Sébastien Auroux

Flow Processing-aware Control Application Placement

Dissertation

submitted to the

Faculty of Electrical Engineering,
Computer Science, and Mathematics

in partial fulfillment of the requirements for the degree of

Doctor rerum naturalium (Dr. rer. nat.)

Paderborn, October 2017

Referees:
Prof. Dr. Holger Karl, University of Paderborn, Germany
Dr. Pablo Serrano Yáñez-Mingot, Universidad Carlos III de Madrid, Spain

Submission: October 2017

Abstract

The traffic demand in mobile access networks has grown substantially in recent years and is expected to continue to do so, both in terms of total volume and data rate required by individual users. The infrastructure of mobile access networks has to keep up with this trend and provide the data rates to satisfy the increasing demands. To achieve this, employing coordination mechanisms is essential to use the available resources efficiently. By exploiting recent network softwarization approaches, such coordination mechanisms can be handled by virtualized *Control Applications (CAs)* that can be flexibly positioned in the network.

In my thesis, I explore the problem of placing these CAs appropriately in the backhaul network of a mobile access network, which I introduce as *Flow processing-aware Control Application Placement Problem (FCAPP)*. FCAPP is a challenging placement problem including tight latency, data rate and processing capacity constraints on the backhaul infrastructure. In particular, coordination mechanisms require a considerable amount of control information and user data to be exchanged between the base stations and to be jointly processed at the host of a CA. To tackle this, FCAPP considers *Data Flow Groups (DFGs)*, a concept that ensures the aforementioned joint processing and, in addition, also allows to express various types of coordination mechanisms.

Over the course of my research, I have considered multiple variations and several solution approaches for FCAPP to (1) efficiently decide initial CA placement and (2) to quickly and flexibly adapt placement decisions during network operation in reaction to traffic load changes. In this thesis, I describe my investigation results on FCAPP, my developed solution approaches and I present extensive evaluation results for all of them. Most notably, I present a fast centralized placement framework including prototype implementation and a distributed algorithm, which both fulfill the aforementioned goals.

Zusammenfassung

Der Datenverkehr in mobilen Zugangsnetzen ist in den letzten Jahren erheblich gewachsen, sowohl im Bezug auf das gesamte Datenvolumen, als auch im Bezug auf die Anforderungen einzelner Nutzer. Um diesen steigenden Anforderungen gerecht zu werden, muss die Infrastruktur mobiler Zugangsnetze diesem Trend standhalten und die geforderten Datenraten zur Verfügung stellen. Damit dies möglich ist, ist es unter anderem essentiell, Koordinationsmechanismen einzusetzen, die für eine effiziente Nutzung der vorhandenen Ressourcen sorgen. Durch die Nutzung jüngster Ansätze zur Realisierung von Netzwerkaspekten durch Software ist es möglich, diese Koordinationsmechanismen mit Hilfe virtualisierter *Kontrollapplikationen*, welche flexibel im Netzwerk positioniert werden können, zu realisieren.

In meiner Arbeit erforsche ich das Problem, diese Kontrollapplikationen angemessen innerhalb des Backhaul-Netzwerks eines mobilen Zugangsnetzes zu platzieren. Ich bezeichne dieses Problem als *Flow processing-aware Control Application Placement Problem (FCAPP)*. FCAPP ist ein anspruchsvolles Optimierungsproblem mit strikten Latenz-, Datenraten- und Verarbeitungsanforderungen für die Backhaul-Infrastruktur. Insbesondere erfordern Koordinationmechanismen, dass eine beträchtliche Menge an Kontrollinformationen und Benutzerdaten zwischen verschiedenen Basisstationen ausgetauscht und gemeinsam am Ausführungsort einer Kontrollapplikationen verarbeitet werden. Um dies zu bewältigen betrachtet FCAPP *Data Flow Groups (DFGs)*. Dabei handelt es sich um ein Konzept, welches die gemeinsame Datenverarbeitung sicherstellt und darüber hinaus auch verschiedene Arten von Koordinationsmechanismen ausdrücken kann.

Im Laufe meiner Forschung habe ich mehrere Variationen und verschiedene Lösungsansätze für FCAPP in Betracht gezogen, um (1) die anfängliche Platzierung von Kontrollapplikationen effizient zu entscheiden und (2) die Platzierungsentscheidungen während des Netzbetriebs schnell und flexibel an die sich ändernde Verkehrslast anzupassen. In dieser Arbeit beschreibe ich meine Untersuchungsergebnisse zu FCAPP, meine entwickelten Lösungsansätze und präsentiere für jeden von ihnen umfangreiche Evaluierungsergebnisse. Insbesondere stelle ich ein schnelles zentralisiertes Platzierungs-Framework mit Prototyp-Implementierung und einem verteilten Algorithmus vor, die beide jeweils die oben genannten Ziele erfüllen.

Acknowledgements

First, I would like to express my appreciation and thanks to Holger Karl for supervising me over the course of my research and for advising me while writing this thesis. You were always open for detailed discussions, always gave valuable feedback and your adamant attention to details definitely drove me to improve my writing style continuously. It is particularly your achievement that I write this sentence without using a comma!

I would also like to thank my current and former colleagues at Paderborn University and from my research projects for all the interesting discussions and memorable shared experiences. A special thanks goes to Pablo Serrano for unhesitatingly accepting to be the second reviewer of my thesis.

Last, but certainly not least, I would like to express my deep-felt gratitude to Laura and to my parents. Laura, thank you so much for your loving support throughout this journey and for always pushing me when needed! Mama, Papa, I just want to let you know how lucky I feel to have such loving and supporting parents. Without your continuous support throughout my life, all of this would not have been possible!

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

3GPP	Third Generation Partnership Project
API	Application Programming Interface
BS	Base Station
CA	Control Application
CoMP	Coordinated Multi-Point
DFG	Data Flow Group
DPI	Deep Packet Inspector
DistCAPA	Distributed flow processing-aware Control Application Placement Algorithm
ETSI	European Telecommunications Standards Institute
FCAPP	Flow processing-aware Control Application Placement Problem
FlexCAPF	Flexible flow processing-aware Control Application Placement Framework
FLP	Facility Location Problem
GA	Genetic Algorithm
GreedyFCAPA	Greedy Flow processing-aware Control Application Placement Algorithm
ICIC	Inter-Cell Interference Coordination
ISP	Internet Service Provider
JB	Joint Beamforming
JP	Joint Processing
JS	Joint Scheduling
LCA	Local Control Application
LDF	Least Demanding First
LTE	Long Term Evolution
MANO	Management and Orchestration
MDF	Most Demanding First
MIQCP	Mixed Integer Quadratically Constrained Program

NAT	Network Address Translator
NFV	Network Function Virtualization
NFVI	Network Function Virtualization Infrastructure
OFDP	OpenFlow Discovery Protocol
ONF	Open Networking Foundation
RAN	Radio Access Network
RCA	Regional Control Application
REST	Representational State Transfer
SDN	Software-Defined Networking
SINR	Signal to Interference plus Noise Ratio
SNR	Signal to Noise Ratio
TAP	Traffic Aggregation Point
TCP	Transmission Control Protocol
TSP	Travelling Salesman Problem
UDP	User Datagram Protocol
UE	User Equipment
VM	Virtual Machine
VNE	Virtual Network Embedding
VNF	Virtual Network Function
WMN	Wireless Mesh Network
WSN	Wireless Sensor Network

1

Introduction

1.1 Motivation

Observing the trend from recent years, a big challenge for current and future wireless access networks is the consistently increasing amount of smart User Equipment (UE) (smartphones, tablets, etc.) and the exponential growth of traffic volume that has to be handled by a network's infrastructure [1, 2, 3]. For instance, the most recent "Global Mobile Data Traffic Forecast" [3] published by Cisco in 2017 provides very interesting insights into current and future developments of these characteristics in mobile access networks. While mobile networks carried 400 petabytes per month in 2011, this number reached 7.2 exabytes per month at the end of 2016, which represents an 18-fold increase in the last 5 years. To put these number into perspective, one has to visualize that the traffic volume of the entire global internet in the year 2000 has been around one exabyte [1]. This increase is in line with the consistent increase of UEs in mobile access networks, which grew from 6.5 billion devices in 2012 to 8 billion devices in 2016. At the same time, the average mobile network downstream speed per UE increased from 0.5 Mbit/s to 6.8 Mbit/s. According to the study's predictions, mobile data traffic is further expected to grow to 49 exabytes per month in 2021, while simultaneously the number of UEs is expected to reach 11.6 billions with an average mobile network downstream speed of over 20 Mbit/s per UE.

Of course, the infrastructure of mobile access networks has to keep up with this trend and provide the data rates to satisfy the increasing demands. Mobile network operators already deploy denser and more heterogeneous cellular networks to meet these requirements [4]. But because of resulting issues, such as inter-cell interference, simply using more and more network equipment or increasing the physical layer capacity by using more spectrum is insufficient. Additionally, it is necessary to enable efficient usage of the available network resources by coordination mechanisms [5]. Some of these mechanisms concern network control, such as Inter-Cell Interference Coordination (ICIC) mechanisms or more generally Software-Defined Networking (SDN) mechanisms, and come with low-latency requirements. Other mechanisms additionally

demand high processing capacity and high data rate from the network infrastructure, e.g. Coordinated Multi-Point (CoMP) transmission and reception. In total, future wireless access networks are expected to include a vast range of different coordination mechanisms [6].

But while the scope and nature of these mechanisms is manifold, all of them induce several *data flows* in the network and result in considerable data processing work to be handled, usually under stringent latency and data rate constraints. These constraints mainly apply to the underlying network, which transports data between the radio access network and the core backbone network. This network is called *backhaul network* (see Section 2.1.1). In particular, with the increase in traffic handled by the radio access network, the backhaul network has to support more control information that has to be exchanged between the Base Stations (BSs) and which has to be processed to coordinate and schedule wireless transmissions. Enabling this and preventing the backhaul network from becoming a bottleneck requires efficient and flexible management of the backhaul network.

Independent of the increasing traffic volume, another trend for future networks is the so-called *network softwarization*, mainly enabled by SDN and Network Function Virtualization (NFV) [7]. The SDN concept decouples network control and packet forwarding in a network by moving network control decisions to a centralized control entity (see Section 2.1.3). Recent work also includes the idea to apply the SDN concept to wireless coordination [8, 9, 10, 11], i.e. the control of coordination mechanisms is moved to a controller node. Accordingly, this results in latency and data rate requirements for the backhaul connection between the controller node and the coordinated BSs to transport the required control information and in processing requirements for the designated controller node for making the required coordination decisions. In particular, it is essential that the control data of the coordinated BSs is collected and *jointly* processed at the same controller node. Moreover, for coordination mechanisms like CoMP (see Section 2.1.2), it can also be necessary to forward and process the user data at the controller node. Since user data is significantly larger in size than the aforementioned control information in general, this further increases the requirements for the controller node and the backhaul network.

Meanwhile, the NFV paradigm is about implementing network functions as *software applications* and thus decoupling them from *dedicated* physical devices (see Section 2.1.4). This concept not only allows to implement network functions, such as the aforementioned controller nodes, as software-based *Control Applications (CAs)*, but also to instantiate these CAs in the network flexibly. On the one hand, this possibility for flexible deployment is a great opportunity, but on the other hand, it brings the challenge to place these applications adequately and efficiently, while also considering all relevant routing requirements. Finding good locations in the network to place CAs is a non-trivial task; as introduced above it requires taking into account tight latency, data rate and processing capacity constraints.

1.2 Contribution

To assist efficient management of future wireless networks, I have decided to investigate the placement problem described in the previous section, which I introduce as *Flow processing-aware Control Application Placement Problem (FCAPP)*. To reduce complexity and to benefit network mechanisms that combine control and data processing aspects (like CoMP), I claim that co-locating both network control and data processing in one CA is desirable. Further, as stated in the previous section, it is not always possible to handle data flows containing control information (and possibly user data) separately since this data has to be jointly processed at a CA. To cope with this, I define *Data Flow Groups (DFGs)* consisting of one or multiple data flows which demand *joint* processing at the same CA. The concept of DFGs not only ensures that all related data is jointly processed but also allows to express various types of coordination mechanisms as a DFG, such as CoMP transmission/reception or other ICIC approaches.

As a result, FCAPP combines latency, data rate and processing capacity constraints in one placement problem. Due to the flexible DFG concept, FCAPP can be applied to all types of coordination mechanisms that allow centralized execution and that can be flexibly instantiated on a host providing the required hardware resources. Even though my work focuses on coordination mechanisms as application scenario, DFGs can also be used to express other types of network mechanisms, which further extends the possible application field for FCAPP.

While there are related problems which partially consider the constraints relevant for FCAPP (see Section 2.3), I did not find any research covering the full extent of FCAPP, which therefore represented an open challenge for future networks. Moreover, many related problems target long-term placements in the field of network planning. But this is not sufficient for mobile access networks where traffic load can change very quickly. In a modern dense and crowded network, many data flows appear and expire every second during high load periods. Further, flash-crowd effects can suddenly cause drastic load changes. Therefore, placing CAs that execute coordination mechanisms in mobile wireless access networks generally requires *flexible operation* on shorter timescales down to seconds and possibly less – yet another challenging task.

All of these challenges have led me to focus my research on the question whether or not it is feasible to

(1) efficiently decide CA placement considering all latency, data rate and processing capacity constraints and
(2) doing this within the order of seconds to milliseconds to flexibly adapt placement decisions during network operation in reaction to traffic load changes to maintain near-optimal network performance.

During the course of my research, I have worked with several variations of FCAPP and developed efficient solution approaches for all of them. I

formulated optimization models, which mainly serve as reference models in the context of my work, as well as various heuristic algorithms, each bringing a special merit depending on the application scenario's concrete use case requirements. As a key result, I propose a heuristic framework that is able to place and flexibly reassign CAs fast and efficiently. Further, I present a distributed algorithm that fulfills the same tasks without the assumption of having all network information centrally available. At last, I also provide a proof of concept prototype implementation. In this thesis, I describe all of my investigation results on FCAPP, my developed solution approaches, and I present extensive evaluation results for all of them.

The results of my research have been published in several venues. The following list gives an overview of the publications that have been published and submitted over the course of my research:

Conference papers:

- S. Auroux and H. Karl. Flow processing-aware Controller Placement in Wireless DenseNets. In *Proceedings of the 25th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. IEEE, 2014

- S. Auroux and H. Karl. Efficient Flow Processing-aware Controller Placement in Future Wireless Networks. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2015

- S. Auroux, M. Dräxler, A. Morelli, and V. Mancuso. Dynamic Network Reconfiguration in Wireless DenseNets with the CROWD SDN Architecture. In *European Conference on Networks and Communications (EuCNC)*, 2015

- S. Auroux and H. Karl. Flexible reassignment of flow processing-aware controllers in future wireless networks. In *Proceedings of the 26th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. IEEE, 2015

- S. Auroux, D. Parruca, and H. Karl. Joint real-time scheduling and interference coordination for wireless factory automation. In *Proceedings of the 27th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. IEEE, 2016

- S. Auroux, S. Scholz, and H. Karl. Assessing Genetic Algorithms for Placing Flow Processing-aware Control Applications. In *Proceedings of European Wireless (EW)*, 2017

- I. Aktas, J. Ansari, S. Auroux, D. Parruca, M. Guirao, and B. Holfeld. A Coordination Architecture for Wireless Industrial Automation. In *Proceedings of European Wireless (EW)*, 2017

Submitted papers:

- H. Afifi, S. Auroux, and H. Karl. Network Function Virtualization for Wireless Acoustic Sensor Networks: An interference-aware placement and routing approach. In *Submitted to IEEE International Conference on Computer Communications (INFOCOM)*, 2018

- H. Afifi, S. Auroux, and H. Karl. MARVELO: Wireless Virtual Network Embedding for Overlay Graphs with Loops. In *Submitted to IEEE Wireless Communications and Networking Conference (WCNC)*, 2018

- S. Auroux and H. Karl. Distributed Placement of Virtualized Control Applications in Mobile Backhaul Networks. In *Submitted to IEEE Wireless Communications and Networking Conference (WCNC)*, 2018

1.3 Structure of the Thesis

The remainder of my thesis is structured as follows:

Chapter 2: Technical Background and Related Work
In this chapter, I explain the technical background required for the following chapters and elaborate on related work. I focus on mobile access networks, networking concepts and algorithmic concepts.

Chapter 3: Flow Processing-aware Control Application Placement with Equal-Share Scheduling [12, 13]
This chapter contains the first formalization of FCAPP based on equal-share processing scheduling. I present an optimization model as reference solution, use it to prove FCAPP to be NP-hard, and then present a multi-layer greedy heuristic called GreedyFCAPA to solve FCAPP fast and efficiently.

Chapter 4: Assessing Genetic Algorithms for Flow Processing-aware Control Application Placement [17]
After presenting a fast heuristic algorithm for FCAPP in Chapter 3, I assess whether the Genetic Algorithm concept is applicable to create heuristic solutions for FCAPP with improved solution quality compared to GreedyFCAPA. I present three Genetic Algorithm (GA) approaches for FCAPP, evaluate them and compare them to the results obtained by GreedyFCAPA.

Chapter 5: Flow Processing-aware Control Application Placement with Proportional-Share Scheduling [21]
In this chapter, I investigate FCAPP with proportional-share scheduling as a more elaborate processing scheduling approach. Again, I present an optimization model as reference solution and present a modified version of GreedyFCAPA to obtain solutions fast and efficiently with near optimal results. The solutions are evaluated and compared to the results of the variation with equal-share scheduling from Chapter 3 to assess the influence of the modified scheduling approach.

Chapter 6: Flexibly Reassigning Control Applications [14, 15]
After considering placement for fixed network states in the previous chapters, I investigate flexible reassignment of CA placement decisions in reaction to changing network load. I elaborate on basic reassignment considerations and then present a flexible placement framework called FlexCAPF, which is based on GreedyFCAPA and is able to place and flexibly reassign CAs during network operation. FlexCAPF is evaluated by means of dynamic network simulation with special regard to the gains of reassignment that takes into account a previous placement.

Chapter 7: Distributed Flow Processing-aware Control Application Placement [21]
So far, the solutions approaches presented all assumed centralized execution. In this chapter, I drop this assumption and present a distributed algorithm for FCAPP called DistCAPA. Just like FlexCAPF, DistCAPA places and flexibly reassigns CAs during network operation. DistCAPA is evaluated with simulations of static and dynamic networks and its results are compared to the ones obtained by FlexCAPF.

Chapter 8: Flow Processing-aware Control Application Placement with Backbone Extension
The network model used in the previous chapters assumed a very simplified connection of the backhaul network to the core backbone network of the operator. In this chapter, I correct this shortcoming and present an extended variation of FCAPP that takes the backbone connection into account appropriately. I describe the extended optimization model and a modified version of FlexCAPF, evaluate both of them and compare them to my previous results to analyze the effects of the modification.

Chapter 9: CoMP-based Evaluation of Flow Processing-aware Control Application Placement
To broaden my assessment of FCAPP, I study a use case based on Coordinated Multi-Point (CoMP) transmission and reception, which represents a more challenging evaluation scenario than the generic evaluation scenario used in the preceding chapters. I first elaborate on the CoMP evaluation scenario and then present and discuss the evaluation results obtained from it.

Chapter 10: SDN Testbed-based Evaluation of Flow Processing-aware Control Application Placement
In this chapter, I show how FlexCAPF can be implemented on top of an SDN-based emulated backhaul network as a proof of concept. I describe the architecture and implementation of the underlying testbed setup and then discuss the emulation results obtained from it.

Chapter 11: Conclusion and Future Research Directions
In this final chapter, I first summarize and conclude the work presented in previous chapters. Finally, I give an overview of possible further research directions for FCAPP.

2

Technical Background and Related Work

This chapter contains necessary technical background for this thesis as well as an overview of related work for FCAPP. First, I give an overview of relevant networking concepts in Section 2.1. Then, I explain the algorithmic concepts that I applied in this thesis in Section 2.2. Last, I present related work for FCAPP in Section 2.3.

2.1 Networking Concepts

In this section, I first elaborate on the hierarchical structure of mobile access networks. Then, I briefly discuss wireless coordination with CoMP techniques as an example. Finally, I summarize the concept of SDN, which represents an important background for my work and will be essential for Chapter 10, and the concept of NFV that my work builds on by considering virtualized *Control Applications (CAs)*.

2.1.1 Mobile Access Networks

For my work, I consider the hierarchical structure of a mobile access network that is commonly used since the introduction of the 3GPP LTE technology [22, 23]. This hierarchical structure consists of

- Radio Access Network (RAN),
- Backhaul Network and
- Backbone Network.

The RAN comprises all wireless connections at the edge of the entire hierarchical network. It consists of the union of all the subnetworks, each of which includes one Base Station (BS) and (usually) several User Equipments (UEs), which are wirelessly connected to that BS. Within the scope of my work, the RAN will not be directly considered, but it is still of high implicit importance

since it represents the origin of the data flows which are injected into the backhaul network.

The *backhaul network* is the part of the mobile access network that interconnects the RAN and the backbone network. The backhaul network has two crucial tasks:

1. routing user data between multiple RAN subnetworks or between the RAN and the backbone network and
2. exchanging control information between the BSs.

As described in Section 1.1, both tasks are of special interest for my work because of the additional control information and possibly user data for wireless coordination approaches that is injected into the backhaul network.

In the past, backhaul networks have been deployed using copper cables, but nowadays, backhaul networks are mostly implemented using optical fiber technologies or wireless technologies, e.g. point-to-point or point-to-multipoint over high-capacity radio links [24]. In particular, optical technology is often considered as the backhaul technology to meet future capacity demands [25, 26]. While my research is independent of the given backhaul technology, I will thus assume optical backhaul networks for my evaluation scenarios.

At last, the *backbone network*, which is also often called *core network*, is the topologically central part of a mobile access network. The backbone network is responsible for functions not related to the radio access but needed for providing a complete mobile-broadband network, e.g. like mobility management, authentication, charging functionality, and setup of end-to-end connections. It further represents the gateway to external networks [22, 27]. In particular, the backbone network can be seen as the gateway to the global internet [28]. For the major part of my research, the backbone network plays a subordinate role, but it will be targeted in Chapter 8.

Figure 2.1 provides an illustration of my considered hierarchical structure of a mobile access network.

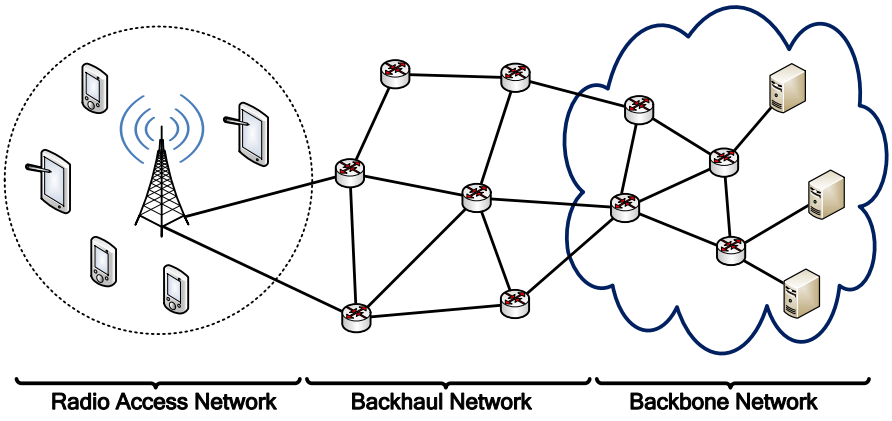


Figure 2.1: Hierarchical structure of mobile access networks

2.1.2 Wireless Coordination

Wireless coordination is a broad term to describe a vast range of mechanisms that coordinate the wireless resources of a wireless access network to efficiently use resources and to improve performance. Commonly, wireless coordination mechanisms require coordination between multiple BSs and therefore exchange control information over the backhaul network. Wireless coordination is a well investigated field and a large variety of different approaches have been presented in the past [29, 30]. For the scope of my thesis, I will only give a short overview of the CoMP approach.

The Coordinated Multi-Point (CoMP) concept describes a range of different techniques to enable coordinated transmission or reception over multiple BSs to/from a single UE to better exploit the available wireless spectrum. This is achieved by serving or receiving from a UE cooperatively by a set of BSs instead of just one BS. It is considered a powerful technology to reduce interference and increase data rates for cellular radio networks such as LTE-Advanced [31]. In addition, CoMP techniques can also enhance effective coverage area to accommodate cell-edge users by utilizing interference signals from different transmission points.

As coordination information and user data have to be exchanged among the coordinated BSs, CoMP transmission or reception requires high data rates (up to multiple Gbit/s per BSs) and low latency (down to 1 ms round trip time between the BSs) from the backhaul network [32]. There is a number of different techniques regarded as CoMP [33], achieving different gains and having different requirements for the backhaul network. But following the 3GPP LTE-Advanced terminology [34], CoMP techniques can be grouped into two main categories on which I will focus in my thesis: Joint Processing (JP) and Joint Scheduling (JS)/Joint Beamforming (JB).

JP is the most complex but also most powerful CoMP technique, bringing an expected improvement to the average cell throughput of up to 60% [35]. For the downlink case, JP brings two options: either a UE receives a joint transmission, i.e. multiple BSs send on the same physical resources simultaneously to create constructive interference at the UE, or the UE dynamically selects the BS with the best signal quality for the current data transmission. In the uplink case, the data transmitted by a UE is received by all cooperating BSs and is then sent from each to a central entity where the received versions of the UE data are jointly decoded. For my considered scenario, this particularly means that for JP the entire UE data has to be forwarded over the backhaul network to and processed at a CA, which requires a considerable amount of backhaul network resources.

Compared to JP, JS/JB is less complex and has fewer requirements but also brings a lower expected gain. In the downlink, UEs receive data transmissions only from one BS but scheduling and beamforming decisions are coordinated among the BSs. In the uplink, UEs served by different coordinated BSs are scheduled such that interference is reduced. In contrast to JP, it is only

necessary to share control information but not the entire UE data over the backhaul network.

Overall, the JP schemes with joint transmission and reception provide a more efficient utilization of the wireless resources compared to JS/JB, but also require significantly more resources from the backhaul network. Regarding the context of my thesis, CoMP partially motivates the definition of DFGs in Section 3.2 and will be an important example scenario in Chapter 9.

2.1.3 Software Defined Networking

The Software-Defined Networking (SDN) concept constitutes an important background for my work and is also the base for the prototype implementation presented in Chapter 10. As briefly indicated in Section 1.1, the SDN concept decouples network control and packet forwarding in a network by moving network control decisions to a centralized control entity, the so-called *SDN controller*. Thus, the SDN controller steers how traffic is routed within a network while the actual forwarding is executed by the *SDN-enabled network devices*, i.e. switches. This separation allows network administrators to directly and dynamically program the network's behavior via the centralized SDN controller while the physical network infrastructure is provided as an abstracted view for *SDN applications* and network services.

