Flexible Smart Manufacturing.” In: *IEEE International Conference on Industrial Informatics (INDIN)*. IEEE. 2019, pp. 1495–1500 (cit. on p. 5).

[S4] Marcel Müller, Daniel Behnke, Patrick-Benjamin Bök, Stefan Schneider, Manuel Peuster, and Holger Karl. “Cloud-Native Threat Detection and Containment for Smart Manufacturing.” In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2020, pp. 347–349 (cit. on p. 5).

[S5] Marcel Müller, Daniel Behnke, Patrick-Benjamin Bök, Stefan Schneider, Manuel Peustery, and Holger Karl. “Putting NFV into Reality: Physical Smart Manufacturing Testbed.” In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2019, pp. 1–6 (cit. on p. 5).

[S6] Askhat Nuriddinov, Wouter Tavernier, Didier Colle, Mario Pickavet, Manuel Peustery, and Stefan Schneider. “Reproducible Functional Tests for Multi-scale Network Services.” In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2019, pp. 1–6 (cit. on p. 5).

[S7] Manuel Peuster, Stefan Schneider, Daniel Behnke, Marcel Müller, Patrick-Benjamin Bök, and Holger Karl. “Prototyping and Demonstrating 5G verticals: The Smart Manufacturing Case.” In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2019, pp. 236–238 (cit. on p. 5).

[S8] Manuel Peuster, Stefan Schneider, Frédéric Tobias Christ, and Holger Karl. “A Prototyping Platform to Validate and Verify Network Service Header-based Service Chains.” In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2018, pp. 1–5 (cit. on p. 4).

[S9] Manuel Peuster, Stefan Schneider, and Holger Karl. "The Softwarised Network Data Zoo." In: *IFIP/IEEE International Conference on Network and Service Management (CNSM)*. 🏆 **Best Poster Award**. IFIP/IEEE. 2019, pp. 1–5 (cit. on pp. 5, 26, 80, 85–87, 170).

[S10] Manuel Peuster, Stefan Schneider, Mengxuan Zhao, George Xilouris, Panagiotis Trakadas, Felipe Vicens, Wouter Tavernier, Thomas Soenen, Ricard Vilalta, George Andreou, et al. "Introducing Automated Verification and Validation for Virtualized Network Functions and Services." In: *IEEE Communications Magazine* 57.5 (2019), pp. 96–102 (cit. on p. 4).

[S11] Haydar Qarawlus and Stefan Schneider. *Distributed DRL Coordination GitHub Repository*. `https://github.com/RealVNF/distributed-drl-coordination`. 2020 (cit. on pp. 121, 122, 132, 170).

[S12] Stefan Schneider. *BSP GitHub Repository*. `https://github.com/CN-UPB/B-JointSP`. 2018 (cit. on pp. 31, 35, 70, 170).

[S13] Stefan Schneider. *DeepCoMP, DD-CoMP, and D3-CoMP GitHub Repository*. `https://github.com/CN-UPB/DeepCoMP`. 2021 (cit. on pp. 141, 143, 157, 168, 170).

[S14] Stefan Schneider. *Machine Learning for Dynamic Resource Allocation GitHub Repository*. `https://github.com/CN-UPB/ml-for-resource-allocation`. 2020 (cit. on pp. 79, 81, 86, 170).

[S15] Stefan Schneider. "Specifying, Scaling, Placing, and Reusing Bidirectional Forwarding Graphs of Virtual Network Functions." Master's Thesis. Paderborn University, Germany, 2017 (cit. on pp. 7, 31).

[S16] Stefan Schneider, Sevil Dräxler, and Holger Karl. "Trade-Offs in Dynamic Resource Allocation in Network Function Virtualization." In: *IEEE Global Communications Conference (GLOBECOM) Workshops*. IEEE. 2018, pp. 1–3 (cit. on pp. 4, 26, 41).

[S17] Stefan Schneider and Mirko Jürgens. *Hierarchical Coordination GitHub Repository*. `https://github.com/CN-UPB/hierarchical-coordination`. 2021 (cit. on pp. 39, 40, 57, 170).

[S18] Stefan Schneider, Mirko Jürgens, and Holger Karl. "Divide and Conquer: Hierarchical Network and Service Coordination." In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IFIP/IEEE. 2021 (cit. on pp. 3, 7, 17, 39, 45, 46, 52, 56, 58–60, 80).

[S19] Stefan Schneider, Holger Karl, Ramin Khalili, and Artur Hecker. "DeepCoMP: Coordinated Multipoint Using Multi-Agent Deep Reinforcement Learning." In: *IEEE Transactions on Network and Service Management (TNSM)* (2022). ℹ **Under Review** 👥 **Invited Talk at Ray Summit 2021** (cit. on pp. 4, 9, 141, 147, 153, 155, 160–168).

[S20] Stefan Schneider, Ramin Khalili, Adnan Manzoor, Haydar Qarawlus, Rafael Schellenberg, Holger Karl, and Artur Hecker. *RealVNF Project Website: Improved Coordination of Chained Virtual Network Functions Under Realistic Conditions*. `https://realvnf.github.io/`. 2021 (cit. on pp. 5, 8, 9, 97, 121).

[S21] Stefan Schneider, Ramin Khalili, Adnan Manzoor, Haydar Qarawlus, Rafael Schellenberg, Holger Karl, and Artur Hecker. "Self-Driving Network and Service Coordination Using Deep Reinforcement Learning." In: *IFIP/IEEE Conference on Network and Service Management (CNSM)*. 🏆 **Best Student Paper Award** 👥 **Invited Talk at the Internet Engineering Task Force (IETF) 110 Meeting**. IFIP/IEEE. 2020 (cit. on pp. 4, 8, 17, 97, 123).

[S22] Stefan Schneider, Ramin Khalili, Adnan Manzoor, Haydar Qarawlus, Rafael Schellenberg, Holger Karl, and Artur Hecker. "Self-Learning Multi-Objective Service Coordination Using Deep Reinforcement Learning." In: *IEEE Transactions on Network and Service Management (TNSM)* 18.3 (2021), pp. 3829–3842 (cit. on pp. 4, 8, 17, 97, 102, 103, 105, 107, 111–120).

[S23] Stefan Schneider and Lars Klenner. *Fully Distributed Network and Service Coordination GitHub Repository*. `https://github.com/CN-UPB/distributed-coordination`. 2020 (cit. on pp. 61, 62, 70, 170).

[S24] Stefan Schneider, Lars Dietrich Klenner, and Holger Karl. "Every Node for Itself: Fully Distributed Service Coordination." In: *IFIP/IEEE Conference on Network and Service Management (CNSM)*. IFIP/IEEE. 2020 (cit. on pp. 3, 7, 61, 67, 70–76, 80).

[S25] Stefan Schneider, Adnan Manzoor, Haydar Qarawlus, and Rafael Schellenberg. *DeepCoord GitHub Repository*. `https://github.com/RealVNF/DeepCoord`. 2021 (cit. on pp. 97–99, 108, 170).

[S26] Stefan Schneider, Adnan Manzoor, Haydar Qarawlus, Rafael Schellenberg, and Sven Uthe. *coord-sim GitHub Repository*. `https://github.com/RealVNF/coord-sim`. 2020 (cit. on pp. 97, 108, 132, 170).

[S27] Stefan Schneider, Adnan Manzoor, Haydar Qarawlus, and Sven Uthe. *Baseline Algorithms GitHub Repository*. `https://github.com/RealVNF/baseline-algorithms`. 2020 (cit. on p. 170).

[S28] Stefan Schneider, Adnan Manzoor, Haydar Qarawlus, and Sven Uthe. *BSP Adapter GitHub Repository*. `https://github.com/RealVNF/bjointsp-adapter`. 2020 (cit. on p. 170).

[S29] Stefan Schneider and Manuel Peuster. *Generic Placement Emulation Framework GitHub Repository*. `https://github.com/CN-UPB/placement-emulation`. 2018 (cit. on p. 170).

[S30] Stefan Schneider, Manuel Peuster, Daniel Behnke, Marcel Müller, Patrick-Benjamin Bök, and Holger Karl. "Putting 5G into Production: Realizing a Smart Manufacturing Vertical Scenario." In: *European Conference on Networks and Communications (EuCNC)*. IEEE. 2019, pp. 305–309 (cit. on pp. 5, 16).

[S31] Stefan Schneider, Manuel Peuster, Kai Hannemann, Daniel Behnke, Marcel Müller, Patrick-Benjamin Bök, and Holger Karl. ""Producing Cloud-Native": Smart Manufacturing Use Cases on Kubernetes." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2019, pp. 1–2 (cit. on pp. 5, 16).

[S32] Stefan Schneider, Manuel Peuster, and Holger Karl. "A Generic Emulation Framework for Reusing and Evaluating VNF Placement Algorithms." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2018, pp. 1–6 (cit. on p. 4).

[S33] Stefan Schneider, Manuel Peuster, Wouter Tavernier, and Holger Karl. "A Fully Integrated Multi-Platform NFV SDK." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2018, pp. 1–2 (cit. on pp. 4, 16, 17).

[S34] Stefan Schneider, Haydar Qarawlus, and Holger Karl. "Distributed Online Service Coordination Using Deep Reinforcement Learning." In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2021 (cit. on pp. 4, 9, 17, 24, 25, 121, 125, 127, 130, 132, 135–139).

[S35] Stefan Schneider, Narayanan Puthenpurayil Satheeschandran, Manuel Peuster, and Holger Karl. "Machine Learning for Dynamic Resource Allocation in Network Function Virtualization." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2020, pp. 122–130 (cit. on pp. 4, 8, 79, 83, 84, 87–94, 99).

[S36] Stefan Schneider, Arnab Sharma, Holger Karl, and Heike Wehrheim. "Specifying and Analyzing Virtual Network Services Using Queuing Petri Nets." In: *IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*. IFIP/IEEE. 2019, pp. 116–124 (cit. on pp. 4, 19).

[S37] Peter Twamley, Marcel Müller, Patrick-Benjamin Bök, George K. Xilouris, Christos Sakkas, Michail Alexandros Kourtis, Manuel Peuster, Stefan Schneider, Panagiotis Stavrianos, and Dimosthenis Kyriazis. "5GTANGO: An Approach for Testing NFV Deployments." In: *European Conference on Networks and Communications (EuCNC)*. 2018. DOI: `10.1109/EuCNC.2018.8442844` (cit. on p. 4).

[S38] Stefan Werner, Stefan Schneider, and Holger Karl. "Use What You Know: Network and Service Coordination Beyond Certainty." In: *IEEE/IFIP Network Operations and Management Symposium (NOMS)*. ℹ **Under Review**. IEEE/IFIP. 2022 (cit. on pp. 4, 170).

[S39] Anastasios Zafeiropoulos, Eleni Fotopoulou, Manuel Peuster, Stefan Schneider, Panagiotis Gouvas, Daniel Behnke, Marcel Müller, Patrick-Benjamin Bök, Panagiotis Trakadas, Panagiotis Karkazis, et al. "Benchmarking and Profiling 5G Verticals' Applications: An Industrial IoT Use Case." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2020, pp. 310–318 (cit. on p. 5).

[S40] Mengxuan Zhao et al. “Verification and Validation Framework for 5G Network Services and Apps.” In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2017, pp. 321–326 (cit. on p. 4).

Bibliography: Others' Work

[1] 3GPP. *LTE Release 11: 3GPP TR 36.819*. Tech. rep. Version 11.1.0. 3rd Generation Partnership Project (3GPP), 2012. URL: `https://www.3gpp.org/specifications/releases/69-release-11` (cit. on pp. 15, 141, 142).

[2] 3GPP. *Release 15: NR; Radio Resource Control (RRC); Protocol specification; TS 38.331*. Tech. rep. Version 15.13.0. 3rd Generation Partnership Project (3GPP), 2021 (cit. on p. 156).

[3] 5GTANGO project consortium. *5GTANGO Development and Validation Platform for Global Industry-specific Network Services and Apps*. `https://5gtango.eu` (accessed Jan 21, 2021). 2021 (cit. on pp. 5, 12).

[4] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. "Tensorflow: A system for large-scale machine learning." In: *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 265–283 (cit. on pp. 108, 132).

[5] Nasir Abbas, Yan Zhang, Amir Taherkordi, and Tor Skeie. "Mobile edge computing: A survey." In: *IEEE Internet of Things Journal* 5.1 (2017), pp. 450–465 (cit. on p. 14).

[6] Sherif Abdelwahab, Bechir Hamdaoui, Mohsen Guizani, and Taieb Znati. "Network function virtualization in 5G." In: *IEEE Communications Magazine* 54.4 (2016), pp. 84–91 (cit. on p. 1).

[7] David Abel, Yuu Jinnai, Sophie Yue Guo, George Konidaris, and Michael Littman. "Policy and value transfer in lifelong reinforcement learning." In: *International Conference on Machine Learning (ICML)*. PMLR. 2018, pp. 20–29 (cit. on p. 170).

[8] Satyam Agarwal, Francesco Malandrino, Carla-Fabiana Chiasserini, and Swades De. "Joint VNF placement and CPU allocation in 5G." In: *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. 2018, pp. 1943–1951 (cit. on p. 2).

[9] Shohreh Ahvar, Hnin Pann Phyu, Sachham Man Buddhacharya, Ehsan Ahvar, Noel Crespi, and Roch Glitho. "CCVP: Cost-efficient centrality-based VNF placement and chaining algorithm for network service provisioning." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2017, pp. 1–9 (cit. on pp. 40, 61, 62).

[10] Abdelkader Aissioui, Adlen Ksentini, Abdelhak Mourad Gueroui, and Tarik Taleb. "On enabling 5G automotive systems using follow me edge-cloud concept." In: *IEEE Transactions on Vehicular Technology* 67.6 (2018), pp. 5302–5316 (cit. on p. 15).

[11] Mohammad Abu Alsheikh, Dinh Thai Hoang, Dusit Niyato, Hwee-Pink Tan, and Shaowei Lin. "Markov decision processes with applications in wireless sensor networks: A survey." In: *IEEE Communications Surveys & Tutorials* 17.3 (2015), pp. 1239–1267 (cit. on p. 143).

[12] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. "Safe reinforcement learning via shielding." In: *AAAI Conference on Artificial Intelligence*. 2018 (cit. on p. 170).

[13] Amazon. *Amazon Web Services (AWS)*. `https://aws.amazon.com/` (Feb 11, 2021). 2021 (cit. on p. 14).

[14] Amazon Web Services (AWS). *AWS Docs (Traffic Dials)*. `https://docs.aws.amazon.com/global-accelerator/latest/dg/about-endpoint-groups-traffic-dial.html` (March 18, 2020). 2020 (cit. on p. 103).

[15] David Amzallag, Reuven Bar-Yehuda, Danny Raz, and Gabriel Scalosub. "Cell selection in 4G cellular networks." In: *IEEE Transactions on Mobile Computing* 12.7 (2013), pp. 1443–1455 (cit. on pp. 142, 144, 156).

[16] Marcin Andrychowicz, Anton Raichuk, Piotr Stańczyk, Manu Orsini, Sertan Girgin, Raphaël Marinier, Léonard Hussenot, Matthieu Geist, Olivier Pietquin, Marcin Michalski, et al. "What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study." In: *International Conference on Learning Representations (ICLR)*. 2021 (cit. on p. 159).

[17] Sylvain Arlot and Alain Celisse. "A survey of cross-validation procedures for model selection." In: *Statistics surveys* 4 (2010), pp. 40–79 (cit. on p. 88).

[18] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. "A view of cloud computing." In: *Communications of the ACM* 53.4 (2010), pp. 50–58 (cit. on p. 14).

[19] Jose A Ayala-Romero, Andres Garcia-Saavedra, Marco Gramaglia, Xavier Costa-Perez, Albert Banchs, and Juan J Alcaraz. "vrAIn: A deep learning approach tailoring computing and radio resources in virtualized RANs." In: *International Conference on Mobile Computing and Networking (MobiCom)*. 2019, pp. 1–16 (cit. on p. 144).

[20] Sara Ayoubi, Noura Limam, Mohammad A Salahuddin, Nashid Shahriar, Raouf Boutaba, Felipe Estrada-Solano, and Oscar M Caicedo. "Machine learning for cognitive network management." In: *IEEE Communications Magazine* 56.1 (2018), pp. 158–165 (cit. on p. 82).

[21] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. "Serverless computing: Current trends and open problems." In: *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20 (cit. on p. 19).

[22] Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, Raouf Boutaba, and Otto Carlos Muniz Bandeira Duarte. "Orchestrating Virtualized Network Functions." In: *IEEE Transactions on Network and Service Management (TNSM)* 13.4 (2016), pp. 725–739 (cit. on pp. 40, 61, 62).

[23] Md Faizul Bari, Shihabur Rahman Chowdhury, Reaz Ahmed, and Raouf Boutaba. "On Orchestrating Virtual Network Functions." In: *IEEE Conference on Network and Service Management (CNSM)*. IEEE. 2015, pp. 50–56 (cit. on p. 40).

[24] Michael Till Beck, Andreas Fischer, Hermann de Meer, Juan Felipe Botero, and Xavier Hesselbach. "A distributed, parallel, and generic virtual network embedding framework." In: *IEEE International Conference on Communications (ICC)*. IEEE. 2013, pp. 3471–3475 (cit. on p. 62).

[25] Chafika Benzaid and Tarik Taleb. "AI-driven zero touch network and service management in 5G and beyond: Challenges and research directions." In: *IEEE Network* 34.2 (2020), pp. 186–194 (cit. on p. 168).

[26] CJ. Bernardos, A. Rahman, JC. Zuniga, LM. Contreras, P. Aranda, and P. Lynch. *Network Virtualization Research Challenges*. RFC 8568. RFC Editor, 2019 (cit. on p. 11).

[27] Anthony Beylerian and Tomoaki Ohtsuki. "Multi-point fairness in resource allocation for C-RAN downlink CoMP transmission." In: *EURASIP Journal on Wireless Communications and Networking* 2016.1 (2016), pp. 1–10 (cit. on pp. 142, 144, 159).

[28] Deval Bhamare, Mohammed Samaka, Aiman Erbad, Raj Jain, Lav Gupta, and H Anthony Chan. "Optimal virtual network function placement in multi-cloud service function chaining architecture." In: *Computer Communications* 102 (2017), pp. 1–16 (cit. on pp. 40, 62, 97, 99, 122).

[29] Marcel Blöcher, Ramin Khalili, Lin Wang, and Patrick Eugster. "Letting off STEAM: Distributed Runtime Traffic Scheduling for Service Function Chaining." In: *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. 2020 (cit. on pp. 99, 123).

[30] André B Bondi. "Characteristics of Scalability and Their Impact on Performance." In: *ACM International Workshop on Software and Performance (WOSP)*. 2000, pp. 195–203 (cit. on pp. 6, 45).

[31] Leo Breiman. "Random forests." In: *Machine learning* 45.1 (2001), pp. 5–32 (cit. on p. 87).

[32] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. "OpenAI Gym." In: *arXiv preprint arXiv:1606.01540* (2016) (cit. on pp. 108, 132, 170).

[33] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. "API design for machine learning software: Experiences from the scikit-learn project." In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 2013, pp. 108–122 (cit. on p. 86).

[34] Canonical. *Linux containers*. (accessed Feb 10, 2021). 2021. URL: `https://linuxcontainers.org/` (cit. on p. 14).

[35] Pedro Casas, Alessandro D'Alconzo, Florian Wamser, Michael Seufert, Bruno Gardlo, Anika Schwind, Phuoc Tran-Gia, and Raimund Schatz. "Predicting QoE in cellular networks using machine learning and in-smartphone measurements." In: *International Conference on Quality of Multimedia Experience (QoMEX)*. IEEE. 2017, pp. 1–6 (cit. on p. 156).

[36] Mohit Chamania and Admela Jukan. "A Survey of Inter-Domain Peering and Provisioning Solutions for the Next Generation Optical Networks." In: *IEEE Communications Surveys & Tutorials* 11.1 (2009), pp. 33–51 (cit. on p. 42).