The Open Networking Foundation (ONF) is a non-profit, user-driven organization dedicated to accelerating the adoption of SDN and NFV and describes the different components of the SDN architecture, illustrated in Figure 2.2, as follows [36]:

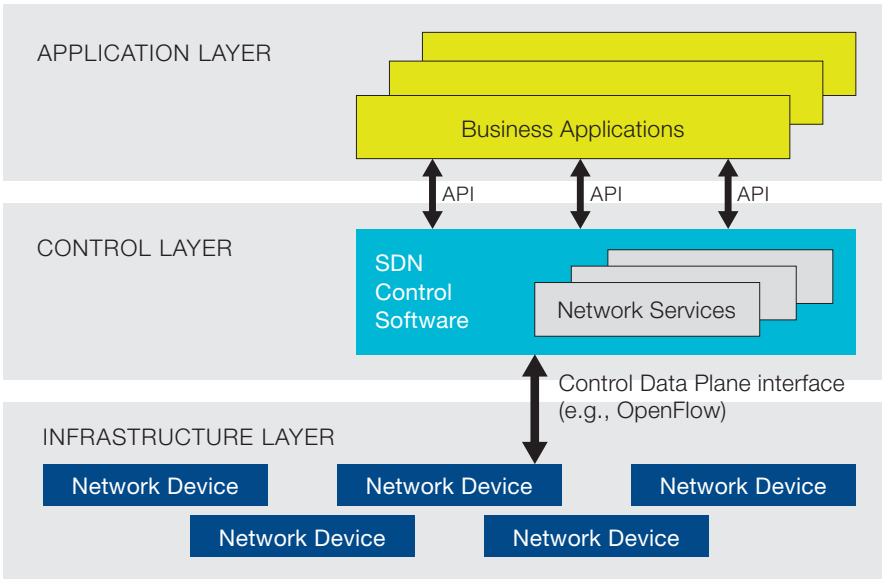


Figure 2.2: Software-defined networking architecture (from [37])

- *SDN Applications* are programs that explicitly, directly, and programmatically communicate their network requirements and desired network behavior to the SDN Controller via Application Programming Interfaces (APIs). In addition, they may consume an abstracted view of the network for their internal decision-making purposes.
- The *SDN Controller* is the logically centralized entity responsible for translating the requirements from SDN Applications down to network devices and providing SDN Applications with an abstract view of the network.
- An *SDN-enabled network device* (e.g. a switch) exposes visibility and uncontended control over its advertised capabilities via the control plane interface. The logical representation may encompass all or a subset of the physical substrate resources.

The OpenFlow protocol [38], developed at Stanford in 2008 and later standardized by the ONF, is a commonly used protocol for communication between the SDN controller and network devices. It allows direct access to and manipulation of the forwarding plane of network devices. Regarding SDN controllers, a larger range of software platforms, mostly open-source, have been developed since the emergence of SDN. One of them is Ryu [39], which is utilized in the FCAPP testbed setup in Chapter 10.

2.1.4 Network Function Virtualization

Mobile access networks or other telecommunication networks typically offer a vast range of network services, such as voice over IP or file sharing. These network services are usually composed of several network functions, like firewalls, load balancers, Network Address Translators (NATs), Deep Packet Inspectors (DPIs) and many others. In current networks, network functions are often provided by physical middle-boxes implemented on dedicated hardware platforms. However, modern networks often require more diverse and new services, potentially with only short lifecycles [40]. Satisfying these demands would require network operators to continuously invest into expensive physical equipment.

The Network Function Virtualization (NFV) concept [41], proposed by multiple leading telecommunications service providers in 2012 [42], attempts to solve this problem. The key idea of NFV is to decouple network functions that were previously deployed as middle-boxes on dedicated hardware and to realize them as Virtual Network Functions (VNFs) instead. As a result, virtual instances of the network functions can be flexibly instantiated, terminated and relocated in the network, independent from the underlying infrastructure and without dedicated hardware. Instead, many network equipment types can be consolidated into large-scale servers or data centers.

To pursue the idea of NFV, an industry specification group for NFV was formed within the European Telecommunications Standards Institute (ETSI).

Subsequently, in 2013, the group released a document including an NFV architectural framework [43] to enable dynamic construction and management of VNF instances. The three key components of the framework, illustrated in Figure 2.3, can be described as follows [43, 44]:

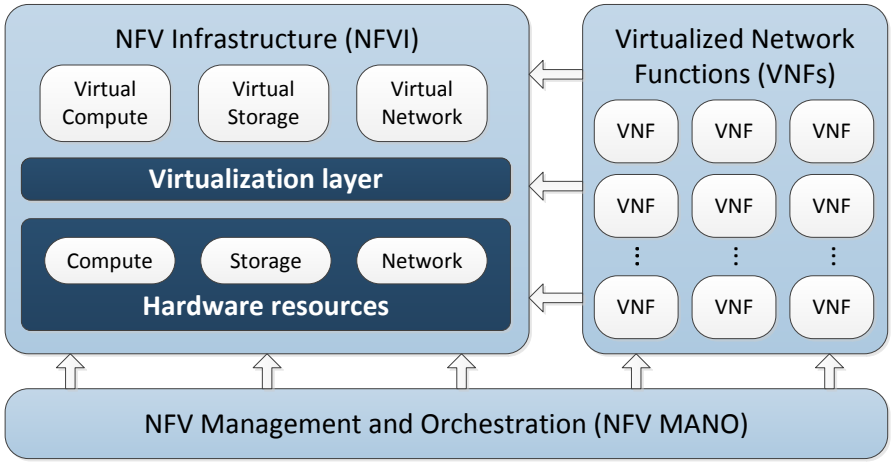


Figure 2.3: NFV architectural framework

- The *Network Function Virtualization Infrastructure (NFVI)* provides the physical and virtual resources required to support the execution of the VNFs. It includes commercial off-the-shelf hardware, accelerator components where necessary, and a software layer which virtualizes and abstracts the underlying hardware.
- The *Virtual Network Functions (VNFs)* represent a collection of software implementations of network functions that are capable of running over the NFVI.
- The *NFV Management and Orchestration (MANO)* covers lifecycle management of VNFs and the orchestration and lifecycle management of physical and/or virtual resources that support infrastructure virtualization.

The NFV concept constitutes a fundamental background for my work. As I have described in Chapter 1, my research targets the placement of virtualized CAs, which constitute a special type of VNFs. Following this, my work can be allocated in the domain of NFV MANO and my presented solution approaches can be considered as VNF placement approaches for all types of VNFs that fit into my view of CAs (see Section 3.1).

2.2 Algorithmic Concepts

In this section, I provide short overviews of the concepts of genetic and distributed algorithms, which I respectively utilize in Chapter 4 and Chapter 7. To illustrate why I consider these concepts, I will also provide some examples of work employing these concepts to problems related to FCAPP.

2.2.1 Genetic Algorithms

Genetic Algorithms (GAs) are a particular type of heuristic algorithms based on evolutionary principles and genetic operators. They are typically applied to discrete optimization problems and are known to provide good heuristic results [45]. In the past, GAs have been successfully applied to solve complex problems in various areas. Examples range from well-known problems such as the Travelling Salesman Problem (TSP) [46] over image processing [47] up to navigating robots autonomously [48].

The idea of algorithms inspired by evolutionary principles has been around at least since Turing [45], but the term *Genetic Algorithm* was coined by Holland in 1973 [49]. GAs operate on a multiset of *individuals*, called *population*, usually of constant size μ. Each individual represents a solution to the problem to solve. To evaluate the quality of a solution, a so-called *fitness function* is defined. As the inner workings of the genetic operators strongly depend on it, the choice of the representation (e.g. a string of numbers, permutation) for the individuals is critical. Their representation is often called *DNA*, consisting of several building blocks called *genes*.

During runtime, a GA creates new individuals using *genetic operators* and selects survivors based on the fitness function. Typically, a GA chooses probabilistically between *crossover operators*, i.e. creating an individual by taking genes from two selected individuals, and *mutation operators*, i.e. creating a new individual by modifying an existing one. If the problem to solve is too complex for a GA directly, so-called *hybrid GAs* are often used. They leverage other heuristic algorithms (e.g. greedy algorithms) to make decisions that the GA itself would not be able to make efficiently [50].

In total, a typical GA consists of the following steps [45]:

1) **Initialization:** The initial population is usually created randomly, leading to individuals with typically rather poor fitness values, but ensuring a high genetic diversity. Diversity is always important as low diversity can easily result in premature convergence far away from an optimal solution.
2) **Parent selection:** Individuals are chosen as *parents* for reproduction, based on their fitness value.
3) **Reproduction:** Parents reproduce using genetic operators, resulting in λ *children*.
4) **Fitness evaluation:** The newly created children need to be evaluated using a problem-dependent fitness function.
5) **Survivor selection:** The new generation of individuals is chosen by picking μ individuals from the parent generation and its offspring.
6) **Termination:** The GA repeats steps 2–5 until either a sufficiently good solution has been found or when some other termination criteria are met (e.g. the best individual's fitness did not improve for a certain number of generations).

As an example for work applying GAs to placement tasks in the networking domain, Xhafa et al. [51] use a GA approach to place routers within a Wireless Mesh Network (WMN). They experiment with several mutation operators and find that their GA approach computes router placements efficiently while almost always achieving full connectivity. The authors of [52] present a GA-based solution to determine the best sensor node placement for a given WMN. They show that their algorithm places sensor nodes better than random placement strategies in a variety of scenarios. However, the WMN scenario differs from mine significantly. A problem more akin to FCAPP is the Virtual Network Embedding (VNE) problem, which deals with embedding virtual networks in a physical substrate network [53]. Mi et. al [54] propose two GA approaches for VNE and compare them with other state of the art approaches for solving VNE. They report that the GA approaches outperform the other approaches in all considered metrics. Even though the VNE scenario misses several aspects that are relevant for solving FCAPP, most importantly the consideration of individual data flows (or DFGs), these results indicate that it could be worth to consider GAs for solving FCAPP.

2.2.2 Distributed Algorithms

Distributed algorithms [55, 56] are algorithms that are designed to run on *distributed systems*, i.e. a consortium of separated but interconnected devices (e.g. computers or routers). The devices of a distributed system are commonly called *nodes* and aim to achieve a common goal by executing a distributed algorithm. The key difference compared to a centralized algorithm is the unavailability of the entire system's state during execution. Each node is only aware of its own local state and any additional information needs to be obtained via communication with other nodes from the system.

Distributed algorithms are used for various problems and in various application areas, including telecommunications or wireless sensor networks [57, 58]. Standard problems solved by distributed algorithms include leader election, consensus, distributed search, spanning tree generation, mutual exclusion, and resource allocation [55].

Developing a distributed algorithm brings several issues. One problem often referred to is that there is no certainty about the validity of received information since the state of the sending node could already have changed once the information is received. Another common problem is the potential heterogeneity of the nodes. Significant differences in processing power at different nodes can, for example, result in an unpredictable behavior of a distributed algorithm without additional synchronization effort. There are many further possible issues, often related to node synchronization problems or unreliable communication links. Nonetheless, it often makes sense to employ distributed algorithms when it is very difficult or even impossible to obtain a system's entire state as required for centralized approaches.

Regarding the application of distributed algorithms to problems related to FCAPP, one close match is the family of Facility Location Problem (FLP) variations [59], a domain where distributed algorithms are widely employed. The authors of [60] present a distributed algorithm for uncapacitated FLP that is inspired by a centralized greedy algorithm and that iteratively selects facilities with best cost efficiency. They prove its correctness and also derive an approximation ratio. Laoutaris et al. [61] more specifically consider the uncapacitated FLP in the context of large-scale networks. The authors propose a partially distributed algorithm that is based on iterative locally centralized optimization on subgraphs. The authors conclude that their approach provides good scalability without serious performance sacrifices compared to centralized optimal solutions. Finally, Keller et al. [62] look at the capacitated FLP in the context of distributed cloud deployment. Their presented distributed algorithm is based on a centralized greedy algorithm and optimized for minimum latency. The algorithm is designed so that all nodes that are still missing a facility are searching for one that is ready to accept them. The authors report promising results and good solution qualities but indicate that finding a meaningful approximation guarantee is very hard. Even though the work by Keller et al. can partially be applied to FCAPP (as I will elaborate in Chapter 7), FLP still misses many aspects of FCAPP. In particular, the FLP is limited to assigning nodes to facilities and thus misses, for instance, the constraints for processing of individual data flows (or DFGs) included in FCAPP.

2.3 Related Work

FCAPP shares properties with various types of optimization problems. In this section, I will focus on the most relevant ones for my work: VNF placement, Virtual Machine (VM) allocation and SDN controller placement. Hereafter, I briefly present each problem, describe their connection to FCAPP, and give an overview of related work from each field.

2.3.1 Virtual Network Function Placement

The VNF placement problem is about placing VNFs within suitable infrastructures while also allocating the required resources. By this definition, the umbrella term VNF placement also comprises FCAPP as I already stated in Section 2.1.4. But in contrast to FCAPP, the work commonly associated with VNF placement considers the placement of network services composed of multiple VNFs with certain interdependencies, so-called *VNF service chains*, and is very akin to the VNE problem [63].

For example, Moens et al. [64] formulate an optimization model for VNF placement aiming to allocate a service chain onto the physical network minimizing the number of servers used. The authors of [65] propose an optimization

model to create chains of VNFs in operator networks and to deploy these chained VNFs based on requirements of the operator. Savi et al. [66] describe an approach for placing simple chains of VNFs that takes processing costs on network nodes into account. The authors of [67] propose a delay-aware solution for VNF scheduling and resource allocation for services. Finally, Xia et al. [68] propose a greedy heuristic to place and chain VNFs in a computationally efficient way.

While most available work on VNF placement deals with offline placement without considering changing demands, a few authors have also published work on VNF reassignment or automated scaling of VNFs and services. For instance, Ghaznavi et al. [69] describe a heuristic algorithm for elastic placement and reassignment of VNFs in response to on-demand workload that targets to minimize operational costs in the network. Dräxler et al. [70] propose a fully automated approach for jointly scaling and placing virtual network services. Regarding resource consideration, their approach comes very close to FCAPP by considering data rate, latency and detailed processing capacity constraints.

But naturally, the work on VNF placement does not incorporate the control aspect that FCAPP adopts from SDN. Also, most importantly, the aspect of joint data processing that is inherent to FCAPP through the concept of DFGs is non-existent in the VNF placement domain.

2.3.2 Virtual Machine Allocation

Another important field of related work is the VM allocation problem that tackles the placement of VMs on physical hosts while taking into account quality of service guarantees and the costs resulting from using the hosts [71]. Several authors have explored VM placement approaches as a solution to overcome oversubscription and improve latency within modern data center networks. Meng et al. [72] describe a traffic-aware VM placement problem to minimize communication costs between different VMs and propose a two-tier approximation algorithm to efficiently solve it. The authors of [73] introduce a network-aware orchestration layer for the discovery of related VMs with dense communication patters in order to collocate them. Alicherry et al. [74] propose a network-aware algorithm for allocating VMs in distributed cloud systems. They aim to minimize the latency between VMs allocated for a user request. Similarly, the authors of [75] describe a distributed approach optimized for response time and quick processing of user requests.

There is also a large variety of VM allocation work that considers autoscaling and reoptimization based on changed system load. For instance, Sedaghat et al. [76] describe an automated solution for adjusting both number and size of VMs in reaction to changing system load. Xiao et al. [77] present a greedy-based approach that employs simple load prediction to migrate VMs from hot spots to low-load areas.

Generally, the work on VM allocation has a lot of differences to FCAPP. First of all, most of the work within this field is tailored towards either optimization latency, which is only one aspect of FCAPP, or operational cost, which is not targeted by FCAPP. Further, similar to VNF placement, VM allocation work lacks network control aspects and detailed consideration of processing costs, which is mostly limited by the processing capacity required for hosting VMs. In particular, joint data processing as needed for my DFGs is completely missing in this domain.

2.3.3 SDN Controller Placement

Last but not least, I consider SDN controller placement as an important field of related work for FCAPP, as it features the same control aspects that are also included in FCAPP. The SDN controller placement problem has been introduced by Heller et al. [78], who examine the trade-offs between minimizing the maximum latency and the average latency in various network topologies. Other work focuses on resilience and reliability of the network [79, 80]. The authors of [79] try to optimize the resilience of the network by designing a novel metric, taking cascading failure analysis into account. Hu et al. [80] focus on network reliability instead and try to maximize the expected percentage of valid control paths with their controller placement models. The authors propose multiple heuristic algorithms, evaluate them using real Internet Service Provider (ISP) topologies and report close to optimal results with a heuristic algorithm based on simulated annealing.

Going one step further, Bari et al. [81] propose an SDN controller placement framework with periodic controller reassignment to optimize the average flow set-up time in a network and present a heuristic algorithm based on simulated annealing. Their evaluation results show that controller reassignment yields lower flow setup time and minimal communication overhead compared to static placements without reassignment. Dixit et al. [82] propose an elastic distributed controller architecture that dynamically adapts the number of controllers according to the current network load and also automatically balances the traffic load by migrating switches from overloaded controllers to lightly-loaded ones. Their evaluation demonstrates that their approach significantly reduces response times during high load periods.

But apart from the network control aspects, SDN controller placement work lacks many aspects of FCAPP such as data rate and processing capacity constraints for data flows and the resulting constraints demanded from the network elements. In general, SDN controller placement work also differs from FCAPP in terms of optimization goals as it commonly focuses on either latency or resilience/reliability. As for VM allocation above, one of these objectives is only one of multiple aspects of FCAPP and the other one is not particularly targeted.

2.3.4 Conclusion

In summary, none of the presented research fields fully covers the extent of FCAPP. Specifically, the aspect of joint data processing that I introduce via DFGs is completely missing. Of course, there are fields where such aspects appear, e.g. convergecast in Wireless Sensor Networks (WSNs) [83, 84], but none of those is further related to FCAPP. Additionally, all the fields differ in other aspects as well. The research on SDN controller placement does not consider data rate and data processing aspects, while the research on VM and VNF placement disregards control assignments and (partially) data processing aspects, all of which I combine with my work on FCAPP.

3

Flow Processing-aware Control Application Placement with Equal-Share Scheduling

In this chapter, I introduce FCAPP and present my first approaches to formalizing and solving it. For this first formulation of FCAPP, I assume *equal-share* processing scheduling (which will be explained in Section 3.2) because it is a natural and easy to implement way to distribute processing capacity among multiple entities.

After describing the target control hierarchy for FCAPP in Section 3.1 and giving the problem statement in Section 3.2, I formulate a corresponding optimization model in Section 3.3. Next, I prove the problem's complexity based on this formulation in Section 3.4 and I present a multi-layer greedy heuristic in Section 3.5. At last, I evaluate both solution approaches in Section 3.6.

3.1 Control Hierarchy

Choosing a good control hierarchy is essential for efficient network control. A flat, one-tier hierarchy would be the most simple choice, but in turn, a flat hierarchy requires a lot of communication overhead to distribute information among all control entities. A two-tier hierarchy is more complex, but an additional tier on top of the lower tier is said to allow better aggregation of network information and also to reduce signaling overhead [11, 85]. Of course, this idea can be extended to employing more than two tiers and thus aggregating information on multiple levels [86].

Which type of control hierarchy performs better in practice generally depends on the underlying network and also on the given use case; determining the best possible hierarchy based on a specific scenario is not within the scope of my work. However, using a two-tier hierarchy allows to include a one-tier hierarchy as a special case, which does not work the other way around. Further, extending a two-tier hierarchy by additional coordination tiers is conceptually

easy, for example by using recursive design [86]. Therefore, FCAPP considers the following two-tier hierarchical structure of Control Applications (CAs):

- *Local Control Applications (LCAs)* and
- *Regional Control Applications (RCAs)*.

The LCAs typically process data, operate on a local scope and on short time scales. The RCAs operate on a broader scope and coordinate the LCAs. This enables the RCAs to compensate for sub-optimal choices which may be owing to the myopic view of the LCAs. Both LCAs and RCAs are seen as CAs acting as functions in the sense of NFV. Therefore, I assume that LCAs and RCAs can be flexibly instantiated or terminated on network equipment that fulfills the necessary hardware requirements, i.e. sufficient memory and processing capacity. It is important to note that if it is desired to omit regional coordination, a one-tier hierarchy can also be expressed by simply setting all RCA requirements to zero (processing, data rate) or infinity (latency), which will be done in Chapter 7.

By way of illustration, Figure 3.1 shows a typical FCAPP scenario and a possible control structure with one RCA and two LCAs.

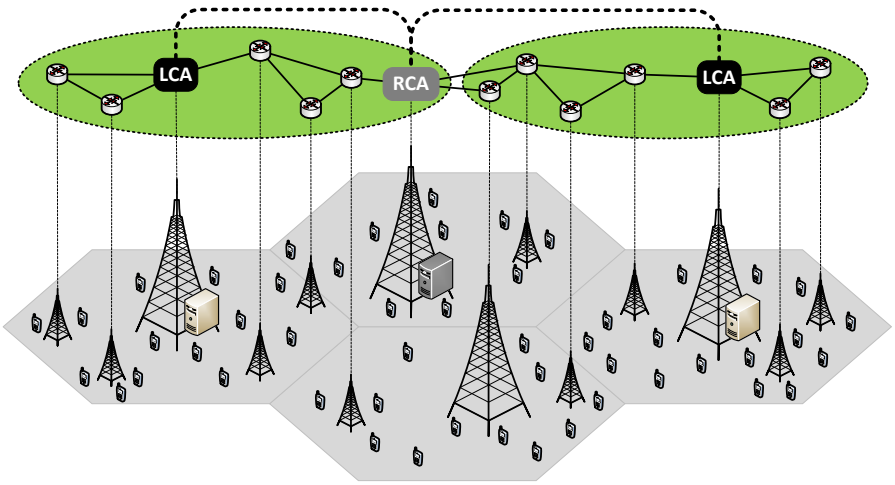


Figure 3.1: Typical FCAPP scenario

3.2 Problem Statement

FCAPP is designed to place the control hierarchy of CAs described above into a given network and considers data processing constraints as motivated in Section 1.1 – in addition to the network constraints usually associated with network control.

I consider a given backhaul network of a wireless access network as a graph $G = (V, E)$ with nodes V, i.e. BSs or switches, and undirected edges E, representing the backhaul links of the network. I assume that only certain

nodes fulfill the hardware requirements for hosting a CA, denoted by $C \subseteq V$. Every such *potential host* $c \in C$ has a processing power of $p_{\text{node}}(c)$ FLOPS and each link $(u, v) \in E$ has a maximum data rate of $b_{\text{cap}}(u, v)$ bit/s and a latency of $l_{\text{cap}}(u, v)$ seconds. The latency of a link is assumed to be independent of the network load. In particular, possible queuing delays are ignored.

A solution for FCAPP requires a *complete control structure*, i.e. each node $v \in V$ is required to be controlled by at least one LCA (a node can be controlled by more than one LCA if needed for optimal network performance) and each LCA is required to be coordinated by exactly one RCA. To process the exchanged control information, being the LCA of a node or the RCA of an LCA requires p_{LCA} or p_{RCA} operations per control information packet exchanged with a controlled node or a coordinated LCA, respectively. In this case, the routing path between a node and its LCA needs to have a round trip latency of at most l_{LCA} seconds, including the processing delay, and a minimum data rate of b_{LCA} bit/s. Similarly, a round trip latency of at most l_{RCA} seconds and a data rate of at least b_{RCA} bit/s are required for the routing path from an LCA to its RCA.

Furthermore, I consider a set F of *Data Flow Groups (DFGs)*. Each DFG consists of one or multiple (to encompass scenarios like CoMP) data flows, each entering the backhaul network at a node from the set of nodes V. Multiple data flows being part of one DFG originate from different network nodes, but all of them demand *joint* processing at the same LCA. The set of nodes that each DFG is originating from is denoted by the connection matrix $W \in \{0,1\}^{|F| \times |V|}$. Every DFG $x \in F$ requires $p_{\text{flow}}(x)$ operations per packet to be executed by the LCA by which it is processed. Further, the routing paths between the LCA and the nodes from which the DFG is originating need to provide a maximum round trip latency $l_{\text{flow}}(x)$ seconds and a minimum data rate of $b_{\text{flow}}(x)$ bit/s. For simplification, I have decided to consider the full round trip for all DFGs, so that this model fully incorporates request and response traffic.

A DFG $x \in F$ is said to be *satisfied* by an LCA $c \in C$ if and only if

(1) c controls all nodes v with $W_{x,v} = 1$ and
(2) the routing paths from all nodes v with $W_{x,v} = 1$ to c have sufficient resources to each provide a data rate of $b_{\text{flow}}(x)$ and a round trip latency $l_{\text{flow}}(x)$.

It is important to recall that the round trip latency in the network also includes the time needed for processing x at c. Therefore, to realize a round trip latency $l_{\text{flow}}(x)$, a sufficient amount of processing capacity from c has to be allocated for x to handle the required $p_{\text{flow}}(x)$ operations per packet in time, in addition to the link delays. It also has to be mentioned that the problem formulation as such only considers these required operations per packet per DFG to encompass possible scenarios, where the processing overhead for one DFG is independent of the number of included data flows. In most scenarios however, it would be intuitive that the processing overhead for one DFG should be linear in the number of included data flows. This can easily be

accomplished by setting the $p_{\text{flow}}(x)$ parameter accordingly, as done in my evaluation scenario in Section 3.6.1.

As mentioned in the beginning of this section, the scheduler for data processing at an LCA is assumed to use equal-share scheduling. This means that the processing capacity of a potential host $c \in C$ is divided *equally* between all controlled nodes, coordinated LCAs and satisfied DFGs.

The objectives of FCAPP are as follows, listed in descending order of importance:

1. create a *valid solution*, i.e. a complete control structure,
2. maximize the number of satisfied DFGs,
3. minimize the number of used CAs.

The idea behind this is as follows: While one would of course like to see all DFGs satisfied at any time, this might not always be possible because of a combination of too many DFGs and too little network resources. So in this case, I see an incomplete control structure, i.e. not all nodes are correctly controlled, as more critical for the network than a couple of yet not processed DFGs (weighing operational stability against possibly increased revenue). Further, more active CAs increase operational costs and hence saving network resources is, without doubt, an important objective. But on the other hand, dropping DFGs reduces network performance and potentially frustrates customers. Therefore, I consider the objective of minimizing the number of used CAs only feasible if it has no impact on the network's performance and I see the DFG satisfaction as more important.

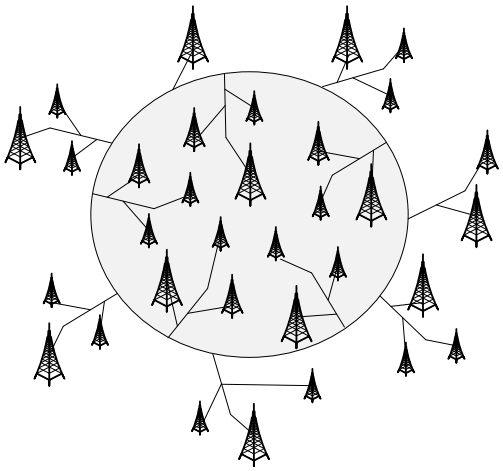


Figure 3.2: Exemplary backbone fiber ring

Regarding the backbone network behind the considered backhaul network, I limit the scope for now by assuming that all nodes are connected to the operator's core network by a suitable infrastructure, e.g. optical metro rings as deployed in metropolitan areas [87] (see Figure 3.2). Further, I assume this infrastructure to provide effectively unlimited data rate and negligible latency.

While this ignores possible effects of the underlying backbone connection for now, I will rectify this shortcoming in Chapter 8.

3.3 Optimization Model

My optimization model for FCAPP with equal-share scheduling is a Mixed Integer Quadratically Constrained Program (MIQCP) [88]. It takes the parameters listed in Table 3.1 as inputs, which correspond to the problem statement in Section 3.2. In the following, this optimization model will be denoted as OPT_{es}.

The quadratic nature of the formulation, in contrast to being entirely linear, stems from two reasons. The first reason is the two-tier control hierarchy, due to which a given node only requires an RCA if and only if it is an LCA. The second reason lies in the data rate and latency requirements for DFG satisfaction. For example, a link only needs to support a DFG's required data rate if and only if it is part of the corresponding LCA-to-node routing path *and* if the corresponding LCA actually satisfies the DFG.