[37] Tzu-Wen Chang, Tung-Wei Kuo, and Ming-Jer Tsai. "Fair VNF Provisioning in NFV Clusters via Node Labeling." In: *IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2020, pp. 1–6 (cit. on pp. 80, 81).

[38] Hong Chen and Avi Mandelbaum. "Discrete flow networks: Bottleneck analysis and fluid approximations." In: *Mathematics of operations research* 16.2 (1991), pp. 408–446 (cit. on p. 20).

[39] Xianfu Chen, Honggang Zhang, Celimuge Wu, Shiwen Mao, Yusheng Ji, and Medhi Bennis. "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning." In: *IEEE Internet of Things Journal* 6.3 (2018), pp. 4005–4018 (cit. on pp. 142, 144).

[40] Yang Chen and Jie Wu. "Flow Scheduling of Service Chain Processing in a NFV-based Network." In: *IEEE Transactions on Network Science and Engineering* (2020) (cit. on p. 99).

[41] Richard Cheng, Gábor Orosz, Richard M Murray, and Joel W Burdick. "End-to-end safe reinforcement learning through barrier functions for safety-critical continuous control tasks." In: *AAAI Conference on Artificial Intelligence*. Vol. 33. 01. 2019, pp. 3387–3395 (cit. on p. 170).

[42] Anshuman Chhabra and Mariam Kiran. "Classifying elephant and mice flows in high-speed scientific networks." In: *International Workshop on Innovating the Network for Data-Intensive Science (INDIS)* (2017), pp. 1–8 (cit. on pp. 76, 169).

[43] Cloud Native Computing Foundation. *KubeEdge: An Open Platform to Enable Edge Computing*. `https://kubeedge.io/en/` (Feb 11, 2021). 2021 (cit. on p. 14).

[44] Cloud Native Computing Foundation. *Kubernetes Documentation: Kubernetes Scheduler*. (accessed Feb 10, 2021). 2021. URL: `https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/` (cit. on p. 15).

[45] Cloud Native Computing Foundation. *Kubernetes: Production-Grade Container Orchestration*. `https://kubernetes.io/` (Jan 31, 2020). 2021 (cit. on pp. 14, 19, 25, 108).

[46] Cloud Native Computing Foundation. *Prometheus: Monitoring System & Time Series Database*. `https://prometheus.io/` (accessed Sep 8, 2020). 2020 (cit. on p. 67).

[47] Rami Cohen, Liane Lewin-Eytan, Joseph Seffi Naor, and Danny Raz. "Near optimal placement of virtual network functions." In: *IEEE Conference on Computer Communications (INFOCOM)*. 2015, pp. 1346–1354 (cit. on p. 81).

[48] Ray Contributors. *Ray GitHub Repository*. `https://github.com/ray-project/ray`. 2021 (cit. on p. 6).

[49] TensorFlow Contributors. *TensorFlow Lite*. `https://www.tensorflow.org/lite`. 2021 (cit. on p. 157).

[50] Qimei Cui, Hui Wang, Pengxiang Hu, Xiaofeng Tao, Ping Zhang, Jyri Hamalainen, and Liang Xia. "Evolution of limited-feedback CoMP systems from 4G to 5G: CoMP features and limited-feedback approaches." In: *IEEE Vehicular Technology Magazine* 9.3 (2014), pp. 94–103 (cit. on p. 15).

[51] Richard Cziva, Christos Anagnostopoulos, and Dimitrios P Pezaros. "Dynamic, latency-optimal vNF placement at the network edge." In: *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. 2018, pp. 693–701 (cit. on p. 1).

[52] Christopher R Dance, Julien Perez, and Théo Cachet. "Conditioned Reinforcement Learning for Few-Shot Imitation." In: *International Conference on Machine Learning (ICML)*. PMLR. 2021, pp. 2376–2387 (cit. on p. 170).

[53] Tim De Bruin, Jens Kober, Karl Tuyls, and Robert Babuška. "Integrating state representation learning into deep reinforcement learning." In: *IEEE Robotics and Automation Letters* 3.3 (2018), pp. 1394–1401 (cit. on p. 170).

[54] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "Imagenet: A large-scale hierarchical image database." In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255 (cit. on p. 170).

[55] D. Dhody, Y. Lee, D. Ceccarelli, J. Shin, and D. King. *Hierarchical Stateful Path Computation Element (PCE)*. Internet Request for Comments RFC 8751. IETF, 2020 (cit. on p. 41).

[56] Mouhamad Dieye, Shohreh Ahvar, Jagruti Sahoo, Ehsan Ahvar, Roch Glitho, Halima Elbiaze, and Noel Crespi. "CPVNF: Cost-efficient proactive VNF placement and chaining for value-added services in content delivery networks." In: *IEEE Transactions on Network and Service Management* 15.2 (2018), pp. 774–786 (cit. on p. 81).

[57] Edsger W. Dijkstra. "A note on two problems in connexion with graphs." In: *Numerische Mathematik* 1.1 (1959), pp. 269–271 (cit. on pp. 65, 68).

[58] Richard Douville, Jean-louis Le Roux, Jean-louis Rougier, and Stefano Secci. "A Service Plane Over the PCE Architecture for Automatic Multidomain Connection-Oriented Services." In: *IEEE Communications Magazine* 46.6 (2008), pp. 94–102 (cit. on p. 41).

[59] Sevil Dräxler. "Scaling, Placement, and Routing for Pliable Virtualized Composed Services." PhD thesis. University of Paderborn, Germany, 2019 (cit. on pp. 31, 80, 81).

[60] Sevil Dräxler and Holger Karl. "SPRING: Scaling, Placement, and Routing of Heterogeneous Services with Flexible Structures." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2019, pp. 115–123 (cit. on pp. 42, 81).

[61] Sevil Dräxler, Holger Karl, and Zoltán Ádám Mann. "JASPER: Joint Optimization of Scaling, Placement, and Routing of Virtual Network Services." In: *IEEE Transactions on Network and Service Management (TNSM)* 15.3 (2018), pp. 946–960 (cit. on pp. 2, 24, 39, 41, 99).

[62] Harris Drucker, Chris JC Burges, Linda Kaufman, Alex Smola, Vladimir Vapnik, et al. "Support Vector Regression Machines." In: *Advances in Neural Information Processing Systems* 9 (1997), pp. 155–161 (cit. on pp. 86, 87).

[63] Gabriel Dulac-Arnold, Nir Levine, Daniel J Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. "Challenges of real-world reinforcement learning: definitions, benchmarks and analysis." In: *Machine Learning* (2021), pp. 1–50 (cit. on p. 143).

[64] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. "Challenges of real-world reinforcement learning." In: *International Conference on Machine Learning (ICML) Workshop on RL4RealLife*. 2019 (cit. on p. 98).

[65] Jack Edmonds and Richard M Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems." In: *Journal of the ACM (JACM)* 19.2 (1972), pp. 248–264 (cit. on p. 47).

[66] Medhat Elsayed, Kevin Shimotakahara, and Melike Erol-Kantarci. "Machine Learning-based Inter-Beam Inter-Cell Interference Mitigation in mmWave." In: *IEEE International Conference on Communications (ICC)*. IEEE. 2020, pp. 1–6 (cit. on pp. 144, 146).

[67] Kenneth Emancipator and Martin H Kroll. "A quantitative measure of nonlinearity." In: *Clinical chemistry* 39.5 (1993), pp. 766–772 (cit. on p. 81).

[68] Vincenzo Eramo, Emanuele Miucci, Mostafa Ammar, and Francesco Giacinto Lavacca. "An approach for service function chain routing and virtual function network instance migration in network function virtualization architectures." In: *IEEE/ACM Transactions on Networking* 25.4 (2017), pp. 2008–2025 (cit. on p. 81).

[69] Yasser Al-Eryani and Ekram Hossain. "The D-OMA method for massive multiple access in 6G: Performance, security, and challenges." In: *IEEE Vehicular Technology Magazine* 14.3 (2019), pp. 92–99 (cit. on pp. 15, 141).

[70] ETSI. *Network Function Virtualization: Introductory White Paper*. `https://portal.etsi.org/NFV/NFV_White_Paper.pdf` (accessed on Feb 10, 2021). 2012 (cit. on p. 11).

[71] ETSI GS NFV-IFA 011 V3.2.1. *Network Function Virtualization (NFV) Release 3; Management and Orchestration; VNF Descriptor and Packaging Specification*. 2019 (cit. on p. 80).

[72] ETSI NFV ISG. *Network Functions Virtualisation (NFV): Use Cases*. Group Specification ETSI GS NFV 001 V1.1.1. 2013 (cit. on p. 35).

[73] ETSI OSM. *Open Source MANO Documentation: SDN Controllers/WIMs as Managers of Software-Defined Network Platforms*. (accessed Feb 10, 2021). 2021. URL: `https://osm.etsi.org/wikipub/index.php/OSM_Scope_and_Functionality#SDN_Controllers.2FWIMs_as_managers_of_Software-Defined_Network_Platforms` (cit. on p. 13).

[74] ETSI OSM. *Open Source MANO: Open Source NFV Management and Orchestration (MANO) software stack aligned with ETSI NFV*. `https://osm.etsi.org` (accessed Feb 10, 2021). 2021 (cit. on p. 12).

[75] Xincai Fei, Fangming Liu, Hong Xu, and Hai Jin. "Adaptive VNF scaling and flow routing with proactive demand prediction." In: *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. 2018, pp. 486–494 (cit. on pp. 82, 99).

[76] Min Feng, Jianxin Liao, Sude Qing, Tonghong Li, and Jingyu Wang. "COVE: Co-operative virtual network embedding for network virtualization." In: *Journal of Network and Systems Management* 26.1 (2018), pp. 79–107 (cit. on p. 62).

[77] Markus Fiedler and Tobias Hoßfeld. "Quality of experience-related differential equations and provisioning-delivery hysteresis." In: *ITC Specialist Seminar on Multimedia Applications-Traffic, Performance and QoE*. IEICE. 2010 (cit. on pp. 142, 146, 158).

[78] Wladyslaw Findeisen, Frederic N Bailey, Mieczyslaw Brdys, Krzysztof Malinowski, Piotr Tatjewski, and Adam Wozniak. *Control and Coordination in Hierarchical Systems*. John Wiley & Sons, 1980 (cit. on p. 41).