Table 3.1: OPT_{es} input parameters

V	set of nodes (i.e. BSs or switches)
$C \subseteq V$	set of nodes that can serve as CA (RCA or LCA)
E	set of undirected links with $E \subseteq V \times V$
F	set of DFGs originating from at least one node $v \in V$
W	matrix with $W_{x,v} = 1$ iff $x \in F$ originates from $v \in V$
$p_{\text{node}}(c)$	processing power at node $c \in C$
$b_{\text{cap}}(v, w)$	maximum data rate for link $(v, w) \in E$
$l_{\text{cap}}(v, w)$	latency of link $(v, w) \in E$
$b_{\text{flow}}(x)$	data rate each data flow of each DFG $x \in F$ requires from the routing path to its LCA
$l_{\text{flow}}(x)$	maximum acceptable round trip latency for DFG $x \in F$ to its LCA
$p_{\text{flow}}(x)$	operations per packet required for processing DFG $x \in F$ at an LCA
b_{LCA}	data rate required from a routing path of a node to its LCA
b_{RCA}	data rate required from a routing path of an LCA to its RCA
l_{LCA}	maximum acceptable round trip latency required from a node and its LCA
l_{RCA}	maximum acceptable round trip latency required from an LCA and its RCA
p_{LCA}	operations per control information packet required at an LCA to control a node
p_{RCA}	operations per control information packet required at an RCA to coordinate an LCA

OPT_{es} determines a solution for FCAPP corresponding to the objectives defined in Section 3.2 and uses the variables listed in Table 3.2 to store the decisions about CA placement and DFG satisfaction. In order to ensure that the DFG satisfaction constraints are fulfilled, OPT_{es} further determines the corresponding routing paths. OPT_{es} uses the binary variables $f_{c,d,u,v}$ to express that $(u,v) \in E$ is used on the routing path from node $d \in V$ to LCA $c \in C$ and, analogously, the binary variables $g_{c,d,u,v}$ to describe the routing paths between an LCA and its RCA.

Table 3.2: OPT_{es} variables

$\text{LCA}_{c,v} \in \{0,1\}$	determines whether $c \in C$ is an LCA for $v \in V$
$\text{RCA}_{c,d} \in \{0,1\}$	det. whether $c \in C$ is the RCA for LCA $d \in C$
$\text{isLCA}_c \in \{0,1\}$	determines whether $c \in C$ is an LCA
$\text{isRCA}_c \in \{0,1\}$	determines whether $c \in C$ is an RCA
$\text{Sat}_{c,x} \in \{0,1\}$	determines whether $c \in C$ satisfies $x \in F$
$\text{isSat}_x \in \{0,1\}$	determines whether $x \in F$ is satisfied by an LCA
$\text{Proc}_c \in \mathbb{Z}_+$	amount of units required processing capacity from $c \in C$
$f_{c,u,v,w} \in \{0,1\}$	determines whether $(v,w) \in E$ is included in the routing path from LCA $c \in C$ to node $u \in V$
$g_{c,d,v,w} \in \{0,1\}$	determines whether $(v,w) \in E$ is included in the routing path from RCA $c \in C$ to LCA $d \in C$

The following constraints have to be met. Some constraints use a big-M constant denoted as $\mathcal{M}$, which is a very large constant.

Every node $d \in V$ needs to be controlled, i.e. there has to be a routing path starting at an LCA (3.1) and ending at d (3.2). For all intermediate nodes on a routing path, the ingress and egress have to be balanced (3.3).

$$\sum_{(c,w)\in E} f_{c,d,c,w} = \text{LCA}_{c,d}, \quad \forall c \in C, d \in V, c \neq d \tag{3.1}$$

$$\sum_{(u,d)\in E} f_{c,d,u,d} = \text{LCA}_{c,d}, \quad \forall c \in C, d \in V, c \neq d \tag{3.2}$$

$$\sum_{(u,v)\in E} f_{c,d,u,v} = \sum_{(v,w)\in E} f_{c,d,v,w}, \quad \forall c \in C,\ d, v \in V,\ c \neq d \neq v \tag{3.3}$$

Similar constraints are needed for the routing paths from each LCA to its RCA (3.4-3.6).

$$\sum_{(c,w)\in E} g_{c,d,c,w} = \text{RCA}_{c,d} \cdot \text{isLCA}_d, \quad \forall c, d \in C, c \neq d \tag{3.4}$$

$$\sum_{(v,d)\in E} g_{c,d,v,d} = \text{RCA}_{c,d} \cdot \text{isLCA}_d, \quad \forall c, d \in C, c \neq d \tag{3.5}$$

$$\sum_{(u,v)\in E} g_{c,d,u,v} = \sum_{(v,w)\in E} g_{c,d,v,w}, \quad \forall c, d \in C,\ v \in V,\ c \neq d \neq v \tag{3.6}$$

For obtaining a complete control structure, each node is required to be controlled by at least one LCA (3.7) and each LCA must be assigned to exactly one RCA (3.8).

$$\sum_{c \in C} \text{LCA}_{c,v} \geq 1, \quad \forall v \in V \tag{3.7}$$

$$\sum_{c \in C} \text{RCA}_{c,d} = \text{isLCA}_d, \quad \forall d \in C \tag{3.8}$$

Further, the LCA (3.9) and RCA (3.10) decision variables need to be set.

$$\mathcal{M} \cdot \text{isLCA}_c \geq \sum_{v \in V} \text{LCA}_{c,v}, \quad \forall c \in C \tag{3.9}$$

$$\mathcal{M} \cdot \text{isRCA}_c \geq \sum_{d \in C} \text{RCA}_{c,d}, \quad \forall c \in C \tag{3.10}$$

As explained in Section 3.2, satisfying all DFGs is not a mandatory requirement for a valid solution for FCAPP. But is has to be guaranteed that a DFG is satisfied by *at most* one LCA (3.11). If a DFG is satisfied, the corresponding decision variable needs to be set (3.12).

$$\sum_{c \in C} \text{Sat}_{c,x} \leq 1, \quad \forall x \in F \tag{3.11}$$

$$\text{isSat}_x = \sum_{c \in C} \text{Sat}_{c,x}, \quad \forall x \in F \tag{3.12}$$

But it is important to ensure that an LCA c can only satisfy a DFG x if the necessary conditions stated in Section 3.2 are fulfilled. At first, c must only satisfy x if c controls all nodes x is originating from. In other words, if there is a $v \in V$ with $W_{x,v} = 1$ and $\text{LCA}_{c,v} = 0$, then c cannot satisfy x. If this is the case, constraint (3.13) forces $\text{Sat}_{c,x}$ to be zero.

$$\text{Sat}_{c,x} \leq \text{LCA}_{c,v}, \quad \forall c \in C,\, x \in F,\, v \in V,\, W_{x,v} = 1 \tag{3.13}$$

Moreover, the data rate limits of each link (3.14) must not be exceeded.

$$\sum_{c \in C, d \in V} f_{c,d,v,w} \cdot \left(b_{\text{LCA}} + \sum_{x \in F} W_{x,d} \cdot \text{Sat}_{c,x} \cdot b_{\text{flow}}(x) \right) + \sum_{c,d \in C} g_{c,d,v,w} \cdot b_{\text{RCA}} \leq b_{\text{cap}}(v,w), \quad \forall (v,w) \in E \tag{3.14}$$

As a next step, the number of served units, i.e. controlled nodes, coordinated LCAs and satisfied DFGs, is determined for each potential host in an auxiliary variable Proc_c to ease notation later on (3.15).

$$\text{Proc}_c = \sum_{x \in F} \text{Sat}_{c,x} + \sum_{d \in V} \text{LCA}_{c,d} + \sum_{d \in C} \text{RCA}_{c,d}, \forall c \in C \tag{3.15}$$

As described earlier, the equal-share scheduling model assigns the same percentage of a host's processing capacity to each served unit. The processing capacity available for each unit can be obtained by dividing a host's processing capacity $p_{\text{node}}(c)$ by Proc_c. Then, the processing time for each unit is obtained by dividing the required amount of operations per packet by this processing capacity share. With this, the following constraints ensure that the latency requirements of each DFG (3.16), LCA (3.17) and RCA (3.18) are met, considering both processing time and link delays. Even though I do not expect control information between LCAs and nodes or RCAs and LCAs to be exchanged continuously, I still treat the corresponding constraints (3.17) and (3.18) as if that were the case as in (3.16). This ensures that the required resources are always reserved and hence available when needed.

$$\begin{aligned}
&\text{Sat}_{c,x} \cdot p_{\text{flow}}(x) \cdot \left(\frac{p_{\text{node}}(c)}{\text{Proc}_c}\right)^{-1} + \text{Sat}_{c,x} \cdot \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \Big(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v)\Big) \\
&\qquad \leq l_{\text{flow}}(x), \quad \forall c \in C, d \in V, x \in F, W_{x,d} = 1 \qquad (3.16) \\
&\text{LCA}_{c,d} \cdot p_{\text{LCA}} \cdot \left(\frac{p_{\text{node}}(c)}{\text{Proc}_c}\right)^{-1} + \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \Big(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v)\Big) \\
&\qquad \leq l_{\text{LCA}}, \quad \forall c \in C, d \in V \qquad (3.17) \\
&\text{RCA}_{c,d} \cdot p_{\text{RCA}} \cdot \left(\frac{p_{\text{node}}(c)}{\text{Proc}_c}\right)^{-1} + \sum_{(v,w)\in E} g_{c,d,v,w} \cdot \Big(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v)\Big) \\
&\qquad \leq l_{\text{RCA}}, \quad \forall c, d \in C \qquad (3.18)
\end{aligned}$$

It is noteworthy that (3.16) includes an additional multiplication by $\text{Sat}_{c,x}$, in contrast to (3.17) or (3.18). To understand this, it is convenient to first explain why the same is not needed in (3.17), in contrast. The reason lies in the nature of the $f_{c,d,v,w}$ variables that can only take on value 1 if and only if $\text{LCA}_{c,d} = 1$ due to constraints (3.1) and (3.2). With a similar connection to the $\text{Sat}_{c,x}$ variables, (3.16) could be replaced with

$$\begin{aligned}
&\text{Sat}_{c,x} \cdot p_{\text{flow}}(x) \cdot \left(\frac{p_{\text{node}}(c)}{\text{Proc}_c}\right)^{-1} + \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \Big(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v)\Big) \\
&\qquad \leq l_{\text{flow}}(x), \quad \forall c \in C, d \in V, x \in F, W_{x,d} = 1. \qquad (3.19)
\end{aligned}$$

But such a connection does not exist, which is why the additional multiplication by $\text{Sat}_{c,x}$ is generally required so that the constraint is still valid if $\text{Sat}_{c,x} = 0$. Sparing the multiplication could reduce the solution space of the model significantly or the model would even become infeasible in case of big link latency and small DFG round trip latency input values. More precisely, there could be no $c \in C, d \in V, x \in F, W_{x,d} = 1$ so that

$$\text{LCA}_{c,d} = 1 \wedge \text{Sat}_{c,x} = 0 \wedge l_{\text{flow}}(x) < \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \Big(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v)\Big)$$

since (3.19) would be violated:

$$\underbrace{\mathrm{Sat}_{c,x}}_{=0} \cdot \frac{p_{\mathrm{flow}}(x) \cdot \mathrm{Proc}_c}{p_{\mathrm{node}}(c)} \leq \underbrace{l_{\mathrm{flow}}(x) - \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \Big(l_{\mathrm{cap}}(v,w) + l_{\mathrm{cap}}(w,v) \Big)}_{<0}.$$

However, it is possible to adopt (3.19) for the optimization model if the existence of such a case can be ruled out with certainty, for example if

$$l_{\mathrm{flow}}(x) \geq \sum_{(v,w)\in E} l_{\mathrm{cap}}(v,w) \quad \forall\, x \in F. \tag{3.20}$$

This turns out to be the case for my chosen evaluation scenario from Section 3.6.1 except for large networks, for which the optimization models would not be practical anyway. Therefore, the implementation of $\mathrm{OPT}_{\mathrm{es}}$ for the evaluation in Section 3.6.2 uses constraint (3.19) if and only if condition (3.20) is fulfilled, otherwise constraint (3.16) is used.

Last but not least, it is necessary to cope with possible loop and corner cases (3.21 – 3.26).

$$\sum_{(v,w)\in E,\, w=c} f_{c,d,v,w} = 0, \quad \forall c \in C,\, d \in V \tag{3.21}$$

$$\sum_{(v,w)\in E,\, v=d} f_{c,d,v,w} = 0, \quad \forall c \in C,\, d \in V \tag{3.22}$$

$$\sum_{(v,w)\in E,\, w=c} g_{c,d,v,w} = 0, \quad \forall c \in C,\, d \in V \tag{3.23}$$

$$\sum_{(v,w)\in E,\, v=d} g_{c,d,v,w} = 0, \quad \forall c \in C,\, d \in V \tag{3.24}$$

$$\mathrm{LCA}_{c,c} = \mathrm{isLCA}_c, \quad \forall c \in C \tag{3.25}$$

$$\mathrm{RCA}_{c,c} = \mathrm{isRCA}_c \cdot \mathrm{isLCA}_c, \quad \forall c \in C \tag{3.26}$$

The objective function (3.27) of $\mathrm{OPT}_{\mathrm{es}}$ is defined as

$$\text{minimize: } \sum_{c\in C} \big(\mathrm{isRCA}_c + \mathrm{isLCA}_c \big) - \omega \cdot \sum_{x\in F} \mathrm{isSat}_x \tag{3.27}$$

With $\omega > 2 \cdot |C|$, (3.27) provides a lexicographic order in which maximizing DFG satisfaction weighs higher than minimizing the number of CAs. Meanwhile, a complete control structure, the most important objective, is already required by constraints (3.7) and (3.8). Therefore, the existence of a valid solution is strictly mandatory for $\mathrm{OPT}_{\mathrm{es}}$ to be feasible.

3.4 Problem Complexity

In this section I prove that FCAPP is NP-hard by providing a polynomial reduction of the NP-hard bin packing problem [89] to FCAPP.

Proof: Given a bin packing problem with bin size B and n items with sizes $a_1, \ldots, a_n$, I construct an FCAPP instance as follows:

- $V = \{c_1, \ldots, c_n, m, v_1, \ldots, v_n\}$,
- $C = \{c_1, \ldots, c_n\}$,
- $E = \{(v_i, m), (m, c_i) \mid i = 1, \ldots, n\}$,
- $F = \{x_1, \ldots, x_n\}$,
- $W_{x_i, v_j} = \begin{cases} 1 & i = j, \\ 0 & \text{otherwise}, \end{cases}$
- $b_{\text{cap}}(v_i, m) = a_i \quad \forall\, i = 1, \ldots, n$,
- $b_{\text{cap}}(m, c_i) = B \quad \forall\, i = 1, \ldots, n$,
- $b_{\text{flow}}(x_i) = a_i \quad \forall\, i = 1, \ldots, n$,
- $p_{\text{node}}(c) = \infty \quad \forall\, c \in C$,
- $l_{\text{cap}}(v, w) = 0 \quad \forall\, (v, w) \in E$,
- $l_{\text{flow}}(x_i) = \infty \quad \forall\, i = 1, \ldots, n$,
- $p_{\text{flow}}(x_i) = 0 \quad \forall\, i = 1, \ldots, n$,
- $b_{\text{LCA}} = b_{\text{RCA}} = p_{\text{LCA}} = p_{\text{RCA}} = 0$,
- $l_{\text{LCA}} = l_{\text{RCA}} = \infty$.

This construction can be done in polynomial time as the number of constraints and number of variables in OPT_{es} are both in $\mathcal{O}(|V|^3 \cdot |E| \cdot |F|)$. The resulting network is illustrated in Figure 3.3.

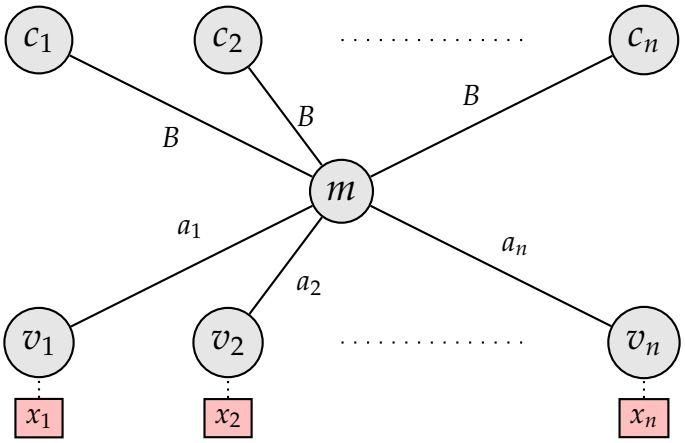


Figure 3.3: Reduction of bin packing to FCAPP

With this construction, all of FCAPP's latency and processing capacity constraints are eliminated, as guaranteeing zero processing capacity and unlimited latency is always trivially given. As for the data rate constraints, the only relevant ones are the link capacity constraints for the DFG satisfaction, since only the DFGs have non-zero data rate requirements in my construction. This

means that in this FCAPP instance, the only requirement for $c_i \in C$ to satisfy $x_j \in F$ is that link (m, c_i) provides a data rate of $b_{\text{flow}}(x_j) = a_j$. Denoting the set of DFGs satisfied by $c_i \in C$ as $\text{Sat}(c_i)$, constraint 3.14 brings

$$\sum_{x_j \in \text{Sat}(c_i)} b_{\text{flow}}(x_j) = \sum_{x_j \in \text{Sat}(c_i)} a_j \quad \leq \quad b_{\text{cap}}(m, c_i) = B.$$

Thus it holds that

$$Sat_{c_i, x_j} = 1 \Leftrightarrow \text{Bin}(a_j) = i \quad \forall\, i, j \in \{1, \ldots, n\}.$$

At last, it is important to note that the constructed network provides enough capacity to satisfy all $x \in F$ as $|C| = |F|$ and $a_i \leq B \; \forall\, i = 1, \ldots, n$. Keeping in mind that an optimal solution of FCAPP minimizes the number of used LCAs, the number of LCAs needed for this FCAPP instance is equivalent to the number of bins needed for the original bin packing instance. Also, the sets of items in every used bin are the same as the $\text{Sat}(c_i)$ of every used LCA $c_i \in C$. Hence, if FCAPP were solvable in polynomial time, the same would apply to the NP-hard bin packing problem. With this polynomial reduction, I can conclude that FCAPP is NP-hard. □

3.5 Multi-layer Greedy Heuristic

In this section, I describe my multi-layer greedy heuristic to solve FCAPP, which I call *Greedy Flow processing-aware Control Application Placement Algorithm (GreedyFCAPA)*. In accordance with the problem statement in Section 3.2, GreedyFCAPA attempts to find a complete control structure while maximizing the amount of satisfied DFGs first and minimizing the amount of used CAs next. Further, GreedyFCAPA also calculates all corresponding routing paths and consistently ensures all affected constraints. Therefore, a solution obtained by GreedyFCAPA is guaranteed to be consistent with OPT_{es}. All procedures in this section use the input parameters from Table 3.1.

Provided with an input network, GreedyFCAPA executes the CPGREEDY procedure shown in Algorithm 3.1. CPGREEDY successively adds LCAs to the network by calling FINDLCA and first focuses on having an LCA assigned to each network node. If during this stage already all potential hosts are used as LCAs, the procedure FORCECONTROL is called, which I will describe further below. When all nodes are controlled by at least one LCA and if there are still potential hosts available, CPGREEDY switches its strategy and fully focuses on DFG satisfaction. If for any reason during this second phase an added LCA did not result in additional DFGs satisfied, the LCA is removed again and *banned*, i.e. no longer considered during the remainder of the execution. The procedure ends if all DFGs are satisfied or if all potential hosts are either banned or already LCAs, i.e. if no remaining DFGs can be satisfied with the network's remaining resources. Right before terminating,

the cleanupLCAcontrols procedure is executed, which removes all LCA-to-node assignments that were established during runtime and that eventually did not serve any purpose, i.e. the LCA does not satisfy any DFG originating from the node and the node is assigned to more than one LCA.

Algorithm 3.1 CPgreedy()

```
option = "neighbors"
while |V| − |V_controlled| > 0 do
  if |LCAs| = |C| then
    forceControl()
    break // no more potential hosts available
  findLCA(option)
if |LCAs| < |C| then
  option = "flows"
  while |F| − |F_satisfied| > 0 do
    Sat_last = |F_satisfied|
    findLCA(option)
    if Sat_last = |F_satisfied| then // no additional DFGs could be satisfied
      C_banned.append(LCAs[-1])
      removeLastAddedLCA()
    if |LCAs| = |C| - |C_banned| then
      break // no more non-banned potential hosts available
cleanupLCAcontrols()
```

The findLCA procedure presented in Algorithm 3.2 determines which potential host is going to be added as an LCA next, according to the *option* set by CPgreedy (see Algorithm 3.1 and Algorithm 3.4). For this purpose, it acquires the best candidates via getLCAcandidates(*option*). But before adding the best candidate as an LCA right away, findLCA first tries to find an RCA for it, using the findRCA procedure, as an LCA that cannot be assigned to an RCA would violate the integrity of the two-tier control hierarchy. If an RCA can be found, the candidate is confirmed as a new LCA and is assigned to the RCA. Finally, Algorithm 3.5 is called which assigns nodes and DFGs to the recently added LCA.

Algorithm 3.2 findLCA(*option*)

```
candidates = getLCAcandidates(option)
for v in candidates do
  c = findRCA(v)
  if c is not None then
    addNewLCA(v)
    break // next LCA determined
```

The findRCA procedure shown in Algorithm 3.3 tries to assign an RCA to a given LCA v. First, it checks the already available RCAs, starting with the closest one. The checkRCAcontrol procedure verifies if assigning an RCA to a node complies with all relevant network constraints (3.14, 3.16 – 3.18)

according to the given routing path. If an existing RCA can be assigned to v, the added RCA assignment is returned, otherwise a new RCA has to be added to the network using a similar strategy as before, except now considering all potential hosts that are not yet RCAs. For the special case of the first RCA in the network, I choose the potential host with the least distance (in number of hops) to all other potential hosts (lowest node ID in case of a tie) in order to benefit all future RCA-to-LCA assignments.

Algorithm 3.3 findRCA(v)

```
paths = {shortestPath(source=c, target=v) for c in RCAs}
sort paths by lengths
for p in paths do
  if checkRCAcontrol(p) then
    return addRCAcontrol(c, v)
paths = {shortestPath(source=c, target=v) for c in C - RCAs} // a new
RCA needs to be added
if |RCAs| = 0 then
  sort paths by average length to all potential hosts // try to place the first
  RCA centrally in the network
else
  sort paths by lengths
for p in paths do
  if checkRCAcontrol(p) then
    return addRCAcontrol(c, v)
return None // failed finding an RCA for v
```

The critical LCA candidate selection is done by the getLCAcandidates procedure which is shown in Algorithm 3.4. getLCAcandidates considers all potential hosts that are not yet banned or used as an LCA. But if possible, the

Algorithm 3.4 getLCAcandidates(*option*)

```
candidates = C - (LCAs + C_banned)
if candidates - RCAs ≠ ∅ then
  candidates = candidates - RCAs // avoid RCAs if possible
if option = "neighbors" then
  sort candidates descending by uncontrolled nodes in {v} ∪ neighbors(v)
  if best value = 0 then // no node with uncontrolled neighbors
    candidates = getLCAcandidates("isolated nodes")
else if option = "isolated nodes" then
  sort candidates by shortest distance to an uncontrolled node
else if option = "flows" then
  sort candidates by highest amount of unsatisfied DFGs connected to v
  if best value = 0 then // no node with unsatisfied DFGs
    candidates = getLCAcandidates("isolated flows")
else if option = "isolated flows" then
  sort candidates by shortest distance to a node with unsatisfied DFGs
```

procedure also excludes any RCA to leave them with resources for coordinating future LCAs. To determine the best LCA candidates, getLCAcandidates then uses the options *neighbors* or $flows$, depending on what has been specified by CPgreedy. But for corner cases, getLCAcandidates is allowed to switch to one of the more specific options *isolated nodes* or *isolated flows* in case the metrics used by *neighbors* or $flows$ do not provide a sufficient result for sorting the candidates.

Algorithm 3.5 addNewLCA(v)

```
paths = {shortestPath(source=v, target=i) for i in V}
F_pot(v) = {}
if |V| − |V_controlled| > 0 then // no valid solution yet
  sort paths by paths to uncontrolled nodes first, path length next
else // valid solution already found, now focus on DFG satisfaction
  sort paths by paths to nodes with unsatisfied DFGs first, path length next
while (|paths| > 0 or |F_pot(v)| > 0) and (|V| − |V_controlled| > 0 or |F| −
|F_satisfied| > 0) do
  if |F_pot(v)| > 0 and (|V| − |V_controlled| = 0 or |V_new| ≥ |V|/|C|) then
    f = getNextDFG(F_pot(v))
    if checkDFGsat(v, f) = True then
      addDFGsat(v, f)
    F_pot(v).remove(f)
  else
    if |paths| = 0 then
      break // no more paths to look at
    p = getNextPath(paths)
    if checkLCAcontrol(p) = True then
      addLCAcontrol(p)
      updatePotentialDFGs(v)
    else
      break // no more resources left at LCA v
```

As stated before, the addNewLCA procedure (Algorithm 3.5) is responsible for assigning nodes and DFGs to a new LCA v. To this end, it calculates the shortest paths from v to all other nodes in the network. Then, it prioritizes the nodes to be assigned to the new LCA by sorting the paths, depending on whether a network already has a complete control structure or not. In the former case, addNewLCA considers paths to uncontrolled nodes first, while in the latter case, the paths are sorted according to the amount of unsatisfied DFGs originating from the target nodes.

After the prioritization is done, nodes are assigned to be controlled by v. To ensure that no model constraint is violated in the process, the procedure checkLCAcontrol verifies if assigning a node to v complies with all relevant network constraints (3.14, 3.16 – 3.18) each time before assigning it to v. When a node is assigned to v, the updatePotentialDFGs function is executed to update the list of the *potential DFGs* for v, i.e. the DFGs that are only entering

the network through nodes that v already controls and thus can be satisfied by v if sufficient resources are available. However, GreedyFCAPA has to hold back before assigning potential DFGs to v. The reason for this is as follows: Assuming a network with many data flows, an LCA would have a lot of potential DFGs after controlling only a few nodes, run out of resources by satisfying them very quickly and hence a complete control structure might not be obtained. Hence, DFGs may only be satisfied by an LCA once it controls at least $\frac{|V|}{|C|}$ nodes that had previously been uncontrolled (represented by V_{new} in Algorithm 3.5) or if there are no more uncontrolled nodes in the network. The choice for $\frac{|V|}{|C|}$ is based on the fact that *on average*, an LCA needs to control at least $\frac{|V|}{|C|}$ nodes to obtain a complete control structure. In addition, this choice will be further assessed in Chapter 4 and Chapter 5.

Eventually, when the conditions are met so that v is allowed to satisfy DFGs, GreedyFCAPA successively tries to assign DFGs to v. The order in which the DFGs are assigned is determined based on requested processing capacity. Two sorting strategies suggest themselves:

- Least Demanding First (LDF)
- Most Demanding First (MDF)

Intuitively, both strategies appear favorable compared to each other depending on the load situation in the network. With exhausted network resources, LDF is expected to perform better because the resulting number of satisfied DFGs should be higher compared to MDF. With sufficient network resources available, though, one would expect that MDF results in fewer LCAs to be used, since assigning less demanding DFGs to LCAs that already satisfy several DFGs is naturally more promising than assigning highly demanding ones. However, the evaluation in Section 3.6 will show whether the practical results really meet this intuitive elaboration or not.

Coming back to assigning DFGs in Algorithm 3.5, again all relevant constraints (3.14, 3.16 – 3.18) are checked using the checkDFGsat procedure before confirming that v henceforth satisfies a certain DFG. In any case, the DFG is then removed from the list of potential DFGs for v, as it is either now satisfied by v or as it cannot be satisfied by v. addNewLCA terminates when no more nodes and no more DFGs can be assigned to v.

Still, the aforementioned requirement for new LCAs to control $\frac{|V|}{|C|}$ nodes that had previously been uncontrolled before satisfying DFGs does not strictly guarantee that all nodes will eventually be controlled. Therefore, as a last resort, a complete control structure is ultimately enforced by the earlier mentioned procedure forceControl (Algorithm 3.6).

forceControl basically forces each eventually uncontrolled node v to be controlled by its closest LCA c. To do this, the procedure successively removes the DFG with the most required processing capacity from c until either c can control v or c could control v if the routing path had additional capacity.

The latter is indicated by addLCAcontrol, which returns different values depending on what constraint makes the LCA-to-node assignment impossible. In this case, additional DFGs that currently occupy capacity of the links of the path are removed, so that the control from c to v can finally be established. In the end, forceControl attempts to reestablish the removed DFG-to-LCA assignments if possible.

Algorithm 3.6 forceControl()

```
F_removed = {}
for v in V − V_controlled do
  c = getClosestLCA(v), path = shortestPath(source=c, target=v)
  sort F_sat(c) descending by required processing capacity
  for f in F_sat(c) do
    F_removed.append((c, f)), removeDFGsat(f)
    if checkLCAcontrol(p) = True then
      addLCAcontrol(p), break
    else if checkLCAcontrol(p) = "processing ok, only link capacity not"
    then
      for (u, w) in path do
        F_(u,w) = {f ∈ F, f is routed over (u, w)}
        sort F_(u,w) (DFGs satisfied by c) by most required data rate first
        for x in F_(u,w) do
          F_removed.append((LCA(x), x)), removeDFGsat(x)
          if b_rem(u, w) ≤ b_LCA then
            break
          addLCAcontrol(p), break
for c, f in F_removed do
  if checkDFGsat(c, f) = True then
    addDFGsat(c, f)
```

To provide a comprising overview, Figure 3.4 summarizes the key aspects of GreedyFCAPA in a flow chart.

3.6 Evaluation

The evaluation of this chapter consists of two parts: at first, I evaluate OPT_{es} against GreedyFCAPA, which is only feasible in very small scenarios owing to the runtime of OPT_{es}. Next, I provide performance results using varying parameters for GreedyFCAPA in larger, real-world size scenarios.

All evaluations are executed on Intel® Xeon® E5-2695 v3 CPUs running at 2.30 GHz. I have implemented OPT_{es} using the Pyomo package for optimization modeling in Python [90] and solved it with Gurobi [91] running in single-threaded mode. GreedyFCAPA has been implemented using Python. All plots contain confidence intervals at a 95% confidence level.

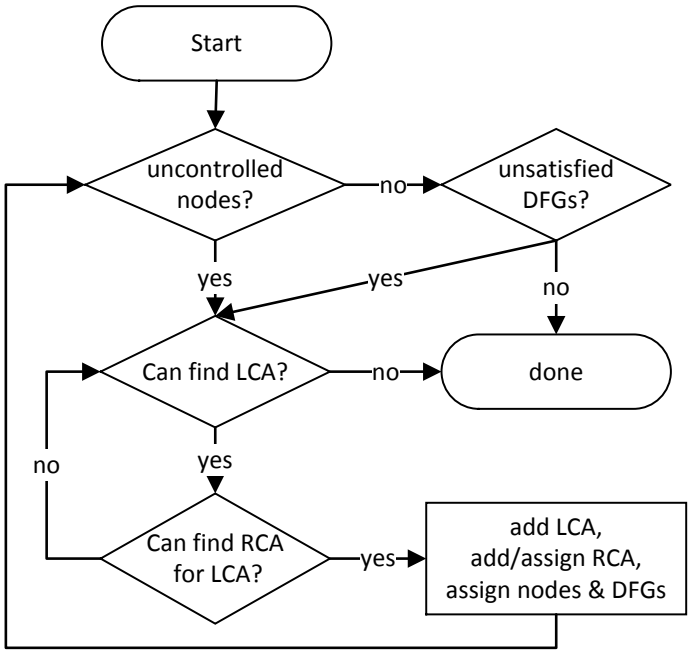


Figure 3.4: GreedyFCAPA flow chart

3.6.1 Evaluation Scenario

To obtain initial performance results, I have created a first, rather generic, evaluation scenario. I will later provide an alternative topology type in Chapter 5 and focus on a more realistic DFG scenario in Chapter 9.

For all instances of this evaluation, the nodes are placed on a regular grid with a mean inter-BS distance of $\bar{s} = 1000\,\text{m}$, which corresponds to an urban scenario [92], and are then shifted in both x and y direction using normally distributed random variables with zero mean and standard deviation $\frac{\bar{s}}{8}$. The backhaul links are generated as *mesh topology*: two nodes $v, w \in V$ are connected if $\text{dist}(v, w) \leq 1.5 \cdot \bar{s}$. The chosen factor 1.5 has consistently produced fully connected but not unrealistically dense topologies. For illustration, Figure 3.5 shows two exemplary mesh topologies used in Section 3.6.3; one with 6×6 nodes and one with 10×10 nodes.

Each node becomes a potential host with a probability of P_C and is then assigned with a processing power of $p_{\text{node}} = 200$ GFLOPS. All links are assigned the same fixed capacity of 2.5 Gbit/s and the latency for each link is determined by its length multiplied by 1.45 and divided by the speed of light, assuming an optical backhaul network [93]. The LCA and RCA parameters from Table 3.1 are chosen as follows: $b_{\text{LCA}} = b_{\text{RCA}} = 100$ Kbit/s, $l_{\text{LCA}} = 1$ ms, $l_{\text{RCA}} = 10$ ms, $p_{\text{LCA}} = p_{\text{RCA}} = 10^6$ operations per control packet.

To assign DFGs to nodes, I assume mobile user equipment as origin and/or destination of the DFGs for this evaluation scenario. I use the GreenTouch connectivity model [92] and each DFG is connected to up to three nodes with

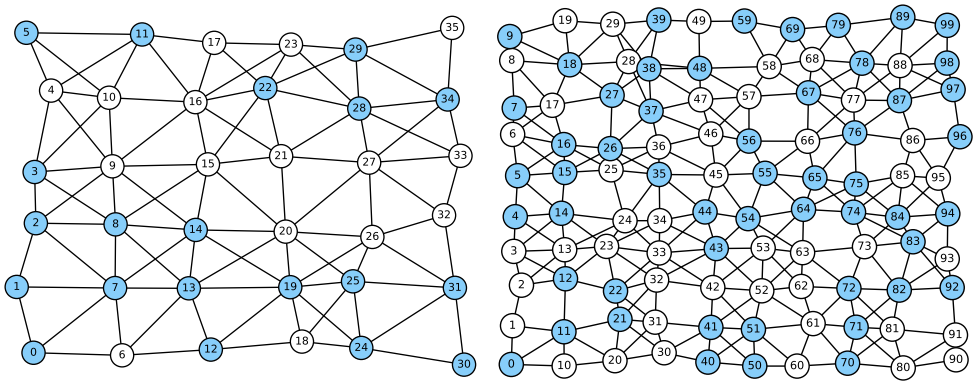


Figure 3.5: Exemplary mesh topologies (with potential hosts highlighted)

the best connectivity until a Signal to Interference plus Noise Ratio (SINR) threshold of 0.0 dB is reached. A DFG belongs to one out of three types, as shown in Table 3.3. I have extrapolated the data for these types using the cellular traffic data for 2016 from [3]. As a base for choosing the $p_{\text{flow}}(x)$ parameters, I introduce a randomized factor $\text{op}(x)$ for each DFG, describing the operational overhead that arises during data processing. For all ranges listed in Table 3.3, the values have been chosen uniformly at random.

Table 3.3: Evaluation scenario: DFG types

type	probability	b_{flow}	l_{flow}	$\text{op}(x)$
audio	0.3	0.5 to 1 Mbit/s	10 ms	$1 \cdot 10^6$ to $2 \cdot 10^6$
video	0.6	1 to 5 Mbit/s	10 ms	$5 \cdot 10^6$ to $1 \cdot 10^7$
other	0.1	1 to 20 Mbit/s	50 ms	$1 \cdot 10^6$ to $1 \cdot 10^8$

Based on this, the processing capacity requested by DFG x is determined by

$$p_{\text{flow}}(x) = \text{op}(x) \cdot \sum_{v \in V} W_{f,v}.$$

To give an impression of the number of data flows per DFG produced by this evaluation scenario, I have generated 1000 DFGs for each network used in Section 3.6.3. Rounded to one decimal place, 71,4% of the DFGs had one data flow, 22,6% had two data flows and 6% had three data flows.

3.6.2 OPT_{es} vs. GreedyFCAPA

To compare the results of OPT_{es} and GreedyFCAPA (MDF and LDF), I have generated instances with 4 and with 9 nodes with $P_C = 1.0$ and multiples of 10 DFGs. These scenarios are big enough to see the important effects and small enough to run in reasonable time.

The time limit for OPT_{es} has been set to a solving time of one hour. To enhance the execution of OPT_{es}, I first executed GreedyFCAPA for all instances and then added additional constraints to OPT_{es}:

$$\sum_{c \in C} \text{isRCA}_c \leq G_{\text{RCA}}, \tag{3.28}$$

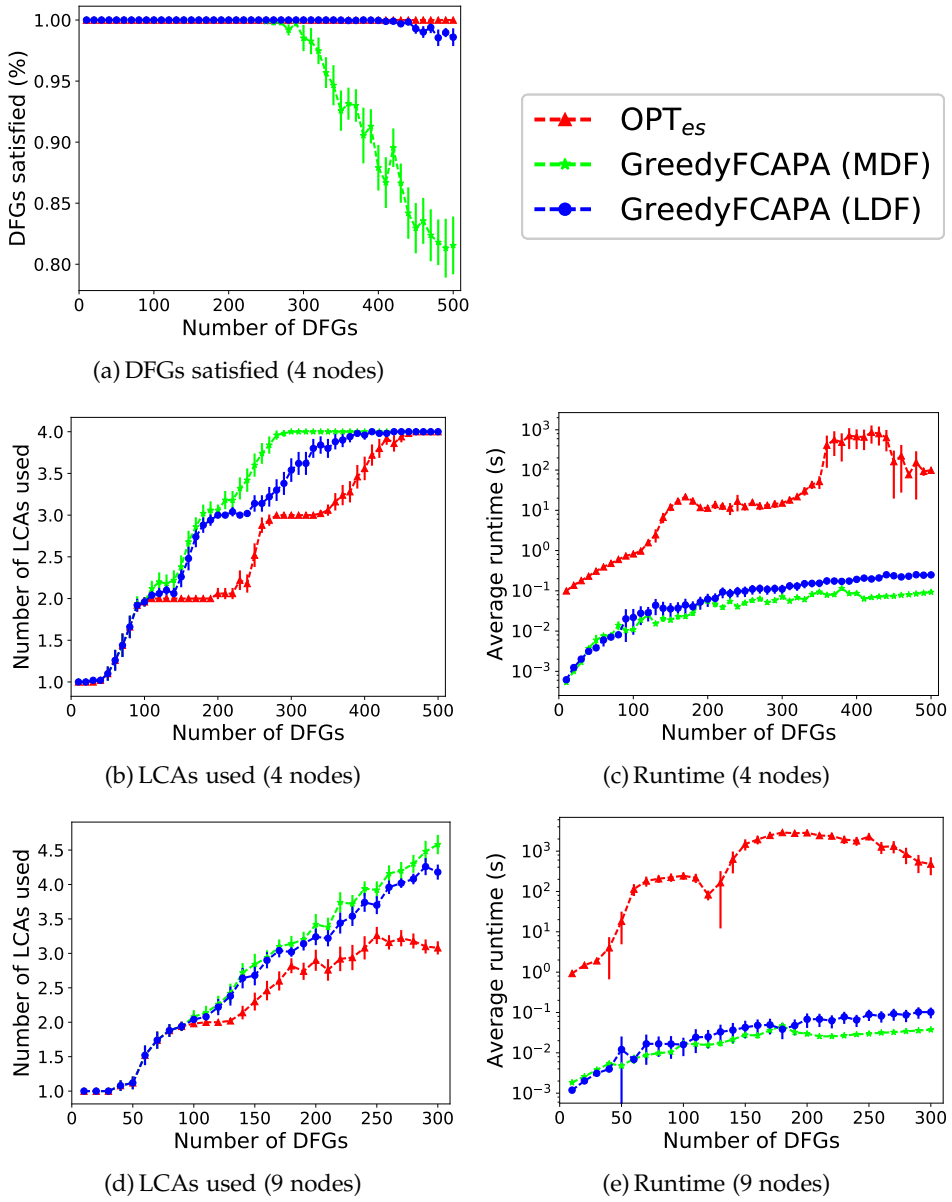
$$\sum_{c \in C} \text{isLCA}_c \leq G_{\text{LCA}}, \tag{3.29}$$

$$\sum_{x \in F} \text{isSat}_x \leq G_{\text{Sat}}, \tag{3.30}$$

where $G_{\text{RCA}}, G_{\text{LCA}}$ and G_{Sat} are the numbers of RCAs/LCAs used and DFGs satisfied by GreedyFCAPA (with MDF setting) for the corresponding instance. The benefit of this is twofold: on the one hand, the solution space for OPT_{es} to search through is significantly reduced; on the other hand, it is prevented that OPT_{es} returns a solution inferior to the one of GreedyFCAPA when stopping at the time limit.

All relevant results of this evaluation part are illustrated in Figure 3.6. A valid solution has been found for all instances; consistently only one RCA was used and for the 9-node instances all DFGs have been satisfied. For the 4-node instances, however, Figure 3.6a reveals that GreedyFCAPA leaves DFGs unsatisfied starting from around 250 DFGs (MDF) and 400 DFGs (LDF), while OPT_{es} is constistently able to satisfy all DFGs up to the limit for this evaluation part of 500 DFGs.

As can be seen in Figure 3.6b and Figure 3.6d, GreedyFCAPA performs just as well as OPT_{es} for few DFGs before performing visibly worse than OPT_{es} once two LCAs are required for solving an instance. It can also be seen that GreedyFCAPA with LDF strategy uses fewer LCAs than GreedyFCAPA with MDF strategy – contrary to the intuitive prediction from Section 3.5. A possible explanation for this could be that the LDF strategy causes lots of highly demanding DFGs not to be satisfied at first, but to be later satisfied together by an additional LCA, which would be beneficial in synergy with the equal-share processing scheme employed by FCAPP. But instead of going into more detail about this here, I refer to Section 5.1, where I elaborate in detail on the downsides of equal-share scheduling. At last, Figure 3.6c and Figure 3.6e show the runtime results. It can be seen that GreedyFCAPA with LDF strategy requires slightly more execution time than GreedyFCAPA with MDF strategy. This is in line with the previous explanation, since this could be caused by highly demanding DFGs being more often rejected and thus being more often reconsidered by different LCAs (within addNewLCA from Algorithm 3.5). Independend of the DFG assignment strategy, however, GreedyFCAPA operates about three to five orders of magnitude faster than OPT_{es}.



(a) DFGs satisfied (4 nodes)

(b) LCAs used (4 nodes)

(c) Runtime (4 nodes)

(d) LCAs used (9 nodes)

(e) Runtime (9 nodes)

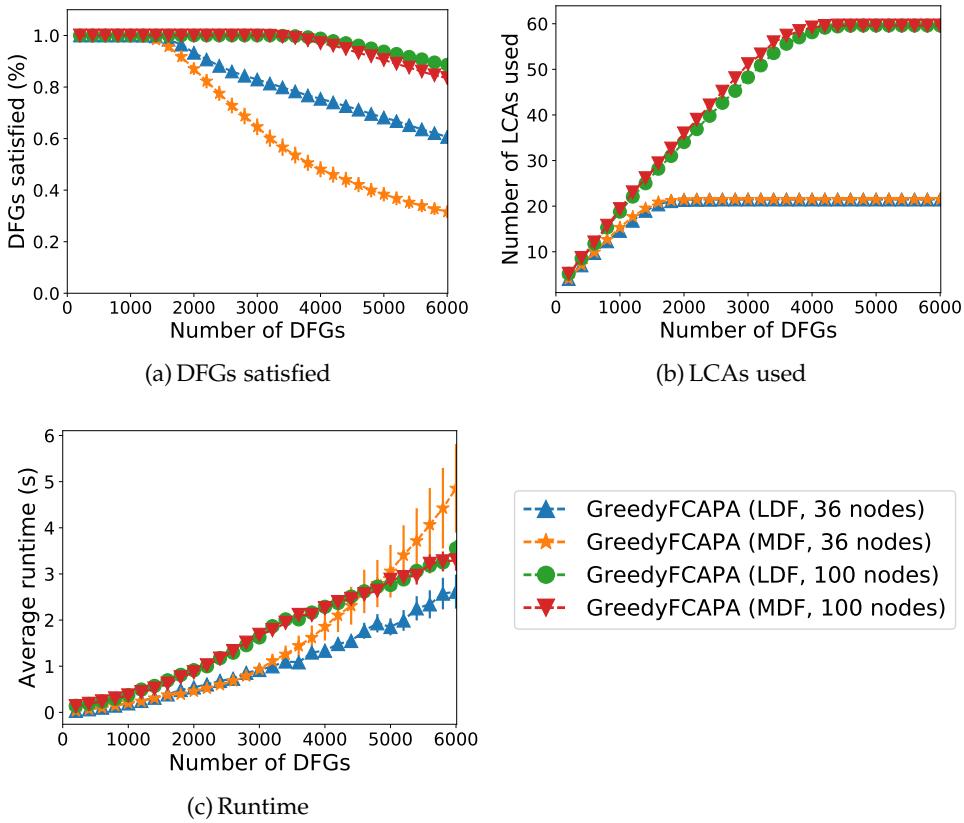
Figure 3.6: Evaluation: OPT_{es} vs. GreedyFCAPA

3.6.3 GreedyFCAPA in Larger Scenarios

To evaluate GreedyFCAPA for larger scenarios, I have generated instances with 36 and 100 nodes, $P_C = 0.6$ and multiples of 200 DFGs. The results can be seen in Figure 3.7.

Again, all instances have obtained a valid solution and consistently only one RCA was used for each instance. However, as can be seen in Figure 3.7a, not all DFGs could always be satisfied; the 36-node and 100-node networks run out of resources to satisfy all DFGs around 1400 and 4400 DFGs. In contrast to the previous evaluation part, the granularity of this evaluation part does

not feature any differences between LDF and MDF for the point where not all DFGs can be satisfied, but GreedyFCAPA with LDF strategy is able to satisfy more DFGs after this point. The percentage of DFGs drops down to around 30% (MDF) and 60% (LDF), respectively, for the 36-node networks with 6000 DFGs. In particular, this corresponds to around 1800 and 3600 DFGs, more than the turning point of around 1400 DFGs, which shows that GreedyFCAPA, especially with LDF strategy, adapts very well to overloaded networks and uses the bigger range of available DFGs to satisfy more of them with the available resources.



(a) DFGs satisfied

(b) LCAs used

(c) Runtime

Figure 3.7: Evaluation: GreedyFCAPA in larger scenarios

Next, Figure 3.7b shows again that the number of used LCAs only slightly depends on the number of nodes in the network but more on the number of DFGs, at least as long as sufficient potential hosts are available. It can be seen that even before the resources of the 36-node networks are exhausted, the 100-node networks need slightly more LCAs. Again, the LDF strategy requires fewer LCAs but only marginally. Last but not least, Figure 3.7c illustrates the runtime for the larger scenarios. It can be observed that the runtime of GreedyFCAPA naturally depends on the amount of nodes in the network, which results in a larger solution space to be considered. But interestingly, the runtime of GreedyFCAPA with MDF strategy increases faster in the 36-

node networks with more DFGs once the network resources are exhausted and even exceeds the one of the 100-node networks with 4600 DFGs. This can be explained by more failed attempts to satisfy DFGs, so that the same DFGs are considered more often when adding different LCAs. Interestingly, GreedyFCAPA with LDF strategy runs faster than GreedyFCAPA with MDF strategy – contrary to the previous evaluation part. An explanation for this is that the sheer amount of not satisfied and reconsidered DFGs in these large instances outweighs the effect that I gave as an explanation beforehand.

In total, it can be said that according to the practical results obtained from this evaluation, the LDF strategy clearly outperforms the MDF strategy in all relevant aspects. As a result, the LDF strategy will be used as a default in the following.

3.7 Observations

In this chapter I have given an initial problem statement for FCAPP using equal-share processing scheduling and I have used this to formulate an MIQCP as a reference model to solve the problem. Based on the created MIQCP (OPT_{es}), I have then proven FCAPP to be NP-hard. To cope with the problem's complexity, I have implemented and evaluated a fast heuristic solution, GreedyFCAPA.

The evaluation reveals that GreedyFCAPA is able to solve FCAPP with a decent solution quality while providing a runtime that might make it suitable for networks with frequently changing network load as outlined in Section 1.2. In particular, in the overall context of my thesis, I have made a first step to show that the idea of performing flow processing-aware control application placement within such networks is indeed feasible. However, the comparison between OPT_{es} and GreedyFCAPA also clearly revealed that there is room for improvement, which will be my leverage point for the subsequent chapter.

4

Assessing Genetic Algorithms for Flow Processing-aware Control Application Placement

In the previous chapter, I have introduced FCAPP and presented an optimization model (OPT_{es}) and a fast heuristic solution (GreedyFCAPA) to provide reasonably good results in a practical amount of time. However, the question remained open whether or not there is an alternate algorithmic approach that gives significantly better results than GreedyFCAPA within a reasonable amount of time. To tackle this question, I decided to explore the concept of Genetic Algorithms (GAs) which I already introduced in Section 2.2.1. As elaborated there, GAs have already been successfully applied to other problems related to FCAPP, such as to VNE [54].

Together with Swante Scholz, I have developed three GA approaches, one pure GA approach and two hybrid GA approaches, which I will describe and evaluate in this chapter. This work has been conducted over the course of Mr. Scholz's bachelor thesis [94] under my supervision. First, I elaborate on the fitness function and selection mechanisms in Section 4.1, which are common to all three approaches. Next, I present the three GA approaches in Sections 4.2, 4.3 and 4.4. Finally, I provide extensive evaluation results in Section 4.5, including parameter evaluations of the three GAs and a comparison with GreedyFCAPA.

4.1 Fitness Function and Selection

While different GA approaches for FCAPP need different genetic operators and representations, the underlying fitness evaluation for comparing the quality of individuals and the approaches for parent and survivor selection can remain the same as the optimization goals of each approach are identical. Corresponding to the FCAPP objectives in Section 3.2, the following aspects are important, in decreasing order of relevance:

1. Minimizing the number of uncontrolled nodes U_{nodes} and uncoordinated LCAs U_{LCA}.
2. Minimizing the number of unsatisfied DFGs U_{DFG}.
3. Minimizing the number of used LCAs and RCAs.

The fitness of each individual can thus be expressed as:

$$\big(U_{\text{nodes}} + U_{\text{LCA}}, U_{\text{DFG}}, |\text{LCAs}| + |\text{RCAs}|\big) \tag{4.1}$$

To compare the fitness of individuals with each other, it is necessary to compute a real-valued *fitness score* from each tuple:

$$f\big((x,y,z)\big) := \omega_1 \cdot x + \omega_2 \cdot y + z, \tag{4.2}$$

with $\omega_2 > 2 \cdot |C|$ and $\omega_1 > \omega_2 \cdot |F|$ to lexicographically order the optimization goals.

Parent selection is done via *tournament selection* [95]. In tournament selection, in order to select an individual for reproduction, two or more individuals of the current generation are selected at random. The best of them is then used for reproduction. This process is repeated until the required number of children have been created.

Regarding survivor selection, all GAs choose the μ individuals with the best fitness values from the $\mu + \lambda$ individuals of the current generation and its offspring. Initially, more complex approaches for survivor selection, e.g. tournament selection, were also considered, but first test evaluations revealed that the aforementioned simpler choice performed just as well in practice.

4.2 Approach 1: Pure Genetic Algorithm

The first GA approach is a pure GA that gives maximal freedom over the solution, resulting in a very powerful and complete GA.

4.2.1 Representation

The DNA of each individual is a tuple of three arrays:

$$\big(\text{RCAs of LCAs}, \text{LCAs of nodes}, \text{LCAs for DFGs}\big)$$

The first array identifies the RCAs for all the $c \in C$ (or -1 if c is not an LCA or an uncoordinated LCA). The second array contains one LCA for each node $v \in V$. While each node might be controlled by multiple LCAs, only one has to be represented in this array since one LCA per node already assures a complete control structure. The third array shows the LCAs satisfying the DFGs (or -1 if the DFG is unsatisfied).

It is important to note that each array entry just expresses that during fitness evaluation, the algorithm will *try* to fulfill the assignment with all the required coordination and control paths. But if there are not enough resources left at an RCA or LCA to fulfill all tasks (node control and DFG satisfaction) assigned to it, the assignment will fail, leaving unrealized tasks that decrease the fitness value of the individual.

4.2.2 Crossover

A very intuitive approach to design the crossover operator is preferring more common parent genes over genes that are less common, thereby encouraging a genetic drift towards fewer CAs being used. Algorithm 4.1 shows how parent genes are favored based on their relative number of occurrences.

Algorithm 4.1 Weighed crossover operator (GA1)

```
function Crossover(ind1, ind2)
  child = new Individual
  c_RCA, c_LCA = new Counter, new Counter
  for (attr, count) in [(RCAs, c_RCA), (LCAs, c_LCA), (Sats, c_LCA)] do
    a, b = ind1.attr, ind2.attr
    for i in {0,...,len(a)-1} do
      count[a[i]] += 1, count[b[i]] += 1
  for (attr, count) in [(RCAs, c_RCA), (LCAs, c_LCA), (Sats, c_LCA)] do
    a, b, c = ind1.attr, ind2.attr, child.attr
    for i in {0,...,len(a)-1} do
      prob = count[a[i]] / (count[a[i]] + count[b[i]])
      if randomFromUnitInterval() < prob then
        c[i] = a[i]
      else
        c[i] = b[i]
  return child
```

4.2.3 Mutation

As shown in Algorithm 4.2, the mutation operator uses three different functions that are chosen depending on the current fitness of the individual, each corresponding to one entry of the fitness tuple. First, mutateControlStructure randomly changes some assignments from the first two arrays of the DNA, thereby trying to reduce the number of control violations. Analogously, mutateDFGAssignment is applied to reduce the number of unsatisfied DFGs if an individual has no control violations (first fitness component) but still has unsatisfied DFGs (second fitness component). At last, decreaseNumberOfCAs tries to reduce the number of used LCAs and RCAs by deactivating one of the least used CAs and randomly reassigning all the tasks it was responsible for to the other CAs still in use. As control structure and DFG satisfaction are

of higher priority than reducing the number of LCAs and RCAs, this function is only used once the control structure is complete and all DFGs have been satisfied.