[79] Andreas Fischer, Juan Felipe Botero, Michael Till Beck, Hermann De Meer, and Xavier Hesselbach. "Virtual network embedding: A survey." In: *IEEE Communications Surveys & Tutorials* 15.4 (2013), pp. 1888–1906 (cit. on pp. 41, 62, 63).

[80] Wolfgang Fischer and Kathleen Meier-Hellstern. "The Markov-modulated Poisson process (MMPP) cookbook." In: *Performance evaluation* 18.2 (1993), pp. 149–171 (cit. on pp. 112, 135).

[81] Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. "Reverse curriculum generation for reinforcement learning." In: *Conference on Robot Learning*. PMLR. 2017, pp. 482–495 (cit. on p. 170).

[82] Robert W. Floyd. "Algorithm 97: Shortest Path." In: *Communications of the ACM* 5.6 (1962), p. 345. ISSN: 0001-0782. DOI: `10.1145/367766.368168`. URL: `https://doi.org/10.1145/367766.368168` (cit. on pp. 32, 33, 104).

[83] Lester Randolph Ford and Delbert R Fulkerson. "Maximal Flow Through a Network." In: *Classic Papers in Combinatorics*. Springer, 2009, pp. 243–248 (cit. on p. 47).

[84] Jerome H Friedman. "Greedy function approximation: A gradient boosting machine." In: *Annals of Statistics* (2001), pp. 1189–1232 (cit. on p. 87).

[85] Carlo Fuerst, Stefan Schmid, Lalith Suresh, and Paolo Costa. "Kraken: Online and elastic resource reservations for multi-tenant datacenters." In: *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. 2016 (cit. on pp. 99, 122).

[86] Jokin Garay, Jon Matias, Juanjo Unzilla, and Eduardo Jacob. "Service description in the NFV revolution: Trends, challenges and a way forward." In: *IEEE Communications Magazine* 54.3 (2016), pp. 68–74 (cit. on p. 13).

[87] Javier Garcıa and Fernando Fernández. "A comprehensive survey on safe reinforcement learning." In: *Journal of Machine Learning Research* 16.1 (2015), pp. 1437–1480 (cit. on p. 170).

[88] Rung-Hung Gau. "Optimal traffic engineering and placement of virtual machines in SDNs with service chaining." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2017, pp. 1–9 (cit. on p. 81).

[89] Stefan Geissler, Stanislav Lange, Florian Wamser, Thomas Zinner, and Tobias Hoßfeld. "KOMon—Kernel-based online monitoring of VNF packet processing times." In: *IEEE International Conference on Networked Systems (NetSys)*. IEEE. 2019, pp. 1–8 (cit. on p. 85).

[90] Tay Ghazar and Nancy Samaan. "Hierarchical Approach for Efficient Virtual Network Embedding Based on Exact Subgraph Matching." In: *IEEE Global Telecommunications Conference (GLOBECOM)*. IEEE. 2011, pp. 1–6 (cit. on p. 41).

[91] Milad Ghaznavi, Aimal Khan, Nashid Shahriar, Khalid Alsubhi, Reaz Ahmed, and Raouf Boutaba. "Elastic virtual network function placement." In: *IEEE Conference on Cloud Networking (CloudNet)*. IEEE. 2015, pp. 255–260 (cit. on pp. 81, 98, 99, 122).

[92] Fabio Giust, Luca Cominardi, and Carlos J Bernardos. "Distributed mobility management for future 5G networks: Overview and analysis of existing approaches." In: *IEEE Communications Magazine* 53.1 (2015), pp. 142–149 (cit. on p. 143).

[93] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks." In: *Conference on Artificial Intelligence and Statistics (AISTATS)*. 2011, pp. 315–323 (cit. on p. 109).

[94] Fred Glover and Manuel Laguna. "Tabu search." In: *Handbook of Combinatorial Optimization*. Springer, 1998, pp. 2093–2229 (cit. on pp. 32, 34).

[95] Morteza Golkarifard, Carla Fabiana Chiasserini, Francesco Malandrino, and Ali Movaghar. "Dynamic VNF placement, resource allocation and traffic routing in 5G." In: *Computer Networks* 188 (2021), p. 107830 (cit. on p. 80).

[96] Google. *Google Cloud Platform (GCP)*. `https://cloud.google.com/` (Feb 11, 2021). 2021 (cit. on p. 14).

[97] Evan Greensmith, Peter L Bartlett, and Jonathan Baxter. "Variance reduction techniques for gradient estimates in reinforcement learning." In: *Journal of Machine Learning Research* 5.Nov (2004), pp. 1471–1530 (cit. on p. 164).

[98] Lin Gu, Deze Zeng, Wei Li, Song Guo, Albert Y Zomaya, and Hai Jin. "Intelligent VNF Orchestration and Flow Scheduling via Model-assisted Deep Reinforcement Learning." In: *IEEE Journal on Selected Areas in Communications* (2019) (cit. on pp. 100, 123).

[99] Lin Gu, Deze Zeng, Sheng Tao, Song Guo, Hai Jin, Albert Y Zomaya, and Weihua Zhuang. "Fairness-aware dynamic rate control and flow scheduling for network utility maximization in network service chain." In: *IEEE Journal on Selected Areas in Communications* 37.5 (2019), pp. 1059–1071 (cit. on p. 99).

[100] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017 (cit. on p. 108).

[101] Jayesh K Gupta, Maxim Egorov, and Mykel Kochenderfer. "Cooperative multi-agent control using deep reinforcement learning." In: *International Conference on Autonomous Agents and Multiagent Systems*. Springer. 2017, pp. 66–83 (cit. on pp. 143, 145, 168).

[102] J. Halpern and C. Pignataro. *Service Function Chaining (SFC) Architecture*. RFC 7665. RFC Editor, 2015. URL: `http://www.rfc-editor.org/info/rfc7665` (cit. on pp. 1, 11, 18).

[103] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee. "Network function virtualization: Challenges and opportunities for innovations." In: *IEEE Communications Magazine* 53.2 (2015), pp. 90–97. ISSN: 0163-6804. DOI: `10.1109/MCOM.2015.7045396` (cit. on pp. 1, 11, 19).

[104] Andrew Hard, Kanishka Rao, Rajiv Mathews, Swaroop Ramaswamy, Françoise Beaufays, Sean Augenstein, Hubert Eichner, Chloé Kiddon, and Daniel Ramage. "Federated learning for mobile keyboard prediction." In: *arXiv preprint arXiv:1811.03604* (2018) (cit. on p. 157).

[105] Christoph Hardegen, Benedikt Pfülb, Sebastian Rieger, Alexander Gepperth, and Sven Reißmann. "Flow-based throughput prediction using deep learning and real-world network traffic." In: *IFIP/IEEE International Conference on Network and Service Management (CNSM)*. IFIP/IEEE. 2019, pp. 1–9 (cit. on p. 99).

[106] Hassan Hawilo, Manar Jammal, and Abdallah Shami. "Network function virtualization-aware orchestrator for service function chaining placement in the cloud." In: *IEEE Journal on Selected Areas in Communications* 37.3 (2019), pp. 643–655 (cit. on p. 62).

[107] Robert Hecht-Nielsen. "Theory of the backpropagation neural network." In: *Neural networks for perception*. Elsevier, 1992, pp. 65–93 (cit. on pp. 108, 132).

[108] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. "Deep reinforcement learning that matters." In: *AAAI Conference on Artificial Intelligence*. 2018 (cit. on pp. 108, 132).

[109] Pablo Hernandez-Leal, Michael Kaisers, Tim Baarslag, and Enrique Munoz de Cote. "A survey of learning in multiagent environments: Dealing with non-stationarity." In: *arXiv preprint arXiv:1707.09183* (2017) (cit. on p. 151).

[110] Juliver Gil Herrera and Juan Felipe Botero. "Resource Allocation in NFV: A Comprehensive Survey." In: *IEEE Transactions on Network and Service Management* 13.3 (2016), pp. 518–532 (cit. on pp. 2, 11, 31, 97, 122, 132).

[111] Irina Higgins, Arka Pal, Andrei Rusu, Loic Matthey, Christopher Burgess, Alexander Pritzel, Matthew Botvinick, Charles Blundell, and Alexander Lerchner. "Darla: Improving zero-shot transfer in reinforcement learning." In: *International Conference on Machine Learning*. PMLR. 2017, pp. 1480–1490 (cit. on p. 170).

[112] Ashley Hill, Antonin Raffin, Maximilian Ernestus, Adam Gleave, Anssi Kanervisto, Rene Traore, Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, and Yuhuai Wu. *Stable Baselines GitHub Repository*. `https://github.com/hill-a/stable-baselines`. 2018 (cit. on pp. 132, 133).

[113] Arthur E Hoerl and Robert W Kennard. "Ridge regression: Biased estimation for nonorthogonal problems." In: *Technometrics* 12.1 (1970), pp. 55–67 (cit. on pp. 86, 87).

[114] Cheol-Ho Hong and Blesson Varghese. "Resource Management in Fog/Edge Computing: A Survey on Architectures, Infrastructure, and Algorithms." In: *ACM Comput. Surv.* 52.5 (Sept. 2019). ISSN: 0360-0300. DOI: `10.1145/3326066`. URL: `https://doi.org/10.1145/3326066` (cit. on pp. 1, 97, 122).

[115] Ines Houidi, Wajdi Louati, and Djamal Zeghlache. "A Distributed Virtual Network Mapping Algorithm." In: *IEEE International Conference on Communications (ICC)*. IEEE. 2008, pp. 5634–5640 (cit. on pp. 41, 62).

[116] Marcus Hutter. "General discounting versus average reward." In: *International Conference on Algorithmic Learning Theory*. Springer. 2006, pp. 244–258 (cit. on p. 150).

[117] IBM. *Iceland is the coolest location for data centers*. (accessed Feb 10, 2021). 2021. URL: `https://www.ibm.com/blogs/nordic-msp/iceland-data-centers/` (cit. on p. 14).

[118] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." In: ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, 2015, pp. 448–456. URL: `http://proceedings.mlr.press/v37/ioffe15.html` (cit. on pp. 126, 148).