Algorithm 4.2 Mutation operator (GA1)

```
function Mutate(ind):
  child = deepCopy(ind)
  if child.fitness.controlStructureViolations > 0 then
    mutateControlStructure(child)
  else if child.fitness.unsatisfiedDFGs > 0 then
    mutateDFGassignment(child)
  else
    decreaseNumberOfCAs(child)
  return child
```

```
function mutateControlStructure(child)
  for a in [child.RCAs, child.LCAs] do
    for i in {0,...,len(a)-1} do
      if randomFromUnitInterval() < α_m then
        a[i] = randomElementOf(C)
```

```
function mutateDFGassignment(child)
  for i in {0,...,len(child.Sats)-1} do
    if randomFromUnitInterval() < α_m then
      child.Sats[i] = randomElementOf(C)
```

```
function decreaseNumberOfCAs(child)
  CAs = child.RCAs + child.LCAs + child.Sats
  c = CA with lowest number of occurrences in CAs
  remove c from CAs
  for a in [self.RCAs, self.LCAs, self.Sats] do
    for i in {0,...,len(a)-1} do
      if a[i] == c then
        a[i] = pick uniformly at random from CAs
```

In the functions above, $\alpha_m \in [0, 1]$ is a parameter specifying the intensity of the mutation operator.

4.3 Approach 2: Hybrid GA based on Post Processing

While the pure GA approach is very powerful in theory, it comes at the cost of searching an unnecessarily large search space. Its representation does not, for example, take into account that DFGs should generally be satisfied by nearby CAs. Therefore, I describe a hybrid GA in this section that uses a less complex representation and is combined with a heuristic that takes such matters into account.

4.3.1 Representation and Fitness Evaluation

To represent the individuals for this hybrid GA, two binary arrays are used to specify for each potential host if it is to be used as RCA and as LCA. For example, an individual for an instance with $|C| = 4$ could be represented as

$$\big((1,0,0,0),(0,0,1,1)\big),$$

meaning that the first potential host is selected as an RCA and the third and fourth are selected as LCAs, while the second host is not used at all.

But as already described in Section 4.2.1, there is no guarantee that such an assignment performs well. To evaluate the fitness of a given individual, a heuristic post-processing step is used that first tries to establish complete RCA-to-LCAs coordination and LCAs-to-node control and then tries to maximize the number of satisfied DFGs based on this assignment.

RCA-to-LCA coordination: For each LCA, the shortest possible coordination path to an RCA is used. The LCAs with the shortest path are coordinated first, LCAs with the same the distance are ordered randomly.

LCA-to-node control: Each node gets controlled by its shortest possible path to an LCA. The nodes with the shortest path are controlled first, nodes with the same distance are ordered randomly.

DFG satisfaction: DFGs are satisfied by the LCA closest to them, using shortest possible paths and starting with the least demanding DFG, in terms of processing requirements, first (LDF). If two DFGs have the same processing requirements, they are ordered randomly.
If a node that has previously been controlled by an LCA to ensure a complete control structure becomes controlled by another LCA to satisfy a DFG (which then of course has to allocate the resources to control the node and to satisfy the DFG), then that first, now redundant control assignment is removed, thereby freeing processing resources that can possibly be used to satisfy other DFGs.

Of course, for each of these assignments, the responsible RCA or LCA has to allocate the required resources accordingly. The implementation of this iterative process was mostly done by reusing the procedures of GreedyFCAPA (Section 3.5). Therefore, I have omitted a more detailed representation.

4.3.2 Crossover

The fitness evaluation for this representation, which I described in the previous section, is rather sophisticated. But in return, the genetic operators can be kept rather simple. Our chosen crossover procedure is described in Algorithm 4.3. It creates a new child by choosing one of the binary operators *"and"* or *"or"* uniformly at random.

Algorithm 4.3 Crossover operator (GA2)

```
function Crossover(ind1, ind2)
  child = new Individual
  if randomFromUnitInterval() < 0.5 then
    OP = or
  else
    OP = and
  for i in {0,...,len(ind1.RCAs)-1} do
    child.RCAs[i] = ind1.RCAs[i] OP ind2.RCAs[i]
  for i in {0,...,len(ind1.LCAs)-1} do
    child.LCAs[i] = ind1.LCAs[i] OP ind2.LCAs[i]
  return child
```

4.3.3 Mutation

Mutation is either performed on the RCAs or on the LCAs, with the LCAs being preferred, as LCA selection is more crucial in most problem instances. Each bit of the chosen attribute is then flipped with a constant probability of α_m. Details can be seen in Algorithm 4.4.

Algorithm 4.4 Mutation operator (GA2)

```
function Mutate(ind)
  child = deepCopy(ind)
  if randomFromUnitInterval() < 0.3 then
    a = child.RCAs
  else
    a = child.LCAs
  for i in {0,...,len(a)-1} do
    if randomFromUnitInterval() < α_m then
      a[i] = not a[i]
  return child
```

4.3.4 Variation with Extended DNA

In Section 4.3.1, I have described that the post-processing heuristic used to determine the fitness of an individual uses LDF ordering for assigning DFGs to LCAs by default. This was chosen due to the corresponding evaluation for GreedyFCAPA in the previous chapter, where LDF ordering clearly dominated MDF ordering. But since GAs are entirely different algorithms, it cannot be assumed that the same holds for this GA.

Instead of predefining a heuristic DFG processing order, however, it is also possible to specify the DFG processing order within the genetic representation itself, simply by adding a third array to each individual's DNA, consisting of a permutation of all DFGs. To better assess the impact of the processing order

on the solution quality, this variation has also been implemented. Crossover on that specific part of each individual is carried out by *alternating position crossover* [45]. During alternating position crossover, starting from an empty offspring, both parents' genes are traversed alternately and appended to the offspring permutation list. Genes already present are skipped. Mutation is performed using shuffle mutation, i.e. each gene on an individual is swapped with a random other gene with a probability of α_m.

4.4 Approach 3: Hybrid GA based on GreedyFCAPA

In addition to the hybrid GA presented in the previous section, it is also possible to design a hybrid GA based on GreedyFCAPA. There are two aspects, in particular, where suboptimal decisions might be made because of the deterministic nature of GreedyFCAPA (see Section 3.5):

1. The ordering of the LCA candidates (Algorithm 3.4) and
2. the assignment of nodes and DFGs to a chosen LCA (Algorithm 3.5).

If one or both of these aspects were to be managed by a more flexible GA, the resulting hybrid GA might give significantly better solutions, albeit at the expense of additional runtime. This idea has been realized by designing a hybrid GA, consisting of GreedyFCAPA with the exception that the order of the LCA candidates is governed by a separate GA.

4.4.1 Representation and Fitness Evaluation

Similar to Section 4.3.4, the obvious choice for representing the order of the LCA candidates is a permutation of the $|C|$ potential hosts. Regarding the initialization of individuals, it would be possible to introduce at least one individual that contains the potential hosts in the same order as GreedyFCAPA would process them, making sure that the GA's end result is at least as good as the one produced by GreedyFCAPA. However, this might lead to premature convergence as that individual would take over the population quite quickly, diminishing its diversity. Therefore, all individuals are initialized with a random permutation instead.

The fitness evaluation is then done by running GreedyFCAPA, but instead of calling the routine for the LCA candidate selection (Algorithm 3.4), the list of candidates provided by the GA's representation is used.

4.4.2 Crossover and Mutation

As the representation consists solely of a permutation, the crossover and mutation operators can be defined by reusing two off-the-shelf operators: alternating position crossover (Section 4.3.4) and *displacement mutation*, which takes a random slice of a list representing a permutation and moves it to a random position in the sequence [46].

4.4.3 Variation with Extended DNA

So far, the hybrid GA considers only one of the two possible improvements that I identified above. But the assignment of DFGs and nodes to $c \in C$, once c has been chosen as an LCA, could also be managed by a GA. As described in Section 3.5, GreedyFCAPA only starts assigning DFGs to c once either all nodes are already controlled or it controls more than $n_{\min} = \frac{|V|}{|C|}$ nodes that were previously uncontrolled. The reason that I presented in Section 3.5 was as follows: In a network with many data flows, an LCA would have a lot of potential DFGs after controlling only a few nodes, run out of resources by satisfying them very quickly and hence a complete control structure would perhaps not be obtained. The choice for $\frac{|V|}{|C|}$ is was made because *on average*, an LCA needs to control at least $\frac{|V|}{|C|}$ nodes to obtain a complete control structure. But apart from that, this setting for $n_{\min}$ is rather arbitrary. Any larger value would technically fulfill the same purpose, but could change the obtained results significantly. Also, a smaller value could possibly perform better for certain networks with low network load.

Given that GA3 already represents a genetic algorithm that interacts with the inner working of GreedyFCAPA, the straight forward option that presents itself is to add $n_{\min}$ to the representation of GA3, also undergoing genetic operations like crossover and mutation. For this *adaptive variant* of GA3, the $n_{\min}$ values are initialized uniformly at random with values within $[0, 3)$. Therefore, the adaptive GA3 variant also considers $n_{\min}$ values smaller than $\frac{|V|}{|C|}$. Crossover of the $n_{\min}$ part of the representation is done by taking the average of the two corresponding values of the parents. Mutation is performed by *nonuniform gaussian mutation* [45], i.e. the $n_{\min}$ value is manipulated by adding a normally distributed random variable $X \sim \mathcal{N}(0, 1)$.

4.5 Evaluation

In this section, I evaluate and discuss the performance of our GA approaches. I will refer to them as GA1, GA2 and GA3, in order of appearances. First, I define default parameters as a reference for further evaluation in Section 4.5.1 and provide initial performance results based on these. Next, I provide several parameter evaluation results for the three GAs in Section 4.5.2 and finally compare with GreedyFCAPA in Section 4.5.3. All GA implementations were done in Python, partially supported by the Distributed Evolutionary Algorithms in Python (DEAP) engine [96].

All calculations are executed in single-threaded mode on Intel® Xeon® E5-2695 v3 CPUs running at 2.30 GHz. Each graph includes confidence intervals with a confidence level of 95%. The used evaluation scenario is completely identical to the one described in Section 3.6.1, although I have limited this evaluation part to using only network instances with 36 nodes. Since a single

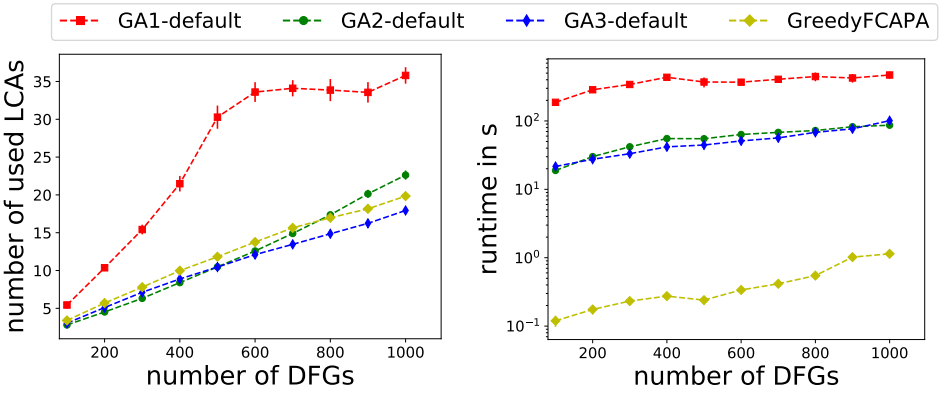
RCA sufficed to ensure a complete control structure for all runs, I will focus on the number of used LCAs in the following.

4.5.1 Default Parameters

To simplify the analysis of the influence of various GA parameters, I first specify default parameters, based on which I evaluate the impact of particular settings. The default settings have been verified by separate evaluations; to not unreasonably enlarge this chapter, I only present a limited selection of the most interesting results. Further results can be found in Scholz [94]. The chosen default settings are as follows:

Population size: $\mu = 20$.
Offspring size: $\lambda = \mu$.
Parent selection: The tournament size is set to 2.
Survivor selection: The best μ individuals from the population and its offspring.
Crossover/Mutation probability: $p_c = 0.2, p_m = 1 - p_c$.
Mutation intensity: $\alpha_m = 0.15$.
DFG satisfaction order (GA2 and GA3): DFGs are ordered by least demanding processing capacity first (or at random in case of a tie).
Termination: All genetic algorithms are run until the best all-time fitness does not improve for 15 generations.

These default settings are used for each evaluation run unless otherwise noted. But before evaluating any specific settings in particular, I provide an initial comparison of all three GAs with GreedyFCAPA based on the default parameters.



(a) Solution quality (b) Runtime

Figure 4.1: Performances with the default settings

Figure 4.1 depicts the performance of GreedyFCAPA and the three GAs for networks with 36 nodes and DFG counts ranging from 100 to 1000. Figure 4.1a shows the number of RCAs and LCAs used. It can be seen that

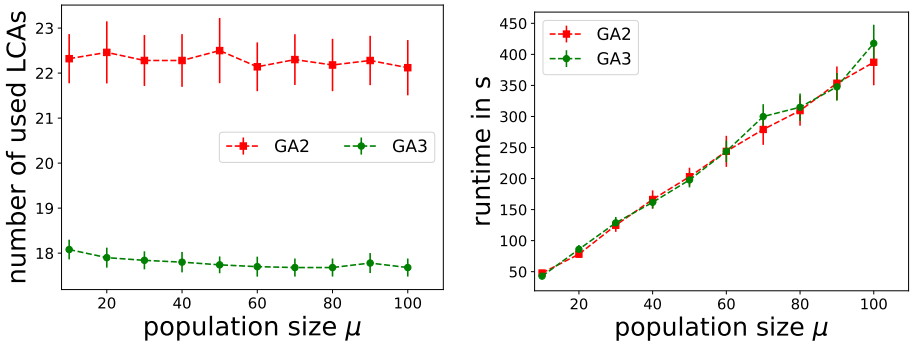
GA3 performs consistently better than GreedyFCAPA, while GA2 performs even better at first but deteriorates with higher network load and eventually gets outperformed by GreedyFCAPA starting from 900 DFGs. GA1 however provides solutions that require up to twice as many LCAs. Regarding runtime, Figure 4.1b illustrates that GA2 and GA3 terminate within several seconds up to few minutes, about three orders of magnitudes more than GreedyFCAPA. GA1 is unreasonably slow, taking another order of magnitude more.

In total, FCAPP seems to be too complex to be solved reasonably well by the pure GA approach GA1 while the initial results of GA2 and GA3 look quite promising. Because the computation time of GA1 would prevent from conducting more extensive evaluation runs, I decided to focus only on GA2 and GA3 in the remainder of this evaluation.

4.5.2 Parameter Evaluation

In this part, I take a closer look at individual parameter settings of GA2 and GA3. First, Figure 4.2 shows the performance of GA2 and GA3 for networks with 1000 DFGs depending on the population size μ.

Figure 4.2a shows that the average solution quality tends to improve only marginally with increasing μ. However, this improvement is consistently included within the confidence intervals of the remaining values for μ, which are particularly big for GA2. Figure 4.2b reveals, as expected, that the runtime increases linearly. As a result of these two observations, I decided to stick with the default value of $\mu = 20$.

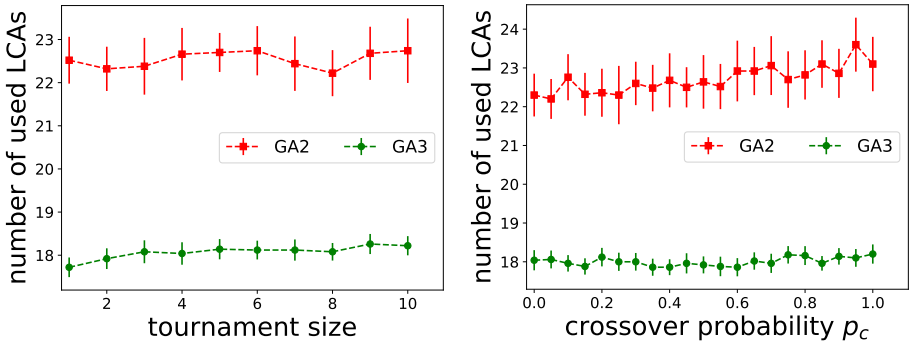


(a) Solution quality (b) Runtime

Figure 4.2: The influence of population size

Next, Figure 4.3 contains plots for parameter evaluations regarding parent selection and crossover/mutation probability, path length definition and DFG satisfaction order. Again, the evaluated networks have a fixed number of 1000 DFGs. Figure 4.3a displays the performance of GA2 and GA3 with different tournament sizes for parent selection. It can be seen that the tournament size has only little influence on the solution quality. It is surprising that a tournament size of 1 (which corresponds to selecting parents uniformly at

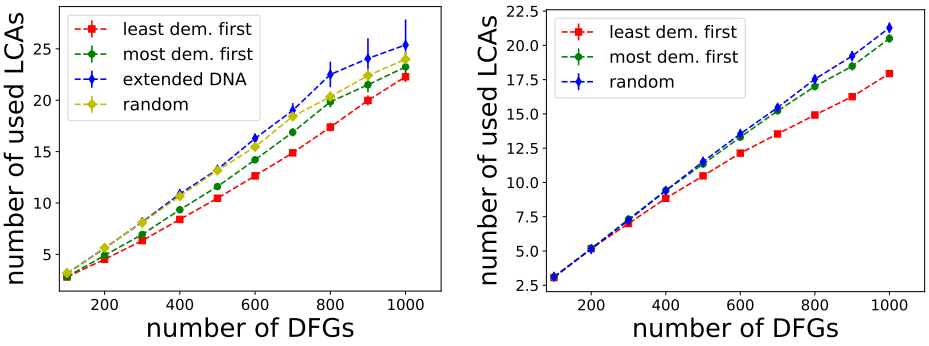
random) performs quite well. Apparently, for GA2 and GA3, the parent selection is no significant driving force for evolutionary improvement of the individuals.



(a) Parent selection (b) Crossover vs. Mutation

Figure 4.3: Parent selection and crossover vs. mutation probability

Then, Figure 4.3b shows the algorithms' performance as a function of the crossover probability p_c. Since $p_m = 1 - p_c$, a value of 0 for p_c means no crossover is performed, while $p_c = 1$ means that no mutation operations occur. It can be seen that the crossover probability does not affect GA3 notably, while GA2 seems to perform slightly better when crossover is rarely used. This could indicate that the mutation operator for GA2 provides a better genetic drift towards better solutions. But again, the differences are mostly covered by the confidence intervals and thus it is difficult to derive any conclusions. Again, I decided to stick to the default choice of $p_c = 0.2$ for the remainder of the evaluation.



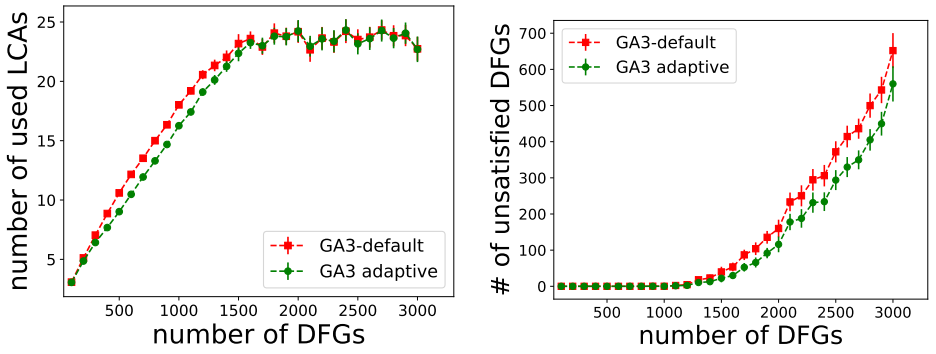
(a) DFG satisfaction order (GA2) (b) DFG satisfaction order (GA3)

Figure 4.4: DFG processing order of GA2 and GA3

As discussed in Sections 4.3 and 4.4, when it comes to satisfying DFGs, an order has to be specified in which the DFGs are considered. By default, the GAs order the DFGs by their required processing capacity $p_{\text{flow}}(f)$, least demanding ones first (LDF). But the DFGs can also be ordered by most demanding DFG first (MDF) or even randomly. Additionally, for GA2, there is the variant from Section 4.3.4, where the DFG processing order undergoes

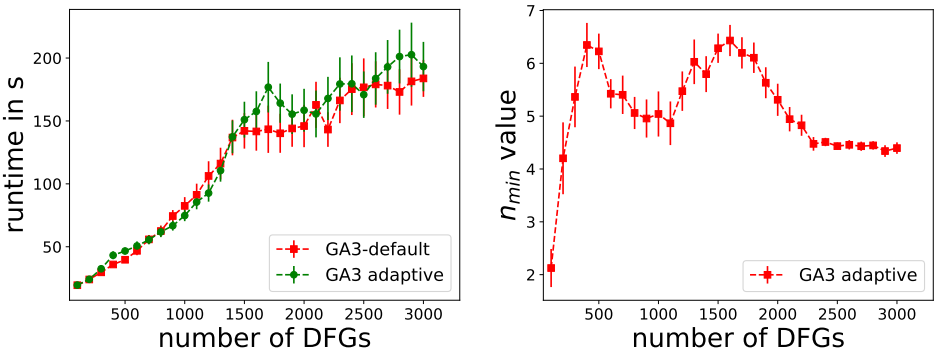
genetic operations just as the rest of the DNA. For all these different strategies, Figure 4.4 shows how well they perform on networks with a number of DFGs ranging from 100 to 1000.

For GA2, Figure 4.4a shows that the algorithm performs best when the DFGs are ordered with LDF strategy, followed by MDF ordering. Surprisingly, even a random order performs better than the GA2 variant with DFG order being part of the DNA. In theory, one would expect to see results at least as good as for the LDF strategy. However, I assume that the convergence towards a possibly better order happens too slowly and thus the GA2 variant is not able to come up with a better order before reaching the condition for termination. As depicted in Figure 4.4b, LDF ordering also performs best for GA3, while MDF ordering only performs slightly better than random ordering. Because of these results, LDF is kept as the default setting for both GA2 and GA3.



(a) LCAs used (GA3 variants)

(b) Unsatisfied DFGs (GA3 variants)



(c) Runtime (GA3 variants)

(d) $n_{\min}$ value (GA3 adaptive)

Figure 4.5: Comparison of the GA3 variants

The last parameter evaluation focuses on the variant of GA3 presented in Section 4.4.3. As mentioned there, the variable $n_{\min}$, which determines the number of formerly uncontrolled nodes that an LCA needs to control before it can satisfy DFGs, is set to $\frac{|V|}{|C|}$ by default. In the following, I compare the default GA3 algorithm with the adaptive GA3 variant with $n_{\min}$ being part of the DNA. The results featuring networks with 100 to 3000 DFGs are illustrated in Figure 4.5.

First, Figure 4.5a shows the number of LCAs used by GA3 default and GA3 adaptive. It can be observed that GA3 adaptive manages to use fewer LCAs until the network recources are exhausted. Figure 4.5b further reveals that GA3 adaptive consistently manages to leave fewer DFGs not satisfied and Figure 4.5c exposes that GA3 adaptive does not even need to run substantially longer to do this. In total, GA3 adaptive clearly outperforms the default GA3.

Finally, Figure 4.5d exhibits the $n_{\min}$ values included in the DNA of the final solutions of GA3 adaptive. Interestingly, no clear trend can be observed from the presented values. Because of the rather small confidence intervals, it can be concluded that the $n_{\min}$ value is rather consistent for a given number of DFGs. But since the DFGs generated for every distance are completely different, a dependence on the concrete DFGs in the network can reasonably be excluded. One possible explanation could be the number of used LCAs, which is also consistent per DFG count. However, one aspect that can be observed is that the $n_{\min}$ value determined by GA3 adaptive is consistently higher than the one chosen by default. Since the instances for this evaluation were generated with a probability of 0.6 for each node to be a potential host, the expected value for this default value is $n_{\min} \approx 1.67$.

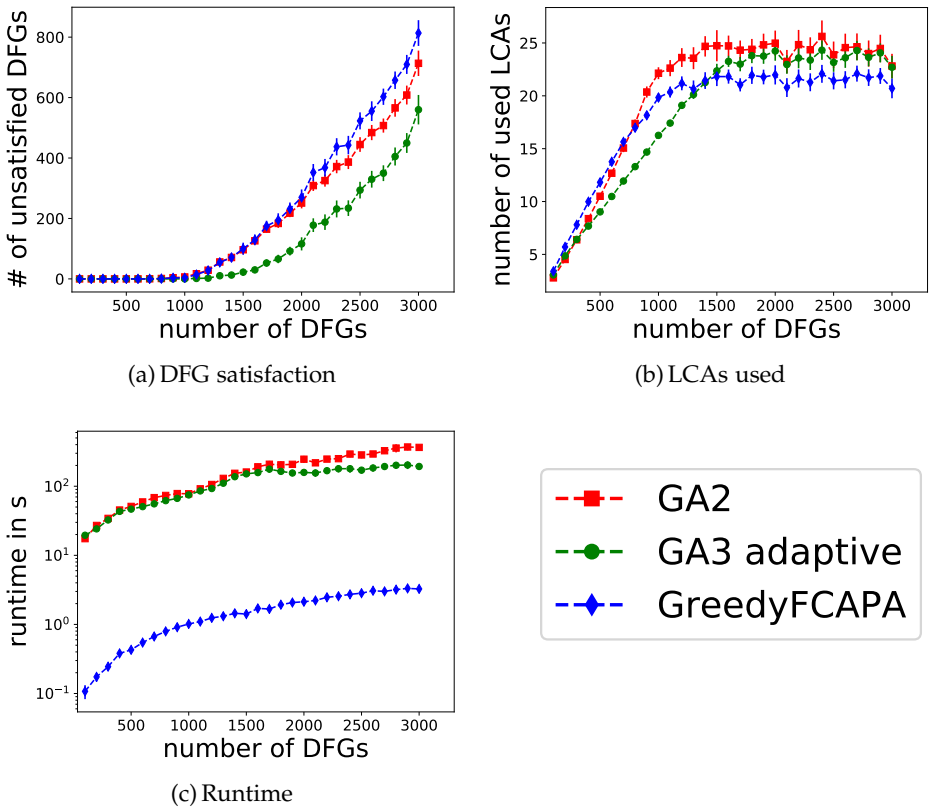
Due to the results above, I will look at the adaptive $n_{\min}$ variant of GA3 instead of the default GA3 in the last part of this evaluation. Unfortunately, the envisioned variant for GA2 failed to provide good results and no parameter improvements could be found for GA2. Therefore, GA2 will remain with the default parameters in the following.

4.5.3 Comparison with GreedyFCAPA

In the final part of this evaluation, I compare the performance of GA2 and the adaptive GA3 variant, now simply denoted as GA3, with GreedyFCAPA for 100 to 3000 DFGs. The results can be seen in Figure 4.6.

Figure 4.6a shows the number of unsatisfied DFGs. It can be seen that once the network resources are continuously exhausted, GreedyFCAPA causes the highest number of unsatisfied DFGs out of the three algorithms. GA2 performs slightly better, whereas GA3 beats both GreedyFCAPA and GA2 distinctly. Figure 4.6b provides the number of LCAs used by the algorithms and includes multiple interesting aspects to be observed. In the beginning, both GA2 and GA3 use fewer LCAs than GreedyFCAPA, but then GreedyFCAPA undercuts GA2 at 800 DFGs and GA3 at 1500 DFGs. There is, however, a significant difference between these two situations. When GreedyFCAPA undercuts GA2, both algorithms still satisfy all DFGs, which means that GreedyFCAPA outperforms GA2 at this point, even though this changes quickly once GA2 satisfied more DFGs than GreedyFCAPA. At 1500 DFGs however, GA3 satisfies way more DFGs than GreedyFCAPA, thus GreedyFCAPA only uses fewer LCAs because it is not capable of satisfying more DFGs by using the remaining ones. At last, Figure 4.6c features the runtime performance of the algorithms. Similar to Figure 4.1b, GreedyFCAPA runs around three orders

of magnitude faster than the GAs. These have a very similar runtime at first, but for more DFGs, it can clearly be seen that GA3 converges faster than GA2, on top of giving better results.



(a) DFG satisfaction

(b) LCAs used

(c) Runtime

Figure 4.6: Comparison of GreedyFCAPA with GA2 and GA3 adaptive

Overall, GA3 clearly outperforms GA2 in all regards, in particular after switching to the adaptive variant. GreedyFCAPA is beaten even more significantly in all metrics relevant to the solution but remains with the significant runtime advantage.

4.6 Observations

In this chapter, I have assessed the concept of GAs for FCAPP. On the one hand, the hybrid GA approaches gave very satisfying results, allowing me to answer my initial question, whether or not there is an alternate algorithmic approach that gives significantly better results than GreedyFCAPA within a reasonable amount of time, in the affirmative. But on the other hand, the pure GA approach gave poor results and some evolutionary principles, e.g. population size, crossover operators and parent selection, were no significant driving force towards the positive results. In contrast, the greatest positive influences have been achieved by choosing hybrid representations that left much of the work required to find a solution to greedy heuristics. So despite

obtaining satisfying results, this naturally raises doubts about whether a different algorithmic concept would have been a better choice for my mission to find better heuristic solutions for FCAPP.

Still, the adaptive GA3 variant represents a solution approach to be preferred over GreedyFCAPA, but only as long as the use case allows a runtime of several minutes. As elaborated in Section 1.2, I assume that such a runtime is generally too long, so that GreedyFCAPA is still to be favored in most cases. Moreover, GA3 revealed the potential for improving GreedyFCAPA by optimizing its parameters. I will revisit this possibility in Chapter 5.4.5.

5

Flow Processing-aware Control Application Placement with Proportional-Share Scheduling

After presenting several solution approaches for my initial formulation of FCAPP based on equal-share scheduling, this chapter describes how FCAPP can also be formulated based on *proportional-share* scheduling.

In Chapter 3, I assumed equal-share processing scheduling because it is a natural and easy way to allocate processing capacity among multiple entities. While this decision was appropriate for an initial problem formulation, it is also evident that equal-share scheduling results in a non-optimal distribution of processing capacity for entities with unequal processing demands as present in FCAPP. Therefore, I decided to extend the formulation of FCAPP based on proportional-share scheduling (which will be further discussed in Section 5.1). While the formulation based on a more elaborate scheduling scheme is expected to be more complex, it can also be expected that the possibility to distribute processing capacity non-equally will result in a significant improvement of solution quality.

I first give a short description of proportional-share scheduling, its theoretical advantages compared to equal-share scheduling and its effects on the problem statement of FCAPP in Section 5.1. Next, I describe the corresponding optimization model in Section 5.2 and elaborate on the modified version of GreedyFCAPA to work with proportional-share scheduling in Section 5.3. At last, I evaluate the solution approaches in Section 5.4 and compare them to the results of Section 5.4 to analyze the effect of the changed processing scheduling approach.

5.1 Proportional-Share Scheduling

As I described in Section 3.2, equal-share processing scheduling means that the processing capacity $p_{\text{node}}(c)$ of a potential host $c \in C$ is divided *equally* between all entities, i.e. all controlled nodes, coordinated LCAs and satisfied

DFGs. Employing equal-share scheduling allows to determine the processing capacity from c available for each unit by only keeping track of the number of units Proc_c served by c. The resulting processing capacity can then be expressed by $\frac{p_{\text{node}}(c)}{\text{Proc}_c}$ as done in constraints (3.16) – (3.18) of OPT_{es} (page 26).

However, this scheduling strategy also has a fundamental downside in the context of FCAPP: Let $c \in C$ be an RCA and/or LCA in a solution to an FCAPP instance. Then, the set of units served by c can be expressed by

$$U_s(c) := \{x \in F | \text{Sat}_{c,x} = 1\} \cup \{v \in V | \text{RCA}_{c,v} = 1 \vee \text{LCA}_{c,v} = 1\}.$$

Each unit $y \in U_s(c)$ requires a certain amount of processing capacity $p_c(y)$ from c to fulfill either constraint (3.16), (3.17) or (3.18). Now let $y^* \in U_s(c)$ be a unit so that

$$p_s(y^*) = \max_{y \in U_s(c)} \quad p_c(y).$$

This allows to deduce

$$p_s(y^*) \leq \frac{p_{\text{node}}(c)}{\text{Proc}_c} = \frac{p_{\text{node}}(c)}{|U_s(c)|} \quad \Leftrightarrow \quad |U_s(c)| \leq \frac{p_{\text{node}}(c)}{p_s(y^*)}.$$

This means that with equal-share scheduling, every unit served by a potential host $c \in C$ directly induces an upper bound for the number of units that c can serve in total. The bigger the differences between the processing requirements of the units served, the more this leads to resources being wasted instead of serving additional units.

Proportional-share scheduling resolves this issue. In contrast to dividing the available processing capacity of a potential host $c \in C$ equally between all served units, proportional-share scheduling *individually* allocates a *certain* share of the processing capacity to each served unit. In particular, the processing capacity is distributed among different units independent of each others' demands, as long as the total amount of allocated resources is less than or equal to $p_{\text{node}}(c)$. In total, proportional-share scheduling admits any combination that equal-share scheduling already allowed, but also allows for a lot of additional assignments that were previously prevented by equal-share scheduling. Figure 5.1 illustrates a toy example which would be invalid for equal-share scheduling (as P1 and P2 receive insufficient processing capacity) but is valid with proportional-share scheduling.

But while proportional-share scheduling obviously allows for a more efficient resource usage compared to equal-share scheduling, there is also a computational downside as proportional-share scheduling requires to determine and store appropriate allocations for each served unit individually. I will

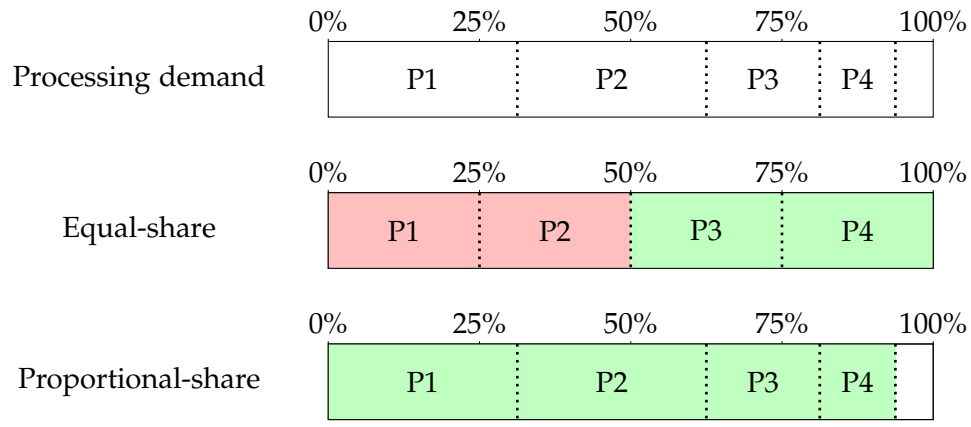


Figure 5.1: Exemplary comparison of equal-share and proportional-share scheduling: P1 and P2 cannot be assigned sufficient resources with equal-share scheduling.

elaborate more on this aspect when describing the optimization model with proportional-share scheduling in the following section.

At last, it is important to note that apart from the different processing scheduling approach, the problem statement from Section 3.2 remains unchanged. Similarly, there is no formal change in problem complexity, since the proof from Section 3.4 still applies without modification.

5.2 Optimization Model with Proportional-Share Scheduling

In this section, I describe my optimization model for FCAPP with proportional-share scheduling, denoted as OPT_{ps}. Just like OPT_{es} from Section 3.3, OPT_{ps} determines a solution for FCAPP corresponding to the objectives defined in Section 3.2 but based on proportional-share scheduling. Therefore, OPT_{ps} takes the same parameters as inputs as OPT_{es}, which are listed once more with shortened description in Table 5.1 for the reader's convenience.

Most of the variables used by OPT_{ps} to store the decisions about CA placement and DFG satisfaction are also identical to the ones used by OPT_{es} (Table 5.2). However, OPT_{ps} needs additional variables to store the decisions about the proportional processing shares, which are listed in Table 5.3.

Regarding the constraints of OPT_{ps}, it is possible to adopt all constraints from OPT_{es} that do not concern processing. Similarly, the objective function remains unchanged. In total, the following constraints have to be met. Some constraints use a big-M constant denoted as $\mathcal{M}$, which is a very large constant. The descriptions for constraints identical to constraints from OPT_{es} in Section 3.3 is intentionally kept short.

Table 5.1: OPT_{ps} input parameters

V	set of nodes
$C \subseteq V$	set of potential hosts
E	set of undirected links with $E \subseteq V \times V$
F	set of DFGs originating from at least one node $v \in V$
W	matrix with $W_{x,v} = 1$ iff $x \in F$ originates from $v \in V$
$p_{\text{node}}(c)$	processing power at node $c \in C$
$b_{\text{cap}}(v,w)$	maximum data rate for link $(v,w) \in E$
$l_{\text{cap}}(v,w)$	latency of link $(v,w) \in E$
$b_{\text{flow}}(x)$	data rate required by each flow of DFG $x \in F$
$l_{\text{flow}}(x)$	maximum acceptable round trip latency for DFG $x \in F$
$p_{\text{flow}}(x)$	operations per packet required for processing DFG $x \in F$
b_{LCA}	data rate required from an LCA-to-node routing path
b_{RCA}	data rate required from an RCA-to-LCA routing path
l_{LCA}	maximum acceptable LCA-to-node round trip latency
l_{RCA}	maximum acceptable RCA-to-LCA round trip latency
p_{LCA}	operations per control information packet required at an LCA to control a node
p_{RCA}	operations per control information packet required at an RCA to coordinate an LCA

Table 5.2: OPT_{ps} variables (identical to OPT_{es})

$\text{LCA}_{c,v} \in \{0,1\}$	determines whether $c \in C$ is an LCA for $v \in V$
$\text{RCA}_{c,d} \in \{0,1\}$	det. whether $c \in C$ is the RCA for LCA $d \in C$
$\text{isLCA}_c \in \{0,1\}$	determines whether $c \in C$ is an LCA
$\text{isRCA}_c \in \{0,1\}$	determines whether $c \in C$ is an RCA
$\text{Sat}_{c,x} \in \{0,1\}$	determines whether $c \in C$ satisfies $x \in F$
$\text{isSat}_x \in \{0,1\}$	determines whether $x \in F$ is satisfied by an LCA
$f_{c,u,v,w} \in \{0,1\}$	determines whether $(v,w) \in E$ is included in the routing path from LCA $c \in C$ to node $u \in V$
$g_{c,d,v,w} \in \{0,1\}$	determines whether $(v,w) \in E$ is included in the routing path from RCA $c \in C$ to LCA $d \in C$

Table 5.3: Additional OPT_{ps} variables

$p_{c,d}^{\text{RCA}} \in \mathbb{R}_+$	processing capacity reserved at RCA $c \in C$ for coordinating LCA $d \in C$
$p_{c,v}^{\text{LCA}} \in \mathbb{R}_+$	processing capacity reserved at LCA $c \in C$ for controlling node $v \in V$
$p_{c,x}^{\text{DFG}} \in \mathbb{R}_+$	processing capacity reserved at LCA $c \in C$ for processing DFG $x \in F$

LCA-to-node routing path constraints:

$$\sum_{(c,w)\in E} f_{c,d,c,w} = \mathrm{LCA}_{c,d}, \quad \forall c \in C, d \in V, c \neq d \tag{5.1}$$

$$\sum_{(u,d)\in E} f_{c,d,u,d} = \mathrm{LCA}_{c,d}, \quad \forall c \in C, d \in V, c \neq d \tag{5.2}$$

$$\sum_{(u,v)\in E} f_{c,d,u,v} = \sum_{(v,w)\in E} f_{c,d,v,w}, \quad \forall c \in C,\, d, v \in V,\, c \neq d \neq v \tag{5.3}$$

RCA-to-LCA routing path constraints:

$$\sum_{(c,w)\in E} g_{c,d,c,w} = \mathrm{RCA}_{c,d} \cdot \mathrm{isLCA}_d, \quad \forall c, d \in C, c \neq d \tag{5.4}$$

$$\sum_{(v,d)\in E} g_{c,d,v,d} = \mathrm{RCA}_{c,d} \cdot \mathrm{isLCA}_d, \quad \forall c, d \in C, c \neq d \tag{5.5}$$

$$\sum_{(u,v)\in E} g_{c,d,u,v} = \sum_{(v,w)\in E} g_{c,d,v,w}, \quad \forall c, d \in C,\, v \in V,\, c \neq d \neq v \tag{5.6}$$

Constraints ensuring a complete control strucure:

$$\sum_{c\in C} \mathrm{LCA}_{c,v} \geq 1, \quad \forall v \in V \tag{5.7}$$

$$\sum_{c\in C} \mathrm{RCA}_{c,d} = \mathrm{isLCA}_d, \quad \forall d \in C \tag{5.8}$$

$$\mathcal{M} \cdot \mathrm{isLCA}_c \geq \sum_{v\in V} \mathrm{LCA}_{c,v}, \quad \forall c \in C \tag{5.9}$$

$$\mathcal{M} \cdot \mathrm{isRCA}_c \geq \sum_{d\in C} \mathrm{RCA}_{c,d}, \quad \forall c \in C \tag{5.10}$$

DFG satisfaction constraints:

$$\sum_{c\in C} \mathrm{Sat}_{c,x} \leq 1, \quad \forall x \in F \tag{5.11}$$

$$\mathrm{isSat}_x = \sum_{c\in C} \mathrm{Sat}_{c,x}, \quad \forall x \in F \tag{5.12}$$

$$\mathrm{Sat}_{c,x} \leq \mathrm{LCA}_{c,v}, \quad \forall c \in C,\, x \in F,\, v \in V,\, W_{x,v} = 1 \tag{5.13}$$

Link capacity constraints:

$$\sum_{c\in C, d\in V} f_{c,d,v,w} \cdot \left(b_{\mathrm{LCA}} + \sum_{x\in F} W_{x,d} \cdot \mathrm{Sat}_{c,x} \cdot b_{\mathrm{flow}}(x) \right) + \sum_{c,d\in C} g_{c,d,v,w} \cdot b_{\mathrm{RCA}} \leq b_{\mathrm{cap}}(v,w), \quad \forall (v,w) \in E \tag{5.14}$$

As described in Section 5.1, the proportional-share scheduling model assigns *individual* shares of a host's processing capacity to each served unit. The processing time for each unit is obtained by dividing the required amount of operations per packet by its processing capacity share. For example, the processing time of a DFG $x \in F$ satisfied by LCA $c \in C$ is determined by

$\frac{p_{\text{flow}}(x)}{p_{c,x}^{\text{DFG}}}$. With this, the following constraints, analoguous to constraints (3.16–3.18) from Section 3.3, ensure that a sufficient amount of processing capacity is allocated for each unit so that the processing time plus link delays of the corresponding routing paths are smaller or equal to the latency requirements of each DFG (5.15), LCA (5.16) and RCA (5.17).

$$\begin{aligned}
&\text{Sat}_{c,x} \cdot \frac{p_{\text{flow}}(x)}{p_{c,x}^{\text{DFG}}} + \text{Sat}_{c,x} \cdot \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \Big(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v)\Big) \\
&\qquad \leq l_{\text{flow}}(x), \quad \forall c \in C, d \in V, x \in F, W_{x,d} = 1 && (5.15)\\
&\text{LCA}_{c,d} \cdot \frac{p_{\text{LCA}}}{p_{c,d}^{\text{LCA}}} + \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \Big(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v)\Big) \\
&\qquad \leq l_{\text{LCA}}, \quad \forall c \in C, d \in V && (5.16)\\
&\text{RCA}_{c,d} \cdot \frac{p_{\text{RCA}}}{p_{c,d}^{\text{LCA}}} + \sum_{(v,w)\in E} g_{c,d,v,w} \cdot \Big(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v)\Big) \\
&\qquad \leq l_{\text{RCA}}, \quad \forall c, d \in C && (5.17)
\end{aligned}$$

But processing capacity should only be provided to actually coordinated LCAs, controlled nodes and satisfied DFGs (5.18) – (5.20) and it must not be possible to assign more processing capacity than available at a potential host (5.21).

$$\begin{aligned}
&p_{c,x}^{\text{DFG}} \leq \mathcal{M} \cdot \text{Sat}_{c,x}, \quad \forall c \in C, x \in F && (5.18)\\
&p_{c,v}^{\text{LCA}} \leq \mathcal{M} \cdot \text{LCA}_{c,d}, \quad \forall c \in C, v \in V && (5.19)\\
&p_{c,d}^{\text{RCA}} \leq \mathcal{M} \cdot \text{RCA}_{c,d}, \quad \forall c, d \in C && (5.20)\\
&\sum_{d\in C} p_{c,d}^{\text{RCA}} + \sum_{v\in V} p_{c,v}^{\text{LCA}} + \sum_{x\in F} p_{c,x}^{\text{DFG}} \leq p_{\text{node}}(c), \quad \forall c \in C && (5.21)
\end{aligned}$$

Loop prevention and corner case constraints:

$$\begin{aligned}
&\sum_{(v,w)\in E,\, w=c} f_{c,d,v,w} = 0, \quad \forall c \in C,\, d \in V && (5.22)\\
&\sum_{(v,w)\in E,\, v=d} f_{c,d,v,w} = 0, \quad \forall c \in C,\, d \in V && (5.23)\\
&\sum_{(v,w)\in E,\, w=c} g_{c,d,v,w} = 0, \quad \forall c \in C,\, d \in V && (5.24)\\
&\sum_{(v,w)\in E,\, v=d} g_{c,d,v,w} = 0, \quad \forall c \in C,\, d \in V && (5.25)\\
&\text{LCA}_{c,c} = \text{isLCA}_c, \quad \forall c \in C && (5.26)\\
&\text{RCA}_{c,c} = \text{isRCA}_c \cdot \text{isLCA}_c, \quad \forall c \in C && (5.27)
\end{aligned}$$

Objective function (identical to OPT_{es}):

$$\text{minimize: } \sum_{c\in C} \big(\text{isRCA}_c + \text{isLCA}_c\big) - \omega \cdot \sum_{x\in F} \text{isSat}_x \quad (\omega > 2 \cdot |C|) \qquad (5.28)$$

Unfortunately, the optimization model in this form poses a practical issue, since (5.15) – (5.17) are fractional constraints that are not supported by common solvers such as Gurobi [91], which can at most handle quadratic constraints. Hence, it would be beneficial to tranform OPT_{ps} into an MIQCP just like OPT_{es}. As a first step, it is easy to at least get rid of denominator variables by multiplying the whole constraint with the corresponding variable:

$$p_{c,x}^{\text{DFG}} \cdot \left(l_{\text{flow}}(x) - \text{Sat}_{c,x} \cdot \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \left(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v) \right) \right)$$
$$\geq \text{Sat}_{c,x} \cdot p_{\text{flow}}(x), \quad \forall c \in C, d \in V, x \in F, W_{x,d} = 1, \tag{5.29}$$
$$p_{c,d}^{\text{LCA}} \cdot \left(l_{\text{LCA}} - \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \left(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v) \right) \right)$$
$$\geq \text{LCA}_{c,d} \cdot p_{\text{LCA}}, \quad \forall c \in C, d \in V, \tag{5.30}$$
$$p_{c,d}^{\text{LCA}} \cdot \left(l_{\text{RCA}} - \sum_{(v,w)\in E} g_{c,d,v,w} \cdot \left(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v) \right) \right)$$
$$\geq \text{RCA}_{c,d} \cdot p_{\text{RCA}}, \quad \forall c, d \in C. \tag{5.31}$$

Now, (5.30) and (5.31) are already quadratic constraints and can be used as a replacement for (5.16) and (5.17). But (5.29) is a cubic constraint and still constitutes a problem. So I decided to go a step back and to reanalyze constraint (5.29) theoretically, focusing on whether or not it really needs to be a cubic constraint. In Section 3.3, I explained why the multiplication by $\text{Sat}_{c,x}$ was necessary for the case that $\text{Sat}_{c,x} = 0$. But as it turns out, this is no longer the case for proportional-share scheduling since according to constraint (5.18) it holds that

$$\text{Sat}_{c,x} = 0 \quad \Rightarrow \quad p_{c,x}^{\text{DFG}} = 0$$

and thus the following is always fulfilled if $\text{Sat}_{c,x} = 0$:

$$\underbrace{p_{c,x}^{\text{DFG}}}_{=0} \cdot \left(l_{\text{flow}}(x) - \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \left(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v) \right) \right) \geq \underbrace{\text{Sat}_{c,x} \cdot p_{\text{flow}}(x)}_{=0}.$$

Therefore, it is now possible to replace (5.29) by (5.32):

$$p_{c,x}^{\text{DFG}} \cdot \left(l_{\text{flow}}(x) - \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \left(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v) \right) \right)$$
$$\geq \text{Sat}_{c,x} \cdot p_{\text{flow}}(x), \quad \forall c \in C, d \in V, x \in F, W_{x,d} = 1. \tag{5.32}$$

Finally, all practical issues of OPT_{ps} are resolved so that it can be implemented and solved by common solvers for optimization problems.

5.3 GreedyFCAPA with Proportional-Share Scheduling

In contrast to the creation of OPT_{ps}, modifying GreedyFCAPA to work with proportional-share scheduling turns out to be rather simple. Since the change only concerns the assignment of processing capacity, the main logic and hence all procedures shown in Section 3.5 remain unchanged. However, all procedures responsible for checking and adding RCA coordination, LCA control and DFG satisfaction have to be modified:

- checkRCAcontrol
- addRCAcontrol
- checkLCAcontrol
- addLCAcontrol
- checkFlowSat
- addFlowSat

Because of the iterative nature of GreedyFCAPA, i.e. units are served successively and a (potential) routing path is always given when checking and adding RCA coordination, LCA control or DFG satisfaction, it is easy to determine whether or not a potential host possesses a sufficient amount of remaining processing capacity to fulfill constraints (5.15) – (5.17) accordingly. Vice versa, when assigning a unit to be served and given a routing path P, it is possible to simply assign exactly the required amount of processing capacity corresponding to (5.15) – (5.17):

$$p_{c,x}^{\text{DFG}} = p_{\text{flow}}(x) \cdot \left(l_{\text{flow}}(x) - \sum_{(v,w) \in P} \left(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v) \right) \right)^{-1}$$

$$p_{c,v}^{\text{LCA}} = p_{\text{LCA}} \cdot \left(l_{\text{LCA}} - \sum_{(v,w) \in P} \left(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v) \right) \right)^{-1}$$

$$p_{c,d}^{\text{RCA}} = p_{\text{RCA}} \cdot \left(l_{\text{RCA}} - \sum_{(v,w) \in P} \left(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v) \right) \right)^{-1}$$

In addition and unlike the equal-share scheduling case, it is only necessary to keep track of a host's remaining processing capacity but no longer necessary to check if the requirements of an already served unit are still fulfilled before serving an additional unit. This could lead to a significantly reduced runtime, which I will try to confirm in the following evaluation.

5.4 Evaluation

In the evaluation of this chapter, I compare the results of the FCAPP solution approaches from Chapter 3 with the modifications presented in this chapter to analyze the possible performance gain through considering proportional-share scheduling instead of equal-share scheduling.

After explaining the evaluation scenario in Section 5.4.1, I first evaluate OPT_{es} and OPT_{ps} against the two versions of GreedyFCAPA, which will be denoted as $\text{GreedyFCAPA}_{\text{es}}$ and $\text{GreedyFCAPA}_{\text{ps}}$ for the remainder of this section, in Section 5.4.2. Because of the high runtime of the optimization models, this is only possible for very small scenarios. Thus, I will also compare $\text{GreedyFCAPA}_{\text{es}}$ and $\text{GreedyFCAPA}_{\text{ps}}$ using larger, real-world size scenarios in Section 5.4.3. Because I expect that the change to proportional-share scheduling will have a major impact on the behavior of the solution approaches, I am also revisiting the two DFG assignment strategies LDF and MDF in these first two evaluation parts. Then, I will analyze the influence of the used backhaul topology on the obtained results in Section 5.4.4, which will be further motivated in Sections Section 5.4.1 and Section 5.4.3. At last, I revisit a possibility for parameter improvement in Section 5.4.5 that was observed in the previous chapter.

All evaluations are executed on Intel® Xeon® E5-2695 v3 CPUs running at 2.30 GHz. Both OPT_{es} and OPT_{ps} have been implemented using the Pyomo package for optimization modeling in Python [90] and solved with Gurobi [91] running in single-threaded mode. All plots contain confidence intervals at a 95% confidence level.

5.4.1 Evaluation Scenario

In addition to the mesh topologies that I introduced in Section 3.6.1, I present an alternative topology model in this section. Since mesh topologies cause high capital and operational expenses in practice, real-world backhaul networks are often built as *ring topologies*, consisting of a high-speed optical fiber ring that interconnects multiple smaller subnetworks, each one with a tree topology [97, 98, 87]. To obtain results based on a more common topology and to possibly observe different performance results based on different topologies, I decided to also model and consider ring backhaul topologies in the following.

Similar to my generated mesh topologies, the nodes of ring topologies are placed on a regular grid with a mean inter-BS distance of $\bar{s} = 1000\,\text{m}$, corresponding to an urban scenario [92], and are then shifted in both x and y direction using normally distributed random variables $X, Y \sim \mathcal{N}(0, \frac{\bar{s}}{8})$. Then, the backhaul links are generated in a two-step process:

1. Determine and connect the ring nodes,
2. Connect all remaining nodes.

For the first step, I start by determining the radius that minimizes the total distance to all nodes in the network, while the center of the ring is always positioned in the center of the regular grid. For an $n \times n$ network and node coordinates denoted as $(v_x, v_y)\,\forall\, v \in V$, it holds that

$$\begin{aligned}
&\text{dist}(v, \text{ring}) = \left| \sqrt{\left(v_x - \frac{(n-1)\bar{s}}{2}\right)^2 + \left(v_y - \frac{(n-1)\bar{s}}{2}\right)^2} - r \right| \\
\Rightarrow \quad & r = \underset{r}{\text{argmin}} \sum_{v \in V} \left(\sqrt{\left(v_x - \frac{(n-1)\bar{s}}{2}\right)^2 + \left(v_y - \frac{(n-1)\bar{s}}{2}\right)^2} - r \right)^2 \\
\Rightarrow \quad & \frac{\mathrm{d}}{\mathrm{d}r} \sum_{v \in V} \left(\sqrt{\left(v_x - \frac{(n-1)\bar{s}}{2}\right)^2 + \left(v_y - \frac{(n-1)\bar{s}}{2}\right)^2} - r \right)^2 = 0 \\
\Leftrightarrow \quad & r = \frac{\bar{s}}{n^2} \sum_{v \in V} \left(\sqrt{\left(v_x - \frac{n-1}{2}\right)^2 + \left(v_y - \frac{n-1}{2}\right)^2} - r \right)^2 .
\end{aligned}$$

Based on this ideal ring, I then proceed to determine and connect the ring nodes by starting with the node closest to the ideal ring and then building the ring clockwise by choosing the neighbor (according to the regular grid) being (1) in the correct direction and (2) closest to the ideal ring. A more detailed representation of this procedure can be found in Algorithm 5.1.