[119] Ralf Irmer, Heinz Droste, Patrick Marsch, Michael Grieger, Gerhard Fettweis, Stefan Brueck, Hans-Peter Mayer, Lars Thiele, and Volker Jungnickel. "Coordinated multipoint: Concepts, performance, and field trial results." In: *IEEE Communications Magazine* 49.2 (2011), pp. 102–111 (cit. on pp. 15, 145).

[120] ITU-T. *Architectural framework for machine learning in future networks including IMT-2020 (Y.3172)*. Recommendation. ITU-T, 2019 (cit. on pp. 1, 2, 19).

[121] Anil K Jain. "Data Clustering: 50 Years Beyond K-means." In: *Pattern Recognition Letters* 31.8 (2010), pp. 651–666 (cit. on p. 57).

[122] Ramesh Jain. "Quality of Experience." In: *IEEE MultiMedia* 11.1 (2004), pp. 96–95 (cit. on p. 146).

[123] Jose M Jerez, Ignacio Molina, Pedro J Garcia-Laencina, Emilio Alba, Nuria Ribelles, Miguel Martin, and Leonardo Franco. "Missing data imputation using statistical and machine learning methods in a real breast cancer problem." In: *Artificial intelligence in medicine* 50.2 (2010), pp. 105–115 (cit. on p. 85).

[124] Mirko Jürgens. "Hierarchical Coordination of Virtual Network Function Chains." Master's Thesis. Paderborn University, Germany, 2020 (cit. on pp. 7, 39).

[125] Abhishek Kadian, Joanne Truong, Aaron Gokaslan, Alexander Clegg, Erik Wijmans, Stefan Lee, Manolis Savva, Sonia Chernova, and Dhruv Batra. "Sim2Real predictivity: Does evaluation in simulation predict real-world performance?" In: *IEEE Robotics and Automation Letters* 5.4 (2020), pp. 6670–6677 (cit. on pp. 157, 170).

[126] Barry L Kalman and Stan C Kwasny. "Why tanh: Choosing a sigmoidal function." In: *International Joint Conference on Neural Networks (IJCNN*. Vol. 4. IEEE. 1992, pp. 578–581 (cit. on pp. 133, 158).

[127] Ioannis Kanellopoulos and Graeme G Wilkinson. "Strategies and best practice for neural network image classification." In: *International Journal of Remote Sensing* 18.4 (1997), pp. 711–725 (cit. on p. 88).

[128] Hyoung Seok Kang, Ju Yeon Lee, SangSu Choi, Hyun Kim, Jun Hee Park, Ji Yeon Son, Bo Hyun Kim, and Sang Do Noh. "Smart manufacturing: Past research, present findings, and future directions." In: *International Journal of Precision Engineering and Manufacturing-Green Technology* 3.1 (2016), pp. 111–128 (cit. on p. 1).

[129] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. "Optimizing the "One Big Switch" Abstraction in Software-Defined Networks." In: *ACM Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*. ACM, 2013, pp. 13–24 (cit. on p. 40).

[130] Manuel Kaspar, Juan D Muñoz Osorio, and Jürgen Bock. "Sim2real transfer for reinforcement learning without dynamics randomization." In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2020, pp. 4383–4388 (cit. on p. 170).

[131] Matthias Keller, Christoph Robbert, and Holger Karl. "Template embedding: Using application architecture to allocate resources in distributed clouds." In: *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE. 2014, pp. 387–395 (cit. on pp. 2, 24).

[132] Michel Gokan Khan, Saeed Bastani, Javid Taheri, Andreas Kassler, and Shuiguang Deng. "NFV-inspector: A systematic approach to profile and analyze virtual network functions." In: *IEEE International Conference on Cloud Networking (CloudNet)*. IEEE. 2018, pp. 1–7 (cit. on pp. 80, 81, 85).

[133] Khimya Khetarpal, Shagun Sodhani, Sarath Chandar, and Doina Precup. "Environments for lifelong reinforcement learning." In: *arXiv preprint arXiv:1811.10732* (2018) (cit. on p. 170).

[134] Stas Khirman and Peter Henriksen. "Relationship between quality-of-service and quality-of-experience for public Internet service." In: *Workshop on Passive and Active Measurement*. 2002 (cit. on pp. 142, 146, 158).

[135] Hoon Kim and Youngnam Han. "A proportional fair scheduling for multicarrier transmission systems." In: *IEEE Communications Letters* 9.3 (2005), pp. 210–212 (cit. on pp. 143, 146).

[136] Hyojoon Kim and Nick Feamster. "Improving network management with software defined networking." In: *IEEE Communications Magazine* 51.2 (2013), pp. 114–119 (cit. on p. 12).

[137] D. King and A. Farrel. *The Application of the Path Computation Element Architecture to the Determination of a Sequence of Domains in MPLS and GMPLS*. Internet Request for Comments RFC 6805. IETF, 2012 (cit. on p. 41).

[138] Diederik P Kingma and Jimmy Ba. "Adam: A method for stochastic optimization." In: *International Conference for Learning Representations*. 2015 (cit. on pp. 109, 158).

[139] Nahida Kiran, Xuanlin Liu, Sihua Wang, and Changchuan Yin. "VNF Placement and Resource Allocation in SDN/NFV-Enabled MEC Networks." In: *IEEE Wireless Communications and Networking Conference Workshops (WCNCW)*. IEEE. 2020, pp. 1–6 (cit. on p. 62).

[140] Jon M Kleinberg. "Single-Source Unsplittable Flow." In: *IEEE Conference on Foundations of Computer Science*. IEEE. 1996, pp. 68–77 (cit. on p. 42).

[141] Lars Dietrich Klenner. "Fully Distributed Scaling, Placement and Routing in Network Function Virtualization." Bachelor's Thesis. Paderborn University, Germany, 2020 (cit. on pp. 7, 61).

[142] 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 (2011), pp. 1765–1775. ISSN: 0733-8716. DOI: 10.1109/JSAC.2011.111002 (cit. on pp. 17, 19, 69, 75, 90, 109, 118, 122, 133, 138).

[143] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. "Software-defined networking: A comprehensive survey." In: *Proceedings of the IEEE* 103.1 (2014), pp. 14–76 (cit. on pp. 11, 12).

[144] Tung-Wei Kuo, Bang-Heng Liou, Kate Ching-Ju Lin, and Ming-Jer Tsai. "Deploying chains of virtual network functions: On the relation between link and server usage." In: *IEEE/ACM Transactions on Networking* 26.4 (2018), pp. 1562–1576 (cit. on pp. 40, 61, 62, 97, 99, 122).

[145] Stanislav Lange, Steffen Gebert, Thomas Zinner, Phuoc Tran-Gia, David Hock, Michael Jarschel, and Marco Hoffmann. "Heuristic approaches to the controller placement problem in large scale SDN networks." In: *IEEE Transactions on Network and Service Management* 12.1 (2015), pp. 4–17 (cit. on p. 108).

[146] Ying Loong Lee, Teong Chee Chuah, Jonathan Loo, and Alexey Vinel. "Recent advances in radio resource management for heterogeneous LTE/LTE-A networks." In: *IEEE Communications Surveys & Tutorials* 16.4 (2014), pp. 2142–2180 (cit. on p. 146).

[147] T. Lei, Y. Hsu, I. Wang, and C. Wen. "Deploying QoS-assured Service Function Chains with Stochastic Prediction Models on VNF Latency." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. 2017 (cit. on p. 82).

[148] Khaled B Letaief, Wei Chen, Yuanming Shi, Jun Zhang, and Ying-Jun Angela Zhang. "The roadmap to 6G: AI empowered wireless networks." In: *IEEE Communications Magazine* 57.8 (2019), pp. 84–90 (cit. on p. 2).

[149] Defang Li, Peilin Hong, Kaiping Xue, and Jianing Pei. "Virtual network function placement and resource optimization in NFV and edge computing enabled networks." In: *Computer Networks* 152 (2019), pp. 12–24 (cit. on p. 62).

[150] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. "Service Mesh: Challenges, state of the art, and future research opportunities." In: *IEEE Conference on Service-Oriented System Engineering (SOSE)*. IEEE. 2019, pp. 122–1225 (cit. on pp. 1, 14, 19).

[151] Xiaomin Li, Jiafu Wan, Hong-Ning Dai, Muhammad Imran, Min Xia, and Antonio Celesti. "A hybrid computing solution and resource scheduling strategy for edge computing in smart manufacturing." In: *IEEE Transactions on Industrial Informatics* 15.7 (2019), pp. 4225–4234 (cit. on p. 1).

[152] Y. Li and M. Chen. "Software-Defined Network Function Virtualization: A Survey." In: *IEEE Access* 3 (2015), pp. 2542–2553 (cit. on pp. 11, 12).

[153] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. "RLlib: Abstractions for distributed reinforcement learning." In: *International Conference on Machine Learning*. PMLR. 2018, pp. 3053–3062 (cit. on pp. 6, 157, 158).

[154] Zhe Liang, Yi Bing Li, and Fang Ye. "Cell Selection Algorithm based on Coordinated Beamforming." In: *Advanced Materials Research*. Vol. 989. 2014, pp. 1671–1675 (cit. on p. 143).

[155] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. "Continuous control with deep reinforcement learning." In: *International Conference on Learning Representations (ICLR)*. 2016 (cit. on pp. 98, 100, 106, 107, 131).

[156] Linux Foundation. *ONAP: Open Network Automation Platform*. `https://www.onap.org` (accessed Feb 10, 2021). 2021 (cit. on p. 12).

[157] Linux Foundation. *OpenDaylight SDN Controller*. `https://www.opendaylight.org/` (accessed Feb 10, 2021). 2021 (cit. on p. 13).

[158] Fang Liu, Jin Tong, Jian Mao, Robert Bohn, John Messina, Lee Badger, and Dawn Leaf. "NIST cloud computing reference architecture." In: *NIST special publication* 500.2011 (2011), pp. 1–28 (cit. on p. 13).

[159] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. "Serverless computing: An investigation of factors influencing microservice performance." In: *IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2018, pp. 159–169 (cit. on p. 1).

[160] Marcelo Caggiani Luizelli, Leonardo Richter Bays, Luciana Salete Buriol, Marinho Pilla Barcellos, and Luciano Paschoal Gaspary. "Piecing together the NFV provisioning puzzle: Efficient placement and chaining of virtual network functions." In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE. 2015, pp. 98–106 (cit. on p. 81).

[161] Marcelo Caggiani Luizelli, Weverton Luis da Costa Cordeiro, Luciana S Buriol, and Luciano Paschoal Gaspary. "A Fix-and-Optimize Approach for Efficient and Large Scale Virtual Network Function Placement and Chaining." In: *Computer Communications* 102 (2017), pp. 67–77 (cit. on pp. 40, 62).

[162] Nguyen Cong Luong, Dinh Thai Hoang, Shimin Gong, Dusit Niyato, Ping Wang, Ying-Chang Liang, and Dong In Kim. "Applications of deep reinforcement learning in communications and networking: A survey." In: *IEEE Communications Surveys & Tutorials* 21.4 (2019), pp. 3133–3174 (cit. on pp. 2, 100, 144).

[163] WANG Luping, WANG Wei, and LI Bo. "CMFL: Mitigating communication overhead for federated learning." In: *International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2019, pp. 954–964 (cit. on p. 130).

[164] Wenrui Ma, Oscar Sandoval, Jonathan Beltran, Deng Pan, and Niki Pissinou. "Traffic aware placement of interdependent NFV middleboxes." In: *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. 2017, pp. 1–9 (cit. on p. 20).

[165] Muthucumaru Maheswaran and Trung Vuong Thien. *Gini5 Documentation*. `https://citelab.github.io/gini5/features/service-function-chaining/` (Feb 10, 2021). 2021 (cit. on p. 13).

[166] Zoltán Ádám Mann. "Allocation of virtual machines in cloud data centers—a survey of problem models and optimization algorithms." In: *ACM Computing Surveys (CSUR)* 48.1 (2015), pp. 1–34 (cit. on p. 122).

[167] Yuyi Mao, Jun Zhang, SH Song, and Khaled B Letaief. "Stochastic joint radio and computational resource management for multi-user mobile-edge computing systems." In: *IEEE Transactions on Wireless Communications* 16.9 (2017), pp. 5994–6009 (cit. on p. 143).

[168] Andrea Marotta, Dajana Cassioli, Cristian Antonelli, Koteswararao Kondepu, and Luca Valcarenghi. "Network solutions for CoMP coordinated scheduling." In: *IEEE Access* 7 (2019), pp. 176624–176633 (cit. on pp. 145, 158).

[169] Patrick Marsch and Gerhard Fettweis. "Static clustering for cooperative multipoint (CoMP) in mobile communications." In: *IEEE International Conference on Communications (ICC)*. IEEE. 2011, pp. 1–6 (cit. on pp. 142, 144, 159).

[170] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. "OpenFlow: Enabling innovation in campus networks." In: *ACM SIGCOMM computer communication review* 38.2 (2008), pp. 69–74 (cit. on p. 11).

[171] Arturas Medeisis and Algimantas Kajackas. "On the use of the universal Okumura-Hata propagation prediction model in rural areas." In: *Vehicular Technology Conference (VTC)*. Vol. 3. IEEE. 2000, pp. 1815–1818 (cit. on p. 158).

[172] Peter Mell, Tim Grance, et al. "The NIST definition of cloud computing." In: (2011) (cit. on p. 13).

[173] Vlado Menkovski, Georgios Exarchakos, and Antonio Liotta. "Online QoE prediction." In: *International Workshop on Quality of Multimedia Experience (QoMEX)*. IEEE. 2010, pp. 118–123 (cit. on pp. 147, 148, 152, 156, 164).

[174] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." In: *Linux J.* 2014.239 (2014). ISSN: 1075-3583. URL: `http://dl.acm.org/citation.cfm?id=2600239.2600241` (cit. on p. 14).

[175] M. D. Mesarovic, D. Macko, and Y. Takahara. *Theory of Hierarchical, Multilevel, Systems, Volume 68*. Elsevier Science, 2000 (cit. on p. 41).

[176] Microsoft. *Microsoft Azure*. `https://azure.microsoft.com/en-us/` (Feb 11, 2021). 2021 (cit. on p. 14).

[177] Rashid Mijumbi, Sidhant Hasija, Steven Davy, Alan Davy, Brendan Jennings, and Raouf Boutaba. "Topology-aware prediction of virtual network function resource requirements." In: *IEEE Transactions on Network and Service Management* 14.1 (2017), pp. 106–120 (cit. on p. 82).

[178] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Raouf Boutaba. "Network function virtualization: State-of-the-art and research challenges." In: *IEEE Communications surveys & tutorials* 18.1 (2015), pp. 236–262 (cit. on pp. 1, 11).

[179] Rashid Mijumbi, Joan Serrat, Juan-Luis Gorricho, Niels Bouten, Filip De Turck, and Steven Davy. "Design and evaluation of algorithms for mapping and scheduling of virtual network functions." In: *Conference on Network Softwarization (NetSoft)*. IEEE. 2015, pp. 1–9 (cit. on pp. 97, 99, 122).

[180] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. "Asynchronous methods for deep reinforcement learning." In: *International Conference on Machine Learning (ICML)*. 2016, pp. 1928–1937 (cit. on p. 131).

[181] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. "Human-level control through deep reinforcement learning." In: *nature* 518.7540 (2015), pp. 529–533 (cit. on p. 106).

[182] Hendrik Moens and Filip De Turck. "VNF-P: A model for efficient placement of virtualized network functions." In: *International Conference on Network and Service Management (CNSM)*. IEEE. 2014, pp. 418–423 (cit. on pp. 40, 62, 97–99, 122).

[183] Somayeh Mosleh, Lingjia Liu, and Jianzhong Zhang. "Proportional-fair resource allocation for coordinated multi-point transmission in LTE-advanced." In: *IEEE Transactions on Wireless Communications* 15.8 (2016), pp. 5355–5367 (cit. on p. 143).

[184] John Moy. *OSPF version 2*. Internet Request for Comments RFC 2328. IETF, 1998 (cit. on p. 63).

[185] Priyanka Naik, Dilip Kumar Shaw, and Mythili Vutukuru. "NFVPerf: Online Performance Monitoring and Bottleneck Detection for NFV." In: *Network Function Virtualization and Software Defined Network (NFV-SDN), 2016 IEEE Conference on*. IEEE. 2016 (cit. on p. 85).

[186] Yasar Sinan Nasir and Dongning Guo. "Multi-agent deep reinforcement learning for dynamic power allocation in wireless networks." In: *IEEE Journal on Selected Areas in Communications* 37.10 (2019), pp. 2239–2250 (cit. on pp. 100, 144).

[187] Network and Signaling Working Group: Verizon, Cisco, Ericsson, Intel, LG, Nokia, Samsung, Qualcomm. *Verizon 5G TF; Network and Signaling Working Group; Verizon 5th Generation Radio Access; Overall Description (Release 1)*. Tech. rep. 2016 (cit. on pp. 155, 157).

[188] Nginx Developers. *NGINX: High Performance Load Balancer, Web Server, and Reverse Proxy*. `https://www.nginx.com/` (accessed Aug 8, 2021). 2021 (cit. on p. 87).

[189] Thi-Minh Nguyen, Serge Fdida, and Tuan-Minh Pham. "A comprehensive resource management and placement for network function virtualization." In: *IEEE Conference on Network Softwarization (NetSoft)*. IEEE. 2017, pp. 1–9 (cit. on p. 81).

[190] Gaurav Nigam, Paolo Minero, and Martin Haenggi. "Coordinated multipoint joint transmission in heterogeneous networks." In: *IEEE Transactions on Communications* 62.11 (2014), pp. 4134–4146 (cit. on p. 144).

[191] Dusit Niyato and Ekram Hossain. "Dynamics of network selection in heterogeneous wireless networks: An evolutionary game approach." In: *IEEE Transactions on Vehicular Technology (TVT)* 58.4 (2008) (cit. on p. 145).

[192] Dejene Boru Oljira, Karl-Johan Grinnemo, Javid Taheri, and Anna Brunstrom. "A model for QoS-aware VNF placement and provisioning." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2017, pp. 1–7 (cit. on pp. 1, 81).

[193] Gurobi Optimization. *Gurobi Solver*. https://www.gurobi.com/. 2020 (cit. on p. 57).

[194] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly. "SNDlib 1.0–Survivable Network Design Library." In: *Networks* 55.3 (2010), pp. 276–286. DOI: `10.1002/net.20371` (cit. on pp. 35, 57, 112, 135).

[195] Metin Ozturk, Mandar Gogate, Oluwakayode Onireti, Ahsan Adeel, Amir Hussain, and Muhammad A Imran. "A novel deep learning driven, low-cost mobility prediction approach for 5G cellular networks: The case of the Control/Data Separation Architecture (CDSA)." In: *Neurocomputing* 358 (2019), pp. 479–489 (cit. on p. 144).

[196] SK Pal and S Mitra. "Multilayer perceptron, fuzzy sets, and classification." In: *IEEE Transactions on Neural Networks* 3.5 (1992), pp. 683–697 (cit. on p. 87).

[197] Shubham Pateria, Budhitama Subagdja, Ah-hwee Tan, and Chai Quek. "Hierarchical Reinforcement Learning: A Comprehensive Survey." In: *ACM Computing Surveys (CSUR)* 54.5 (2021), pp. 1–35 (cit. on p. 169).

[198] Michael Patriksson. "A survey on the continuous nonlinear resource allocation problem." In: *European Journal of Operational Research* 185.1 (2008), pp. 1–46 (cit. on p. 87).

[199] Jianing Pei, Peilin Hong, Miao Pan, Jianqing Liu, and Jingsong Zhou. "Optimal VNF Placement via Deep Reinforcement Learning in SDN/NFV-Enabled Networks." In: *IEEE Journal on Selected Areas in Communications* (2019) (cit. on pp. 100, 123).

[200] Manuel Peuster and Holger Karl. "Understand Your Chains and Keep Your Deadlines: Introducing Time-constrained Profiling for NFV." In: *IEEE/IFIP International Conference on Network and Service Management (CNSM)*. IEEE/IFIP. 2018, pp. 240–246 (cit. on pp. 84, 85).