Algorithm 5.1 CREATERING()

ring_completed = *False*
$v^* = \text{argmin}_{v \in V}\ \text{dist}(v, \text{ring})$
$v^{\text{curr}} = v^*$
while ring_completed = *False* **do**
  $\text{grad} = -\left(v_x^{\text{curr}} - \frac{(n-1)\bar{s}}{2}\right) \setminus \left(v_y^{\text{curr}} - \frac{(n-1)\bar{s}}{2}\right)$
  $(a, b) = \left(\text{sgn}\left(v_y^{\text{curr}} - \frac{(n-1)\bar{s}}{2}\right), \text{grad} \cdot \text{sgn}\left(v_y^{\text{curr}} - \frac{(n-1)\bar{s}}{2}\right)\right)$
  $\text{cand} = \{v \in \text{neighbors}(v^{\text{curr}})\ \textbf{if}\ \langle (a, b), (v_x - v_x^{\text{curr}}, v_y - v_y^{\text{curr}}) \rangle \geq 0\}$
  if $v^* \in$ cand **then**
    $v^{\text{next}} = v^*$
    ring_completed = *True*
  else
    $v^{\text{next}} = \text{argmin}_{v \in \text{cand}}\ \text{dist}(v, \text{ring})$
  connect v^{curr} and v^{next}
  $v^{\text{curr}} = v^{\text{next}}$

After creating the ring, the remaining nodes can finally be connected. To do this, I always connect the unconnected node being closest to an already connected node, which results in multiple tree subnetworks with a ring node as root. As illustration, Figure 5.2 shows a generated 6×6 ring topology. On the left side, I show only the generated nodes together with the determined ideal ring and on the right side, I depict the resulting graph including the generated backhaul links.

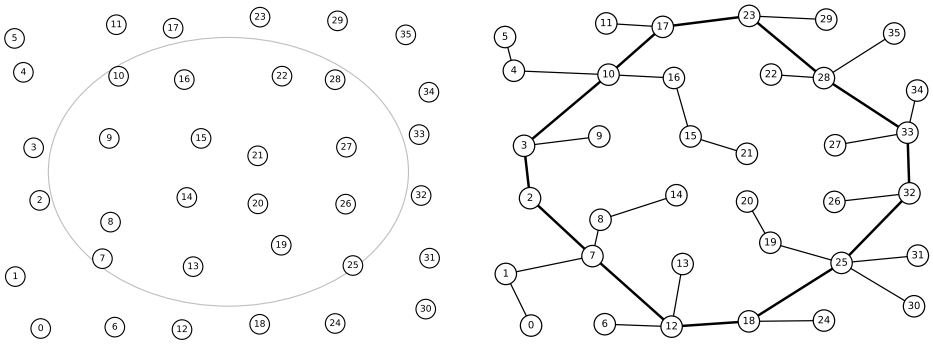


Figure 5.2: Exemplary generated ring topology

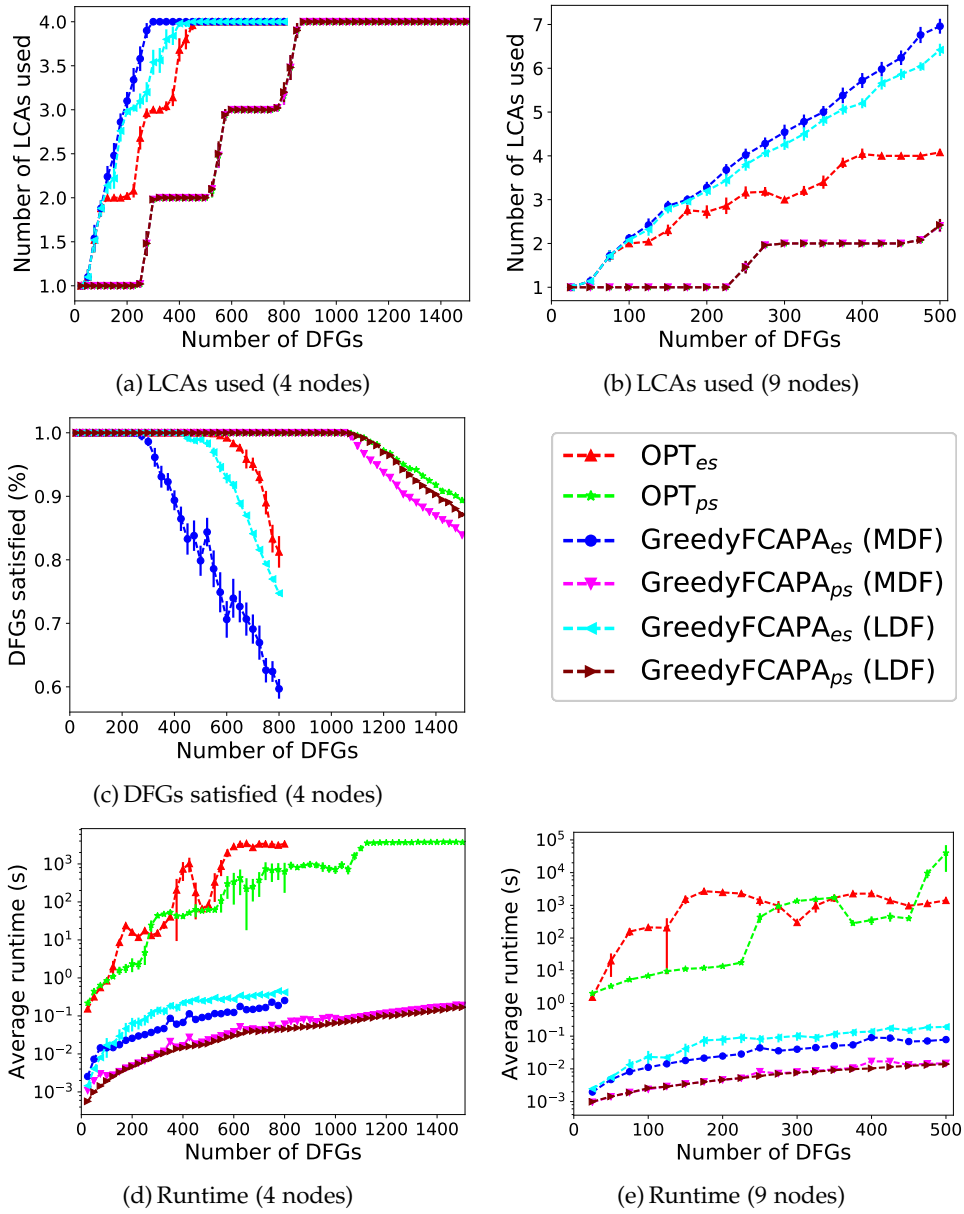
Regarding the link parameters, all links that are part of the ring are assigned a capacity of 5 Gbit/s while the remaining links are assigned a capacity of 2.5 Gbit/s. As in Section 3.6.1, the latency for each link is determined by its length multiplied by 1.45 and divided by the speed of light, assuming an optical backhaul network. In the remainder of this evaluation, each node becomes a potential host with a probability of P_C and is then assigned with a processing power of $p_{\text{node}} = 200$ GFLOPS. The DFG generation as well as all RCA, LCA and DFG parameters are kept identical to the evaluation scenario from Section 3.6.1.

5.4.2 Optimization Models vs. GreedyFCAPA Variants

To compare OPT_{es}, OPT_{ps}, $\text{GreedyFCAPA}_{\text{es}}$ and $\text{GreedyFCAPA}_{\text{ps}}$, I have generated mesh topologies with 4 and with 9 nodes, setting $P_C = 1.0$ to not unnecessarily reduce the solution space of the small networks. DFGs have been generated as multiples of 25 (i.e. 25, 50, 75,...) as long as the instances could still be solved in reasonable time. Fortunately, OPT_{ps} still provided a reasonable runtime for higher numbers of DFGs in the 4-node networks, in contrast to OPT_{es}, which allows to reveal more interesting aspects as will be seen further below. As in Section 3.6.2, I enhanced the optimization models by feeding them with the results of the respective heuristic approaches for reducing their search space. Similarly, a 1-hour time limit was set for their solving time. The few instances that did not find any valid solution within this time were allowed to run longer until a valid solution was found.

For all these instances, a valid solution has been found and consistently only one RCA was used. For the 9-node networks, all DFGs have been satisfied, so the corresponding plot was omitted. The remaining results can be seen in Figure 5.3. First, Figures 5.3a and 5.3b show the number of LCAs used by the algorithms. While OPT_{es} and $\text{GreedyFCAPA}_{\text{es}}$ show the same behavior already known from Section 3.6.2, OPT_{ps} and $\text{GreedyFCAPA}_{\text{ps}}$ reveal very pleasant performance. Not only does proportional-share scheduling outperform equal-share scheduling significantly, which could have been expected,

but in addition, GreedyFCAPA$_{ps}$ apparently always finds a solution using the optimal number of LCAs, unlike GreedyFCAPA$_{es}$ and independent of the DFG assignment strategy (LDF or MDF). Additionally, it can be observed that proportional-share scheduling seems to scale with the number of DFGs in a more obvious way than equal-share scheduling.



(a) LCAs used (4 nodes)

(b) LCAs used (9 nodes)

(c) DFGs satisfied (4 nodes)

(d) Runtime (4 nodes)

(e) Runtime (9 nodes)

Figure 5.3: Evaluation: optimization models vs. GreedyFCAPA variants

Next, Figure 5.3c depicts the percentage of DFGs satisfied in the 4-node networks. OPT$_{es}$ and GreedyFCAPA$_{es}$ are not able to satisfy all DFGs as already seen in Section 3.6, with a notable gap between OPT$_{es}$ and the GreedyFCAPA$_{es}$ variants. Because it was possible to also execute OPT$_{ps}$ for higher DFG counts, it can also be seen how OPT$_{ps}$ and GreedyFCAPA$_{ps}$ behave once the resources

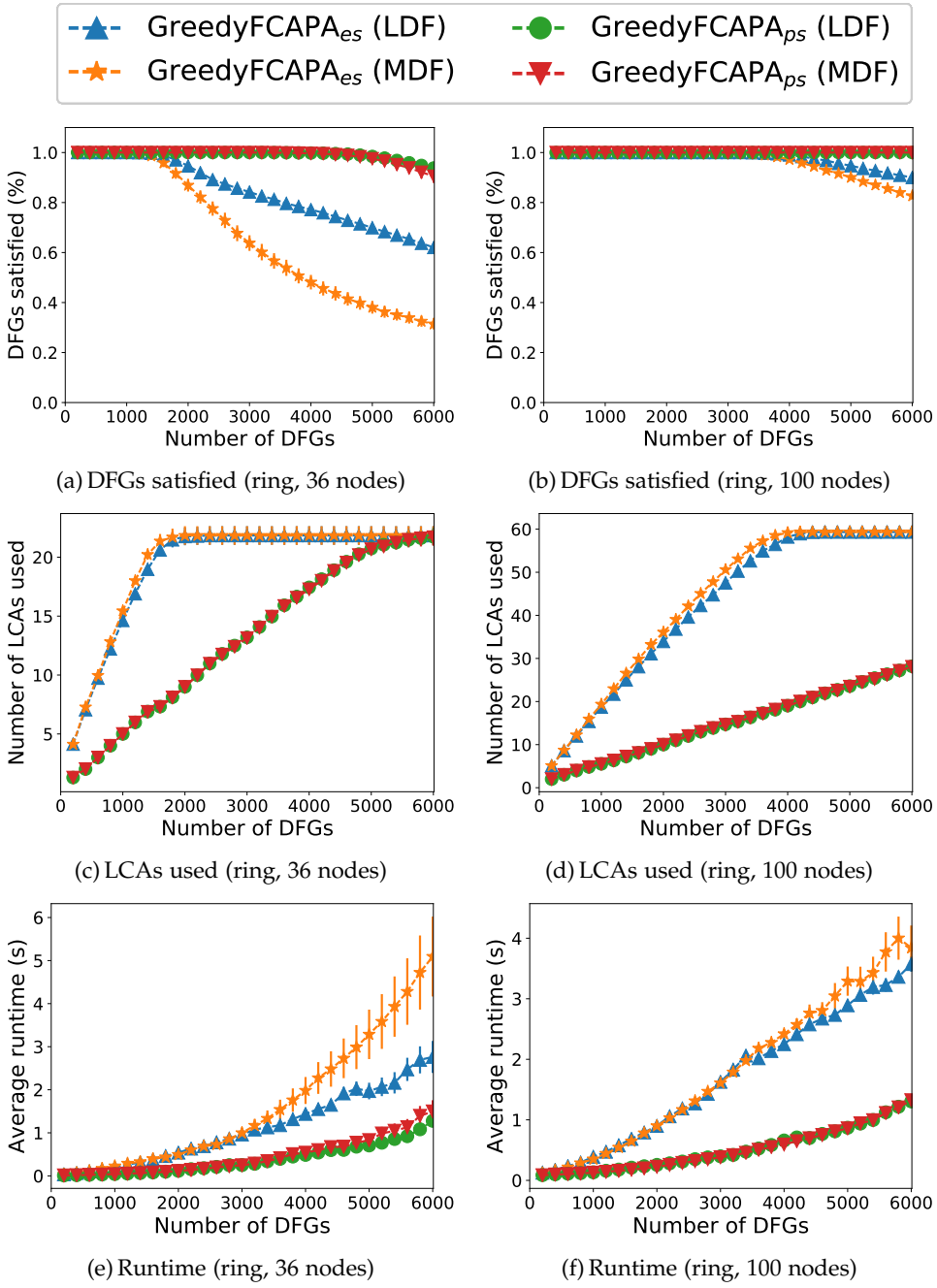
are no longer sufficient to satisfy all DFGs at around 1050 DFGs. First of all, GreedyFCAPA$_{ps}$ (LDF and MDF) is able to satisfy all DFGs as long as OPT$_{ps}$. Then, GreedyFCAPA$_{ps}$ with MDF strategy satisfies visibly fewer DFGs than OPT$_{ps}$, yet the gap is much smaller compared to the difference between OPT$_{es}$ and GreedyFCAPA$_{es}$ with MDF strategy. But GreedyFCAPA$_{ps}$ with LDF strategy continues to satisfy as many DFGs as OPT$_{ps}$ up to 1200 DFGs, before a small gap between both algorithms appears.

At last, Figures 5.3d and 5.3e illustrate the runtime required for the algorithms in logarithmic scale. Looking at GreedyFCAPA$_{es}$ and GreedyFCAPA$_{ps}$, it can clearly be seen that GreedyFCAPA$_{ps}$ runs significantly faster than GreedyFCAPA$_{es}$. This can be explained by the fact that GreedyFCAPA$_{ps}$ has to perform less checking when serving new units as remarked in Section 5.3. Moreover, GreedyFCAPA$_{ps}$ seems to work equally fast for both LDF and MDF strategies. Overall, GreedyFCAPA$_{ps}$ runs about three to six orders of magnitude faster than the optimization models.

5.4.3 GreedyFCAPA Variants in Larger Scenarios

For evaluating GreedyFCAPA$_{es}$ and GreedyFCAPA$_{ps}$ in larger scenarios, I have generated networks with 36 and 100 nodes, with both mesh topology and ring topology, $P_C = 0.6$ and multiples of 200 DFGs. For all instances, only one RCA was used and for the 100-node networks all DFGs have been satisfied. After looking at the results, my first observation was that there were very few and only minor differences between the mesh and ring topologies with otherwise identical parameters. Therefore, I analyze the differences separately in Section 5.4.4 to obtain deeper insight. In this evaluation part, I only show the results obtained from the ring topologies, which can be considered identical to the ones obtained from the mesh topologies, in Figure 5.4.

The plots in Figure 5.4 confirm what could also be observed in Section 5.4.2. Figures 5.4a and 5.4b show the percentage of DFGs satisfied in the 36-node and 100-node ring topologies. For 36 nodes in particular, it can be seen that GreedyFCAPA$_{ps}$ satisfies all DFGs up to around 4000 DFGs, significantly longer than GreedyFCAPA$_{es}$ where the resources are exhausted at around 1400 DFGs. For more than 5000 DFGs, GreedyFCAPA$_{ps}$ with LDF strategy again satisfies more DFGs than GreedyFCAPA$_{ps}$ with MDF strategy. Figures 5.4c and 5.4d illustrate the number of used LCAs. They show that GreedyFCAPA$_{ps}$ outperforms GreedyFCAPA$_{es}$ significantly, requiring less than half of the LCAs on average. Figures 5.4e and 5.4f depict the runtimes for the same networks, showing that GreedyFCAPA$_{ps}$ runs up to four times faster than GreedyFCAPA$_{es}$. Regarding the DFG assignment strategies LDF and MDF, two observations can be made. First, the influence of the chosen strategy is apparently way smaller on GreedyFCAPA$_{ps}$ compared to GreedyFCAPA$_{es}$. Second, the LDF strategy is still to be preferred over the MDF strategy for GreedyFCAPA$_{ps}$: the DFG satisfaction rate is higher, the number of used LCAs is equal and the runtime is either equal or marginally lower. As a result, I will continue to use the LDF assignment strategy as a default in the following.



(a) DFGs satisfied (ring, 36 nodes)

(b) DFGs satisfied (ring, 100 nodes)

(c) LCAs used (ring, 36 nodes)

(d) LCAs used (ring, 100 nodes)

(e) Runtime (ring, 36 nodes)

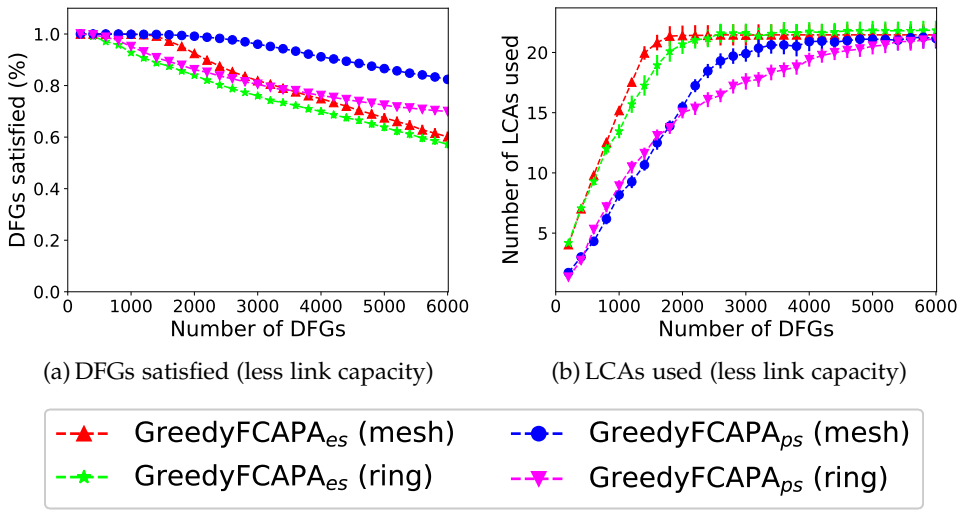
(f) Runtime (ring, 100 nodes)

Figure 5.4: Evaluation: GreedyFCAPA variants in larger scenarios

5.4.4 Mesh vs. Ring Topology Analysis

As described in the previous section, there were very few and only minor differences between the mesh and ring topologies with otherwise identical parameters. But given that both types of topology mainly differ in availability of links and thus link capacity, a logical explanation could be that my chosen

evaluation parameters cause the processing capacity to be the main bottleneck for DFG satisfaction. So in this section, I present the results obtained from the same instances as in Section 5.4.3 with the only difference that the link capacities of all network links have been scaled by a factor of 0.1. This should cause available link capacity and routing path lengths to be a more significant factor.



(a) DFGs satisfied (less link capacity)

(b) LCAs used (less link capacity)

Figure 5.5: Evaluation: GreedyFCAPA variants in larger scenarios with reduced link capacities

The relevant results of the 36-node topologies are illustrated in Figure 5.5. As the 100-node topologies showed a very similar behavior, I have omitted them here. Figure 5.5a depicts the percentage of DFGs satisfied in the 36-node mesh and ring topologies. Now, it can clearly be seen that more DFGs are not satisfied in the ring topologies than for the mesh topologies, for both GreedyFCAPA$_{es}$ and GreedyFCAPA$_{ps}$. In particular, GreedyFCAPA$_{ps}$ in the ring topologies is even outperformed by GreedyFCAPA$_{es}$ in the mesh topologies for up to 3400 DFGs. Certainly, this stems from the lower number of available links in the ring topologies. In addition, it is necessary to use longer routing paths in the ring topology, so that the same number of routing paths generally requires more link capacity from a ring topology compared to a mesh topology. Interestingly, the difference for GreedyFCAPA$_{es}$ is visibly smaller. It seems that for the inferior equal-share scheduling, processing capacity is still a major bottleneck despite the reduced link capacity.

Next, Figure 5.5b depicts the number of LCAs used; again the difference is more significant for GreedyFCAPA$_{ps}$. GreedyFCAPA$_{ps}$ first requires more LCAs in the ring topologies than in the mesh topologies, but starting from around 2000 DFGs this turns around. This can be explained as follows: since the link capacities have been reduced significantly, the links around a placed LCA are more quickly exhausted and thus additional LCAs have to be placed into parts of the network with less exhausted links. But with even more DFGs, the links are so exhausted that some DFGs cannot be satisfied even by placing

additional LCAs. Once more, GreedyFCAPA$_{es}$ is clearly less affected by this change. Despite the reduced link capacities, the inferior processing scheduling likely still leads to the processing capacity being the main bottleneck.

5.4.5 $n_{\min}$ Parameter Analysis

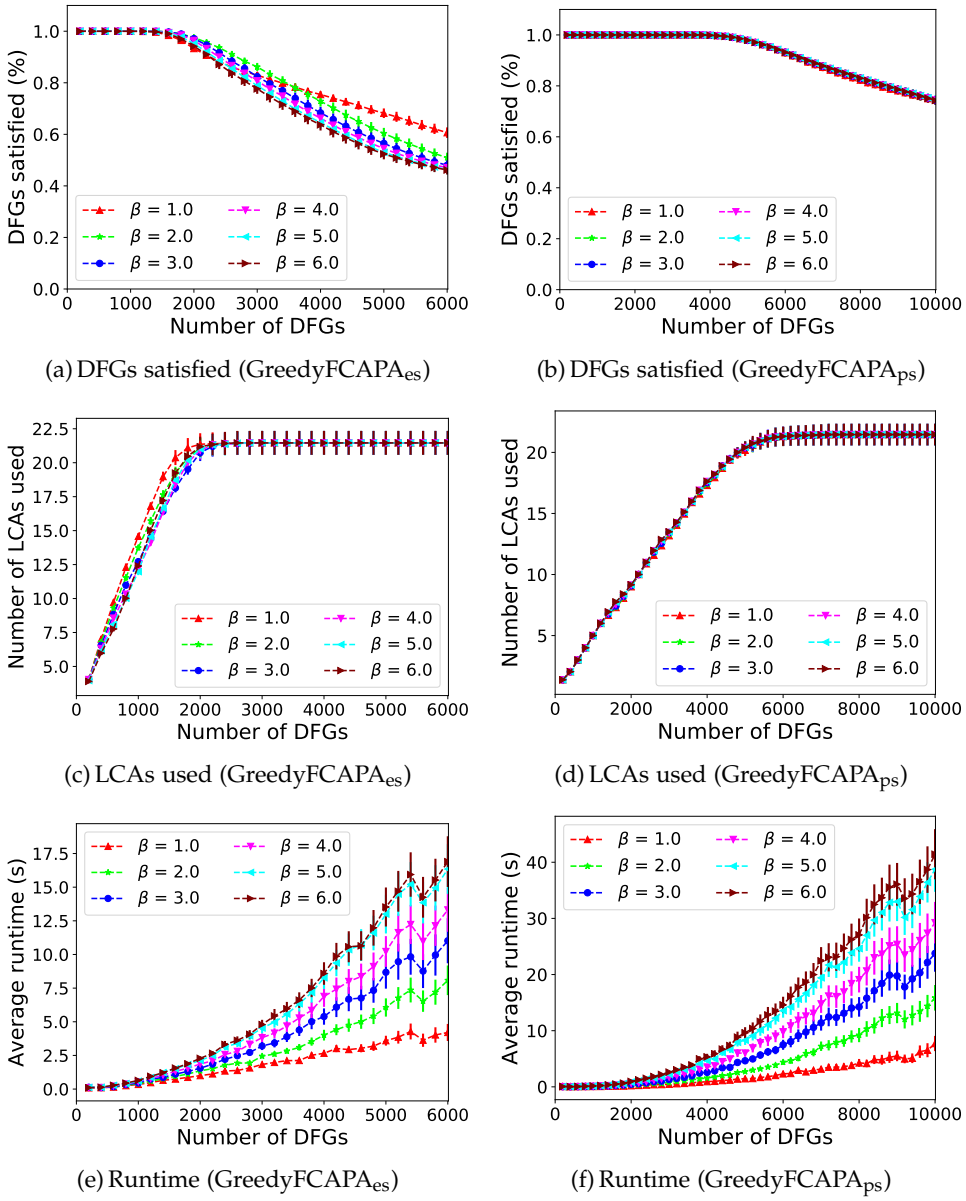
After observing the performance improvements obtained from switching from equal-share scheduling to proportional-share scheduling, I am now revisiting the observations from Section 4.6. At least for equal-share scheduling, the adaptive GA3 variant (Section 4.4.3) achieved notable performance improvements by performing genetic operations on

1. the order of the LCA candidates (Algorithm 3.4) and
2. the threshold for new LCAs to satisfy DFGs $n_{\min}$ (Algorithm 3.5).

However, the possibilities for an iterative algorithm such as GreedyFCAPA to adopt this are very limited. In particular, the order of the LCA candidates is already handled by best-effort heuristic metrics and attempting different orderings would basically mean re-executing multiple times, increasing the runtime by the same factor. For this reason, I decided to only look into the possibility of improving the $n_{\min}$ parameter, even though this lowers the performance gain expectations. As a reminder, the instances for this evaluation are generated with a probability of 0.6 for each node to be a potential host, thus the expected value for the currently used $n_{\min}$ is $\frac{|V|}{|C|} = \approx 1.67$. The results shown in Figure 4.5d suggest that it is beneficial to consider higher values. Therefore, I conducted a study using the 36-node networks from the previous sections with $n_{\min} = \beta \cdot \frac{|V|}{|C|}$, $\beta \in \{1, \ldots, 6\}$, where $\beta = 1$ corresponds to the current choice of $n_{\min}$.

The results of the study, which can be seen in Figure 5.6, depict the results for GreedyFCAPA$_{es}$ on the left and the results for GreedyFCAPA$_{ps}$ on the right. First of all, due to switching back to the default link parameters, the results obtained from the mesh and ring topologies were again very similar, so that I am only presenting the ones of the 36-node mesh topologies. Figures 5.6a and 5.6b exhibit the percentage of DFGs satisfied. For GreedyFCAPA$_{es}$ it can be seen that a higher β results in more DFG satisfied once the state of 100% DFG satisfaction is left. But there is no such affect on GreedyFCAPA$_{ps}$. The explanation for this is very related to my analysis of equal-share scheduling in Section 5.1, where I elaborated that the bigger the differences between the processing requirements of the units served, the more resources of a potential host are wasted. Keeping this in mind, it is now essential to remember that using a higher β value, i.e. requiring more new nodes to be controlled before being allowed to satisfy DFGs, practically means that every newly added LCA has a higher number of potential DFGs available once it is allowed to satisfy them. As a result, the differences between the processing requirements of the units served will decrease because of the applied LDF strategy. Hence, less resources are wasted and more DFGs are satisfied per LCA. However, as

also explained in Section 5.1, proportional-share scheduling does not suffer from this downside, which is why GreedyFCAPA$_{ps}$ is not notably affected by changing the β value.