[201] Manuel Peuster and Holger Karl. "Understand Your Chains: Towards Performance Profile-based Network Service Management." In: *5th European Workshop on Software Defined Networks (EWSDN'16)*. IEEE. 2016 (cit. on pp. 80, 81, 85).

[202] David Pisinger and Stefan Ropke. "Large neighborhood search." In: *Handbook of Metaheuristics*. Springer, 2010, pp. 399–419 (cit. on p. 34).

[203] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. "FairCloud: Sharing the Network in Cloud Computing." In: *ACM SIGCOMM Conference*. 2012, pp. 187–198 (cit. on pp. 2, 24, 41, 99).

[204] Prometheus. *Documentation*. `https://prometheus.io/docs/prometheus/latest/configuration/configuration/` (March 18, 2020). 2020 (cit. on pp. 98, 108, 123).

[205] Erika Puiutta and Eric MSP Veith. "Explainable reinforcement learning: A survey." In: *International Cross-Domain Conference for Machine Learning and Knowledge Extraction*. Springer. 2020, pp. 77–95 (cit. on p. 170).

[206] Faizan Qamar, Kaharudin Bin Dimyati, Mhd Nour Hindia, Kamarul Ariffin Bin Noordin, and Ahmed M Al-Samman. "A comprehensive review on coordinated multi-point operation for LTE-A." In: *Computer Networks* 123 (2017), pp. 19–37 (cit. on pp. 143, 158).

[207] Haydar Qarawlus. "Deep Reinforcement Learning for Scaling, Placement, and Routing in Network Function Virtualization." Master's Thesis. Paderborn University, Germany, 2020 (cit. on pp. 9, 121).

[208] Yinan Qi, Mythri Hunukumbure, Maziar Nekovee, Javier Lorca, and Victoria Sgardoni. "Quantifying data rate and bandwidth requirements for immersive 5G experience." In: *2016 IEEE International Conference on Communications Workshops (ICC)*. IEEE. 2016, pp. 455–461 (cit. on p. 162).

[209] Pham Tran Anh Quang, Yassine Hadjadj-Aoul, and Abdelkader Outtagarts. "A deep reinforcement learning approach for VNF Forwarding Graph Embedding." In: *IEEE Transactions on Network and Service Management* 16.4 (2019), pp. 1318–1331 (cit. on pp. 100, 123).

[210] P. Quinn, U. Elzur, and C. Pignataro. *Network Service Header (NSH)*. Internet Request for Comments RFC 8300. IETF, 2018, pp. 1–40 (cit. on pp. 63, 126).

[211] Sabidur Rahman, Tanjila Ahmed, Minh Huynh, Massimo Tornatore, and Biswanath Mukherjee. "Auto-Scaling Network Service Chains using Machine Learning and Negotiation Game." In: *IEEE Transactions on Network and Service Management* (2020) (cit. on p. 99).

[212] Talha Faizur Rahman. "Broadband Radio Interfaces Design for "4G and Beyond" Cellular Systems in Smart Urban Environments." PhD Thesis. Trento University, Italy, 2015 (cit. on p. 15).

[213] Rancher. *Rancher: From datacenter to cloud to edge, Rancher lets you deliver Kubernetes-as-a-Service*. `https://rancher.com/` (Feb 11, 2021). 2021 (cit. on p. 14).

[214] Varun S Reddy, Andreas Baumgartner, and Thomas Bauschert. "Robust embedding of VNF/service chains with delay bounds." In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*. IEEE. 2016, pp. 93–99 (cit. on pp. 80, 81).

[215] Roberto Riggio, Shah Nawaz Khan, Tejas Subramanya, Imen Grida Ben Yahia, and Diego Lopez. "LightMANO: Converging NFV and SDN at the Edges of the Network." In: *IEEE/IFIP Network Operations and Management Symposium (NOMS)*. IEEE. 2018, pp. 1–9 (cit. on p. 132).

[216] R. V. Rosa, C. Bertoldo, and C. E. Rothenberg. "Take Your VNF to the Gym: A Testing Framework for Automated NFV Performance Benchmarking." In: *IEEE Communications Magazine* 55.9 (2017), pp. 110–117. ISSN: 0163-6804. DOI: `10.1109/MCOM.2017.1700127` (cit. on pp. 80, 81, 85).

[217] Sebastian Ruder. "An overview of gradient descent optimization algorithms." In: *arXiv preprint arXiv:1609.04747* (2016) (cit. on p. 133).

[218] Ryu Contributors. *Ryu: Component-Based Software Defined Networking Framework*. `https://ryu-sdn.org/` (accessed Feb 10, 2021). 2021 (cit. on p. 13).

[219] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. "Autopilot: Workload autoscaling at Google." In: *European Conference on Computer Systems*. 2020, pp. 1–16 (cit. on p. 99).

[220] Omer Sagi and Lior Rokach. "Ensemble learning: A survey." In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8.4 (2018), e1249 (cit. on p. 86).

[221] Fady Samuel, Mosharaf Chowdhury, and Raouf Boutaba. "PolyViNE: Policy-Based Virtual Network Embedding Across Multiple Domains." In: *Journal of Internet Services and Applications* 4.1 (2013), p. 6 (cit. on p. 41).

[222] Narayanan Puthenpurayil Satheeschandran. "Performance Prediction in Virtual Network Function Placement Using Machine Learning Techniques." Master's Thesis. Paderborn University, Germany, 2019 (cit. on pp. 8, 79).

[223] William Saunders, Girish Sastry, Andreas Stuhlmüller, and Owain Evans. "Trial without Error: Towards Safe Reinforcement Learning via Human Intervention." In: *International Conference on Autonomous Agents and MultiAgent Systems*. 2018, pp. 2067–2069 (cit. on p. 170).

[224] Ankit Saxena and Ravi Sindal. "Strategy for resource allocation in LTE-A." In: *International Conference on Signal Processing, Communication, Power and Embedded System (SCOPES)*. IEEE. 2016, pp. 29–34 (cit. on p. 159).

[225] Liron Schiff, Stefan Schmid, and Petr Kuznetsov. "In-band synchronization for distributed SDN control planes." In: *ACM SIGCOMM Computer Communication Review* 46.1 (2016), pp. 37–43 (cit. on p. 104).

[226] Simon Schmitt, Jonathan J Hudson, Augustin Zidek, Simon Osindero, Carl Doersch, Wojciech M Czarnecki, Joel Z Leibo, Heinrich Kuttler, Andrew Zisserman, Karen Simonyan, et al. "Kickstarting deep reinforcement learning." In: *Computing Research Repository (CoRR)*. 2018 (cit. on p. 170).

[227] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. "Trust region policy optimization." In: *International Conference on Machine Learning (ICML)*. 2015, pp. 1889–1897 (cit. on p. 131).

[228] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. "Proximal policy optimization algorithms." In: *arXiv preprint arXiv:1707.06347* (2017) (cit. on pp. 131, 149).

[229] Stefano Secci, Jean-Louis Rougier, and Achille Pattavina. "On the Selection of Optimal Diverse AS-Paths for Inter-Domain IP/(G) MPLS Tunnel Provisioning." In: *Telecommunication Networking Workshop on QoS in Multiservice IP Networks*. IEEE. 2008, pp. 235–241 (cit. on p. 42).

[230] Krisantus Sembiring and Andreas Beyer. "Dynamic resource allocation for cloud-based media processing." In: *ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*. 2013, pp. 49–54 (cit. on p. 82).

[231] Nadir Shah, Paolo Giaccone, Danda B Rawat, Ammar Rayes, and Nan Zhao. *Solutions for adopting software defined network in practice*. 2019 (cit. on p. 12).

[232] Ryan Shea, Jiangchuan Liu, Edith C-H Ngai, and Yong Cui. "Cloud gaming: Architecture and performance." In: *IEEE network* 27.4 (2013), pp. 16–21 (cit. on p. 1).

[233] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. "Making middleboxes someone else's problem: Network processing as a cloud service." In: *ACM SIGCOMM Computer Communication Review* 42.4 (2012), pp. 13–24 (cit. on p. 13).

[234] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. "Edge computing: Vision and challenges." In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646 (cit. on p. 14).

[235] Xingui Shi, Xiangming Wen, Yong Sun, Linyan Li, and Wenmin Ma. "A novel distributed VNet mapping algorithm." In: *IEEE International Conference on Communication Technology*. IEEE. 2012, pp. 311–316 (cit. on p. 62).

[236] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. "Mastering the game of go without human knowledge." In: *Nature* 550.7676 (2017), pp. 354–359 (cit. on p. 119).

[237] Software Campus. *Software Campus: Your Path to Becoming an IT Executive*. `https://softwarecampus.de/en/` (Feb 4, 2021). 2021 (cit. on p. 5).

[238] Ruben Solozabal, Josu Ceberio, Aitor Sanchoyerto, Luis Zabala, Bego Blanco, and Fidel Liberal. "Virtual Network Function Placement Optimization With Deep Reinforcement Learning." In: *IEEE Journal on Selected Areas in Communications* 38.2 (2019), pp. 292–303 (cit. on p. 100).

[239] An Song, Wei-Neng Chen, Tianlong Gu, Huaqiang Yuan, Sam Kwong, and Jun Zhang. "Distributed Virtual Network Embedding System With Historical Archives and Set-Based Particle Swarm Optimization." In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* (2019) (cit. on p. 62).

[240] Squid Developers. *Squid Cache: Optimising Web Delivery*. `http://www.squid-cache.org/` (accessed Aug 8, 2021). 2021 (cit. on p. 87).

[241] Emma Strubell, Ananya Ganesh, and Andrew McCallum. "Energy and policy considerations for deep learning in NLP." In: *Annual Meeting of the Association for Computational Linguistics (ACL)*. 2019 (cit. on p. 119).

[242] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018 (cit. on pp. 132, 142, 143, 149, 150, 154, 155).

[243] Tarik Taleb, Konstantinos Samdanis, Badr Mada, Hannu Flinck, Sunny Dutta, and Dario Sabella. "On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration." In: *IEEE Communications Surveys & Tutorials* 19.3 (2017), pp. 1657–1681 (cit. on pp. 155, 157).

[244] Ole Tange et al. "GNU parallel – The command-line power tool." In: *The USENIX Magazine* 36.1 (2011), pp. 42–47 (cit. on p. 35).

[245] Sahrish Khan Tayyaba, Hasan Ali Khattak, Ahmad Almogren, Munam Ali Shah, Ikram Ud Din, Ibrahim Alkhalifa, and Mohsen Guizani. "5G vehicular network resource management for improving radio access through machine learning." In: *IEEE Access* 8 (2020), pp. 6792–6800 (cit. on p. 1).

[246] Rachel Thomas. *Google's AutoML: Cutting Through the Hype*. `https://www.fast.ai/2018/07/23/auto-ml-3/` (accessed June 15, 2021). 2018 (cit. on p. 142).

[247] Antti Tolli, Hadi Ghauch, Jarkko Kaleva, Petri Komulainen, Mats Bengtsson, Mikael Skoglund, Michael Honig, Eeva Lahetkangas, Esa Tiirola, and Kari Pajukoski. "Distributed coordinated transmission with forward-backward training for 5G radio access." In: *IEEE Communications Magazine* 57.1 (2019), pp. 58–64 (cit. on p. 143).

[248] Antti Toskala and Harri Holma. *WCDMA For UMTS: HSDPA Evolution And LTE*. Wiley, 2007 (cit. on p. 158).

[249] Alexander Trott, Stephan Zheng, Caiming Xiong, and Richard Socher. "Keeping your distance: Solving sparse reward tasks using self-balancing shaped rewards." In: *Advances in Neural Information Processing Systems (NeurIPS)*. 2019, pp. 10376–10386 (cit. on p. 129).

[250] George Tychogiorgos, Athanasios Gkelias, and Kin K Leung. "Towards a fair non-convex resource allocation in wireless networks." In: *IEEE International Symposium on Personal, Indoor and Mobile Radio Communications*. IEEE. 2011, pp. 36–40 (cit. on p. 87).

[251] Leslie G. Valiant. "A scheme for fast parallel communication." In: *SIAM Journal on Computing* 11.2 (1982), pp. 350–361 (cit. on p. 66).

[252] Steven Van Rossem, Wouter Tavernier, Didier Colle, Mario Pickavet, and Piet Demeester. "Profile-based Resource Allocation for Virtualized Network Functions." In: *IEEE Transactions on Network and Service Management* (2019) (cit. on pp. 80–82).

[253] Ramachandran Vijayarani and Lakshmanan Nithyanandan. "Dynamic cooperative base station selection scheme for downlink CoMP in LTE-advanced networks." In: *Wireless Personal Communications* 92.2 (2017), pp. 667–679 (cit. on pp. 142, 144, 146).

[254] Tingwu Wang, Renjie Liao, Jimmy Ba, and Sanja Fidler. "Nervenet: Learning structured policy with graph neural networks." In: *International Conference on Learning Representations*. 2018 (cit. on p. 170).

[255] Xiaoke Wang, Chuan Wu, Franck Le, and Francis CM Lau. "Online learning-assisted VNF service chain scaling with network uncertainties." In: *IEEE International Conference on Cloud Computing (CLOUD)*. IEEE. 2017, pp. 205–213 (cit. on pp. 2, 100, 123).

[256] Sarah Wassermann, Thibaut Cuvelier, Pavol Mulinka, and Pedro Casas. "Adaptive and Reinforcement Learning Approaches for Online Network Monitoring and Analysis." In: *IEEE Transactions on Network and Service Management* 18.2 (2020), pp. 1832–1849 (cit. on p. 84).

[257] Qing Wei, David Perez-Caparros, and Artur Hecker. "Dynamic Flow Rules in Software Defined Networks." In: *IEEE European Workshop on Software-Defined Networks (EWSDN)*. IEEE. 2016, pp. 25–30 (cit. on pp. 64, 108).

[258] Yuhuai Wu, Elman Mansimov, Roger B Grosse, Shun Liao, and Jimmy Ba. "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation." In: *Advances in neural information processing systems (NeurIPS)*. 2017, pp. 5279–5288 (cit. on pp. 131, 132).

[259] Dionysis Xenakis, Nikos Passas, Lazaros Merakos, and Christos Verikoukis. "Mobility management for femtocells in LTE-advanced: Key aspects and survey of handover decision algorithms." In: *IEEE Communications surveys & tutorials* 16.1 (2013), pp. 64–91 (cit. on p. 143).

[260] Wenfeng Xia, Yonggang Wen, Chuan Heng Foh, Dusit Niyato, and Haiyong Xie. "A survey on software-defined networking." In: *IEEE Communications Surveys & Tutorials* 17.1 (2014), pp. 27–51 (cit. on p. 12).

[261] Yikai Xiao, Qixia Zhang, Fangming Liu, Jia Wang, Miao Zhao, Zhongxing Zhang, and Jiaxing Zhang. "NFVdeep: adaptive online service function chain deployment with deep reinforcement learning." In: *International Symposium on Quality of Service*. 2019, pp. 1–10 (cit. on pp. 100, 123).

[262] Zhifeng Xiao and Yang Xiao. "Security and privacy in cloud computing." In: *IEEE communications surveys & tutorials* 15.2 (2012), pp. 843–859 (cit. on p. 14).

[263] Annie Xie, James Harrison, and Chelsea Finn. "Deep reinforcement learning amidst lifelong non-stationarity." In: *International Conference on Machine Learning (ICML) Workshop on Lifelong ML*. 2020 (cit. on p. 145).

[264] Lina Xu. "Context aware traffic identification kit (TriCK) for network selection in future HetNets/5G networks." In: *IEEE International Symposium on Networks, Computers and Communications (ISNCC)*. IEEE. 2017, pp. 1–5 (cit. on p. 14).

[265] Zhiyuan Xu, Jian Tang, Jingsong Meng, Weiyi Zhang, Yanzhi Wang, Chi Harold Liu, and Dejun Yang. "Experience-driven networking: A deep reinforcement learning based approach." In: *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. 2018, pp. 1871–1879 (cit. on pp. 100, 110, 123).

[266] Ji Yang, Zhang Yifan, Wang Ying, and Zhang Ping. "Average rate updating mechanism in proportional fair scheduler for HDR." In: *IEEE Global Telecommunications Conference (GLOBECOM)*. IEEE. 2004, pp. 3464–3466 (cit. on p. 160).

[267] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. "Federated machine learning: Concept and applications." In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 10.2 (2019), pp. 1–19 (cit. on pp. 151, 157).

[268] Bo Yi, Xingwei Wang, Keqin Li, Min Huang, et al. "A comprehensive survey of network function virtualization." In: *Computer Networks* 133 (2018), pp. 212–262 (cit. on pp. 1, 11).

[269] Jungkeun Yoon, Mingyan Liu, and Brian Noble. "Random waypoint considered harmful." In: *IEEE International Conference on Computer Communications (INFOCOM)*. Vol. 2. IEEE. 2003, pp. 1312–1321 (cit. on p. 158).

[270] Lei You and Di Yuan. "Joint CoMP-cell selection and resource allocation in fronthaul-constrained C-RAN." In: *International Symposium on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt)*. IEEE. 2017, pp. 1–6 (cit. on pp. 142, 144, 156).

[271] Faqir Zarrar Yousaf, Michael Bredel, Sibylle Schaller, and Fabian Schneider. "NFV and SDN—Key technology enablers for 5G networks." In: *IEEE Journal on Selected Areas in Communications* 35.11 (2017), pp. 2468–2478 (cit. on p. 15).

[272] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. "Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration." In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 17–29 (cit. on p. 42).

[273] Chaoyun Zhang, Paul Patras, and Hamed Haddadi. "Deep learning in mobile and wireless networking: A survey." In: *IEEE Communications Surveys & Tutorials* 21.3 (2019), pp. 2224–2287 (cit. on p. 2).

[274] Chongjie Zhang and Victor Lesser. "Coordinating multi-agent reinforcement learning with limited communication." In: *International Conference on Autonomous Agents and Multi-agent Systems*. 2013, pp. 1101–1108 (cit. on p. 143).

[275] Qi Zhang, Quanyan Zhu, Mohamed Faten Zhani, Raouf Boutaba, and Joseph L Hellerstein. "Dynamic service placement in geographically distributed clouds." In: *IEEE Journal on Selected Areas in Communications* 31.12 (2013), pp. 762–772 (cit. on p. 1).

[276] Qixia Zhang, Yikai Xiao, Fangming Liu, John CS Lui, Jian Guo, and Tao Wang. "Joint optimization of chain placement and request scheduling for network function virtualization." In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2017, pp. 731–741 (cit. on p. 99).

[277] Xiaoxi Zhang, Chuan Wu, Zongpeng Li, and Francis CM Lau. "Proactive vnf provisioning with multi-timescale cloud resources: Fusing online learning and online optimization." In: *IEEE International Conference on Computer Communications (INFOCOM)*. IEEE. 2017, pp. 1–9 (cit. on pp. 82, 99).

[278] Xu Zhang, Hao Chen, Yangchao Zhao, Zhan Ma, Yiling Xu, Haojun Huang, Hao Yin, and Dapeng Oliver Wu. "Improving cloud gaming experience through mobile edge computing." In: *IEEE Wireless Communications* 26.4 (2019), pp. 178–183 (cit. on p. 1).

[279] Ziyao Zhang, Liang Ma, Kin K Leung, Leandros Tassiulas, and Jeremy Tucker. "Q-placement: Reinforcement-learning-based service placement in software-defined networks." In: *IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE. 2018, pp. 1527–1532 (cit. on p. 123).

[280] Donghao Zhou, Zheng Yan, Yulong Fu, and Zhen Yao. "A survey on network data collection." In: *Journal of Network and Computer Applications* 116 (2018), pp. 9–23 (cit. on p. 108).

[281] Yiming Zhou, Cong Shen, and Mihaela van der Schaar. "A non-stationary online learning approach to mobility management." In: *IEEE Transactions on Wireless Communications* 18.2 (2019), pp. 1434–1446 (cit. on pp. 142, 145, 146, 159).