(a) DFGs satisfied (GreedyFCAPA$_{es}$)

(b) DFGs satisfied (GreedyFCAPA$_{ps}$)

(c) LCAs used (GreedyFCAPA$_{es}$)

(d) LCAs used (GreedyFCAPA$_{ps}$)

(e) Runtime (GreedyFCAPA$_{es}$)

(f) Runtime (GreedyFCAPA$_{ps}$)

Figure 5.6: Evaluation: n_{min} parameter analysis for GreedyFCAPA$_{es}$ and GreedyFCAPA$_{ps}$

Similar observations can be made for the number of used LCAs depicted in Figures 5.6c and 5.6d. For GreedyFCAPA$_{es}$, higher β values result in fewer LCAs used (until all LCAs are used anyways). But the number of LCAs used by GreedyFCAPA$_{ps}$ is again unaffected by varying β. Obviously, this stems from the same effect that I explained above. Finally, the runtime results illustrated in Figures 5.6e and 5.6f are quite astonishing. Already for GreedyFCAPA$_{es}$, a higher n_{min} value significantly raises the runtime, but for

GreedyFCAPA$_{ps}$ the increase is even larger. While this runtime increase is higher than I would have expected, the increasing runtime as such can be explained by the larger search space created for each LCA by forcing it to control more nodes before satisfying any DFGs.

In total, it can be concluded that the positive effect of varying the $n_{\min}$ parameter observed in the previous chapter must have been tightly connected to the underlying equal-share scheduling. On the one hand, it is of course disappointing that this possible improvement is no longer valid for GreedyFCAPA$_{ps}$, but on the other hand, the increased robustness of GreedyFCAPA$_{ps}$ compared to GreedyFCAPA$_{es}$ is a satisfactory result. Obviously, I will stick to $n_{\min} = \frac{|V|}{|C|}$ based on these results.

5.5 Observations

In this chapter, I have revised my initial assumption from Chapter 3 of equal-share scheduling and have analyzed FCAPP based on proportional-share scheduling. While the implementation of proportional-share scheduling raised several issues for creating the corresponding optimization model, the adaptation into GreedyFCAPA turned out to be rather simple with astonishing performance results. As the evaluation part of this chapter showed, GreedyFCAPA$_{ps}$ used significantly less LCAs, satisfied more DFGs, had a significantly shorter runtime and also behaved more stable than GreedyFCAPA$_{es}$. Also, GreedyFCAPA$_{ps}$ attained close to optimal results compared with OPT$_{ps}$. Therefore, proportional-share scheduling is clearly to be favored over equal-share scheduling and I will consider the proportional-share version of FCAPP and the corresponding implementations in the remainder of this thesis.

Apart from the processing scheduling results, this chapter also includes the performance comparison of different backhaul topologies and the results turned out to be very robust as almost no differences could be observed. However, as I stated in Section 5.4.3, this only provides a conclusion for my chosen evaluation scenario. To avoid overlooking important aspects, I produced additional evaluation results with reduced link capacities that revealed significant differences between the two types of topologies. The question how much the topology influences the results and performance of FCAPP will also be revisited in later chapters. Finally, I took the opportunity to get back to the observations from Section 4.6, searching for further performance improvements. But the results clearly indicated that this opportunity was no longer relevant for proportional-share scheduling.

So in total, GreedyFCAPA with proportional-share scheduling represents a very powerful heuristic solution for a fixed network state. Indeed, having a fixed network state has been an implicit assumption for my evaluations so far. In the subsequent chapter, I will explore efficient solutions for the case when the network state, in particular the network load, changes.

6

Flexibly Reassigning Control Applications

The switch from equal-share to proportional-share processing resulted in a significant performance improvement for my FCAPP solution approaches. However, there is still one important aspect that I have neglected so far. All presented approaches have in common that they only consider CA placement for a fixed network state. But finding a good placement for a given state is only suitable for an *initial placement* situation and only solves the problem temporarily. As the network load of a mobile access network can change very quickly, the performance of a static one-time placement will typically become worse over time. In typical networks, many data flows appear and expire every second, which makes it necessary to modify DFG-to-LCA assignments or to reconfigure the current CA placement.

In this chapter, I thus rectify this shortcoming by addressing flexible CA reassignment in reaction to network load changing over time. I first describe my considerations for flexible reassignment in Section 6.1. These considerations inspired me to extent GreedyFCAPA to a Flexible flow processing-aware Control Application Placement Framework (FlexCAPF), which I elaborate in Section 6.2. At last, the performance of FlexCAPF is evaluated by means of dynamic network simulation in Section 6.3.

6.1 Reassignment Considerations

As explained above, it is crucial for mobile access networks to revisit and adjust CA placement decisions over time. The most obvious and easiest solution is to compute a new controller placement from scratch when the performance of the current placement deteriorates. But this approach has significant downsides. First, such a placement is expected to result in a lot of different assignments compared to the previous placement, i.e. a lot of nodes will be controlled by different LCAs and DFGs will be satisfied by different LCAs. Therefore, affecting such a placement decision in the network will cause much reconfiguration overhead. Second, generating a completely new

placement seems to be a waste of computational resources. In particular, even with a fast heuristic algorithm, the runtime for a completely new placement might be too high for frequently changing network load.

The alternative to computing a new placement from scratch is to consider flexible reassignment based on the current, *already existing* CA placement. The advantage of this approach is that existing assignments can be reused. Hence, reconfiguration overhead is reduced and with proper reassignment mechanisms in place, the runtime to generate a placement that maintains high network performance can be significantly decreased. In particular, it seems plausible to assume that for minor load changes in the network, the existing CA positions can be kept and that only a few LCA-to-node and DFG-to-LCA assignments have to be modified to adapt to the current network state. If this assumption holds, it justifies a reassignment approach rather than a from-scratch recomputing approach. But such an approach also has a downside. While a new placement from scratch is optimized for the current network state, a reassignment builds up on the current placement, which was optimized for a previous, outdated network state. As a result, retained LCA-to-node and DFG-to-LCA assignments might be non-optimal for the current network state and the resource usage efficiency might be worse compared to a placement computed from scratch. For this reason, designing adequate mechanisms for reassignment is very important when building on the existing placement.

As my focus for this thesis lies on efficient CA placement decisions that happen quickly enough to flexibly adapt placement decisions during network operation, even for high and quickly changing load, I study how to efficiently build on the existing placement. To do so, I have decided to extend GreedyFCAPA into a flexible CA placement framework with additional procedures to handle different reassignment situations, all optimized for transforming a given placement into a suitable one for an updated network state.

But before determining possible situations that warrant a reassignment, it is first important to extend the problem statement by a notion of time for the DFGs. So far, this was not necessary since only fixed network states were considered. However, when looking at reassignment over time, it is necessary to define when a DFG enters a network and when it expires again. I do this by defining an arrival time $t_{\text{in}}(x)$ and a duration $t_{\text{dur}}(x)$ for each DFG $x \in F$ (Table 6.1).

Table 6.1: Additional DFG parameters

$t_{\text{in}}(x)$	arrival time of DFG $x \in F$
$t_{\text{dur}}(x)$	duration of DFG $x \in F$

Now, given an existing CA placement with a complete control structure already in place, there are several cases which should result in (1) modifying assignments for the current CA placement or (2) changing the current CA placement. One such case is of course new DFGs arriving to the network. If

such a DFG can be satisfied with the current placement, the corresponding DFG-to-LCA assignment has to be established. If this is not possible, a new LCA has to be added to the network, i.e. the CA placement is changed.

In addition, it would also be possible to re-evaluate DFGs, which could not be satisfied during a previous execution, at a later point in time. But because it is uncertain whether or not a DFG request is still relevant at some point in time past its initial arrival time, I decided to assume that this issue is handled outside the scope of my work. This means that not satisfied DFGs are rejected and if the desire to process this DFG still exists at a later point in time, I assume it to be injected into the backhaul network again then. My framework will then perceive it as a new incoming DFG request and handle it accordingly.

Apart from additional load in the network, it is also important to assess when the network load has decreased by a margin that allows to save resources by deactivating CAs. I call such situations *low-load situations*. Last but not least, I also want to consider the unlikely, yet crucial case where a CA host fails, i.e. an RCA or LCA is suddenly removed from the network.

In total, I consider the following events as triggers to change the network configuration:

1. Incoming DFGs
2. Low-load situations
3. CA host failures

Finally, since I already consider CA host failures, one could also think of the case when additional resources are added to the network, e.g. a new potential host or additional processing capacity for an existing host. While this is of course possible, this does not need a special treatment as new resources will automatically be utilized if needed for incoming DFGs. Also, in case of major changes to the network resources, it would probably be advisable to compute a new CA placement from scratch for once to optimally utilize resources that were added in strategically good locations.

6.2 Flexible Multi-layer Greedy Framework

In this section, I present my Flexible flow processing-aware Control Application Placement Framework (FlexCAPF). FlexCAPF fully incorporates GreedyFCAPA as presented in Sections 3.5 and 5.3 for initial CA placement. In addition, FlexCAPF includes procedures to handle the different cases described in the previous section in which a reassignment is expedient, which are described in the following.

6.2.1 Satisfying Incoming DFGs

Every time a new DFG appears in the network, FlexCAPF at first tries to assign it to one of the existing LCAs by calling the BROWSELCASFORDFG procedure (Algorithm 6.1), which is designed specially for satisfying individual incoming DFGs fast and efficiently. Unlike the ADDNEWLCA procedure (Algorithm 3.5), where an LCA satisfies as many DFGs as possible from its nearby nodes, BROWSELCASFORDFG specifically tries to assign a certain DFG to an existing LCA. To do this, the procedure first looks at the LCAs that happen to already control all nodes that the new DFG x is originating from, starting with the LCA with the least distance to all nodes x is originating from. In case of a tie, the LCA with the highest number of DFGs already satisfied from these nodes is considered first. If no such LCA exists or can satisfy x, then all remaining LCAs are considered, starting with the LCA with the smallest accumulated distance to all nodes (i) x is originating from and (ii) are not yet controlled by that LCA. This way, the fewest link capacity is consumed for creating the additional control assignments.

Algorithm 6.1 BROWSELCASFORDFG(x)

```
LCAs* = {LCAs controlling all v ∈ V with W_{x,v} = 1}
sort LCAs* ascending by average distance to all nodes x is originating from first,
by number of DFGs already satisfied from these nodes descending next
for c in LCAs* do
  if CHECKDFGSAT(c, x) = True then
    return ADDDFGSAT(c, x)
LCAs+ = LCAs - LCAs* // now check all other LCAs
sort LCAs+ ascending by accumulated distance to all nodes (i) x is originating
from and (ii) are not yet controlled by the LCA
for c in LCAs+ do
  paths = {SHORTESTPATH(source=c, target=v) if c does not control v and W_{x,v} = 1}
  for p in paths do
    if CHECKLCACONTROL(p) = True then
      ADDLCACONTROL(p)
    else
      break // c cannot satisfy x
  if (all nodes assigned) and CHECKDFGSAT(c, x) = True then
    return ADDDFGSAT(c, x)
  else // c eventually cannot satisfy x, remove obsolete control assignments
    for p in paths do
      REMOVELCACONTROL(p)
```

If BROWSELCASFORDFG fails to satisfy an incoming DFG, i.e. no existing LCA is able to satisfy the DFG, FlexCAPF launches CPGREEDY, the main procedure of GreedyFCAPA (Algorithm 3.1). As all network nodes are already controlled by an LCA at this point, the procedure will immediately pass to the second phase of the initial CA placement, i.e. it will add an additional LCA that will satisfy the new DFG if possible. Additionally, CPGREEDY will automatically add an additional RCA if necessary.

6.2.2 Low-load Situations

While it is easy to determine when adding an additional LCA is necessary, assessing when it is possible to remove an LCA while still satisfying all DFGs is much more complex. To do this, I first define a notion of *load* for an LCA. After the switch to proportional-share scheduling, a very intuitive choice regarding an LCA's processing capacity presents itself:

$$\text{load}(c) = p_{\text{node}}(c) - p_{\text{rem}}(c),$$

where $p_{\text{node}}(c)$ is the total processing capacity of $c \in C$ (Table 5.1) and $p_{\text{rem}}(c)$ is its *remaining* processing capacity that is retained by FlexCAPF (see Section 5.3). To detect low-load situations, FlexCAPF continuously monitors the LCA load in the network and creates an estimate of the number of LCAs actually needed for the current network load as described in Algorithm 6.2.

Algorithm 6.2 getLCAestimate()

```
LCAs_tmp = LCAs, LCAs_needed = {}, est = 0
sort LCAs_tmp descending by load(c), c ∈ LCAs
while L_lowload · Σ_{c∈LCAs_needed} p_node(c) < Σ_{c∈LCAs} load(c) and est < |LCAs| do
  est = est + 1
  LCAs_needed.append(LCAs_tmp[est])
return est
```

In short, the getLCAestimate procedure adds up the processing capacity of the current LCAs, starting with the one with the highest load, until the current network load could be handled by the combined processing capacities of the selected LCAs in theory. But in practice, it has to be recalled that the processing capacity that a host has to allocate for controlling a node or processing a DFG depends on the path delay from the host to the node or the nodes the DFG is originating from (longer path delay means that processing has to happen faster to stay within the delay budget). Hence, the number given by getLCAestimate is really just an estimate, as it cannot be guaranteed that the selected LCAs can handle all tasks of the non-selected LCAs. For this reason, the calculation also includes a parameter $L_{\text{lowload}} \in (0, 1]$, which steers the intensity of the low-load handling. A low choice for L_{lowload} will result in conservative estimates, while choosing L_{lowload} very close to 1 would lead to very aggressive, possibly unrealistic estimates.

Algorithm 6.3 LowLoad()

```
est = getLCAestimate()
while |LCAs| > est do
  removeLCAwithLeastLoad()
browseLCAs()
if (|V| − |V_controlled|) > 0 or (|F| − |F_satisfied|) > 0 then
  CPgreedy()
if |RCAs| > 1 then
  rearrangeLCAs
```

If the LCA estimate stays below the current number of used LCAs over the course of a time period of T_{lowload} seconds, FlexCAPF executes the LowLoad procedure that is described in Algorithm 6.3. The waiting time of T_{lowload} is crucial because it prevents too frequent removal and re-addition of LCAs when the network load fluctuates around a level which is just between needing a certain LCA and not needing it.

Algorithm 6.4 browseLCAs()

$n_{\min} = (|V| - |V_{\text{controlled}}|) \cdot |C|^{-1}$, *paths* = {}
for c **in** LCAs **do**
  $F_{\text{pot}}(c)$ = getPotentialDFGs(c), $V_{\text{new}}(c)$ = {}
  for v **in** V **if** $\text{LCA}_{c,v} = 0$ **do**
    if (v in $|V| - |V_{\text{controlled}}|$) or ($\exists\, x \in F : W_{x,v} = 1, \text{isSat}_x$ = False) **then**
      paths.append(shortestPath(source=c, target=v))
if $|V| - |V_{\text{controlled}}| > 0$ **then** // no valid solution yet
  isSolved = False
  sort *paths* by paths to uncontrolled nodes first, path length next
else // valid solution already found, focus on DFG satisfaction
  isSolved = True
  sort *paths* by paths to nodes with unsatisfied DFGs first, path length next
while ($|paths| > 0$ or $\sum_{c \in \text{LCAs}} |F_{\text{pot}}(c)| > 0$) and ($|V| - |V_{\text{controlled}}| > 0$ or $|F| - |F_{\text{satisfied}}| > 0$) **do**
  if isSolved = False and $|V| = |V_{\text{controlled}}|$ **then**
    isSolved = True
    sort *paths* by paths to nodes with unsatisfied DFGs first, path length next
  if ($\sum_{c \in \text{LCAs}} |F_{\text{pot}}(c)| > 0$) and (isSolved or $|paths| = 0$) **then**
    c = random.choice($\{c \in$ LCAs, $|F_{\text{pot}}(c)| > 0\}$)
  else
    $c = paths[0][0]$
  if $|F_{\text{pot}}(c)| > 0$ and (isSolved or $|V_{\text{new}}(c)| \geq n_{\min}$ or $|paths| = 0$) **then**
    x = getNextDFG($F_{\text{pot}}(c)$)
    if checkDFGsat(c, x) = True **then**
      addDFGsat(c, x)
    $F_{\text{pot}}(c)$.remove(x)
  else
    p = *paths*[0], *paths*.remove(p)
    if checkLCAcontrol(p) = True **then**
      addLCAcontrol(p), updatePotentialDFGs(c)
    else // no more resources left at LCA c
      $F_{\text{pot}}(c)$ = {}, *paths* = $\{p \in paths,\ p[0] \neq c\}$

LowLoad successively removes the LCA with the least load until the number equal to the estimate is reached and then calls the browseLCAs procedure. As can be seen in Algorithm 6.4, the procedure browses through all current LCAs and tries to assign uncontrolled nodes and not satisfied DFGs to them. This procedure works very similar to the addNewLCA procedure (Algorithm 3.5) which I have already described in detail in Section 3.5, apart from considering all current LCAs simulateously instead of just one newly added LCA. Therefore, I am omitting a more detailed description here. If there are still uncontrolled nodes or not satisfied DFGs after browseLCAs has finished, CPgreedy is launched to correct this as in Section 6.2.1.

Algorithm 6.5 rearrangeLCAs()

```
RCAs_tmp = RCAs
sort RCAs_tmp ascending by number of coordinated LCAs
for c in RCAs_tmp do
  GiveAwayLCAs(c) = True, CanTakeLCAs(c) = True
for c in RCAs_tmp do
  if GiveAwayLCAs(c) = False then
    continue
  LCAs_tmp = {v ∈ C, RCA_{c,d} = 1}
  for v in LCAs_tmp do
    for d in reversed(RCAs_tmp) do
      if CanTakeLCAs(d) = False then
        continue
      path = shortestPath(source=d, target=v)
      if checkRCAcontrol(p) then
        remRCAcontrol(c, v), addRCAcontrol(path)
        GiveAwayLCAs(d) = False, CanTakeLCAs(c) = False
        break
```

At the very end of LowLoad, the procedure rearrangeLCAs (Algorithm 6.5) is called in case the current placement contains more than one RCA. This procedure aims at reducing the number of used RCAs by moving LCAs from RCAs with few coordinated LCAs, starting with the one with least coordinated LCAs, to RCAs with more coordinated LCAs, starting with the one with most coordinated LCAs. Once an RCA has taken over an LCA, it is no longer allowed to give away LCAs. Vice versa, an RCA that has already given away an LCA is no longer allowed to take LCAs. Thus, the procedure converges rather quickly. It has to be noted that the procedure remRCAcontrol() automatically removes an RCA once it does no longer coordinate any LCA.

6.2.3 Handling CA Host Failures

Finally, I describe how FlexCAPF deals with CAs suddenly disappearing from the network, which can be caused by the unlikely event of a CA host failure in a real-world implementation. According to my considered CA hierarchy, the failure of an RCA will cause its coordinated LCAs to suddenly be uncoordinated and the failure of an LCA might cause its controlled nodes to be uncontrolled, in case that they previously had only one LCA and/or its satisfied DFGs to be no longer satisfied. With the procedures of GreedyFCAPA, which I presented in Section 3.5, however, handling such a situation is very straightforward. The latter case, which results in uncontrolled nodes and/or unsatisfied DFGs is already automatically dealt with, since FlexCAPF's main procedure CPgreedy will always attempt to find new LCAs for uncontrolled nodes and unsatisfied DFGs (as already seen in Sections 6.2.1 and 6.2.2).

In contrast, the case of uncoordinated LCAs does not exist in the initial CA placement execution flow, but accounting for it is possible without much additional effort. I have done this by extending CPGREEDY to check for uncoordinated LCAs initially (see Algorithm 6.6). If such LCAs exist, FlexCAPF uses the FINDRCA procedure for each of them, which will find an RCA if possible and automatically add a new RCA if necessary. If, however, finding an RCA is not possible, an uncoordinated LCA will be removed and the resulting uncontrolled nodes and unsatisfied DFGs will be handled in the remainder of CPGREEDY as described above.

Algorithm 6.6 CPGREEDY() – extension

```
U_LCA = {d ∈ LCAs, RCA(d) = None}
for d in U_LCA do
  c = FINDRCA(d)
  if c is None then
    REMOVELCA(d)
...
```

To close this section, Figure 6.1 summarizes the key aspects of FlexCAPF in a flow chart.

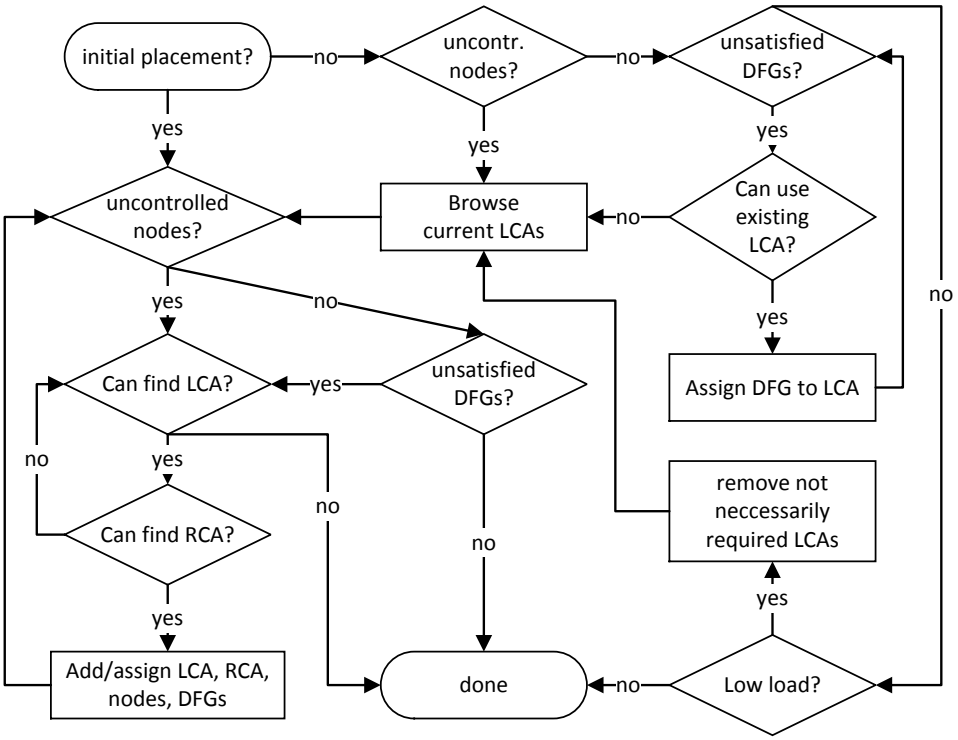


Figure 6.1: FlexCAPF flow chart

6.3 Evaluation

In this section, I present the results from a dynamic network simulation to evaluate FlexCAPF. In particular, I compare the results from FlexCAPF's flexible reassignment against the results obtained by recomputing a new CA placement from scratch. Every simulation has been run 30 times with different 64-bit random seeds. All calculations are executed in single-threaded mode on Intel® Xeon® E5-2695 v3 CPUs running at 2.30 GHz. All plots contain confidence intervals at a 95% confidence level.

6.3.1 Evaluation Scenario

For this evaluation I have simulated the network operation using FlexCAPF over the course of 48 simulated hours. To do this, I use four generated network topologies, having either 36 or 100 nodes and backhaul links with mesh or ring topology, which were generated according to the descriptions from Sections 3.6.1 and 5.4.1.

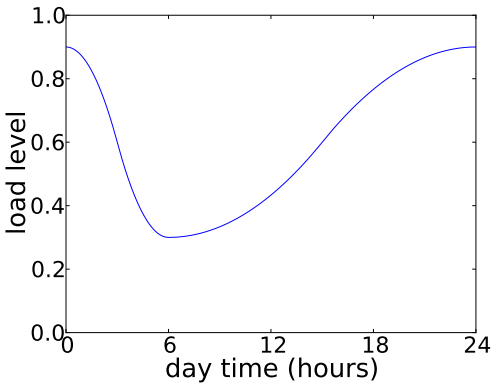


Figure 6.2: Evaluation: daily load curve

Incoming DFGs are generated using a non-stationary Poisson process with $\lambda = |V| \cdot \text{loadlevel}(t)$, simulated using the thinning method [99]. My daily load curve loadlevel(t), with t being the current time in seconds, is derived from [100] and shown in Figure 6.2. The duration of a DFG is determined using an exponential variable with parameter 0.02 $\frac{1}{\text{s}}$, which results in an expected DFG duration of 50 seconds and thus, according to Little's law [101], in an *approximated* (due to loadlevel(t) consistently changing with increasing t) expected number of DFGs in the network of $50 \cdot |V| \cdot \text{loadlevel}(t^*)$ at time t^*. The DFGs as such are generated according to the description in Section 3.6.1. Regarding to the low-load parameters described in Section 6.2.2, I chose $T_{\text{lowload}} = 60$ seconds and $L_{\text{lowload}} = 0.9$, which gave good results during initial evaluation runs.

All simulations are launched at $t = -3600$ seconds to start the 48-hour network monitoring at $t = 0$ in a running state, skipping a warm-up period. To generate the data for this evaluation, I have extracted the data from FlexCAPF each time

the set of used CAs was modified in the course of the simulation, i.e. the cases where incoming DFGs could be satisfied by an existing LCA are not included. At these points, I also perform an initial controller placement on an empty copy of the current network for comparison. This means, if a new incoming DFG can quickly be satisfied by an existing LCA (i.e. BROWSELCASFORDFG succeeded, see Section 6.2.1), I will not consider this a reassignment in the following as the set of used CAs is not affected. For the sake of illustration, Figure 6.3 contains the four used simulation networks.

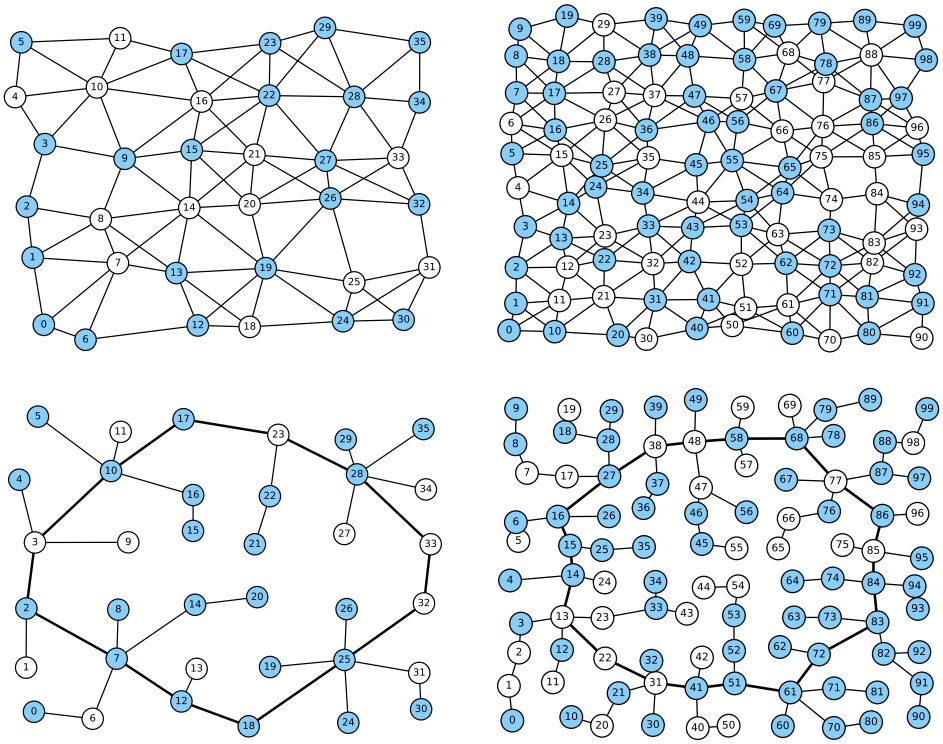


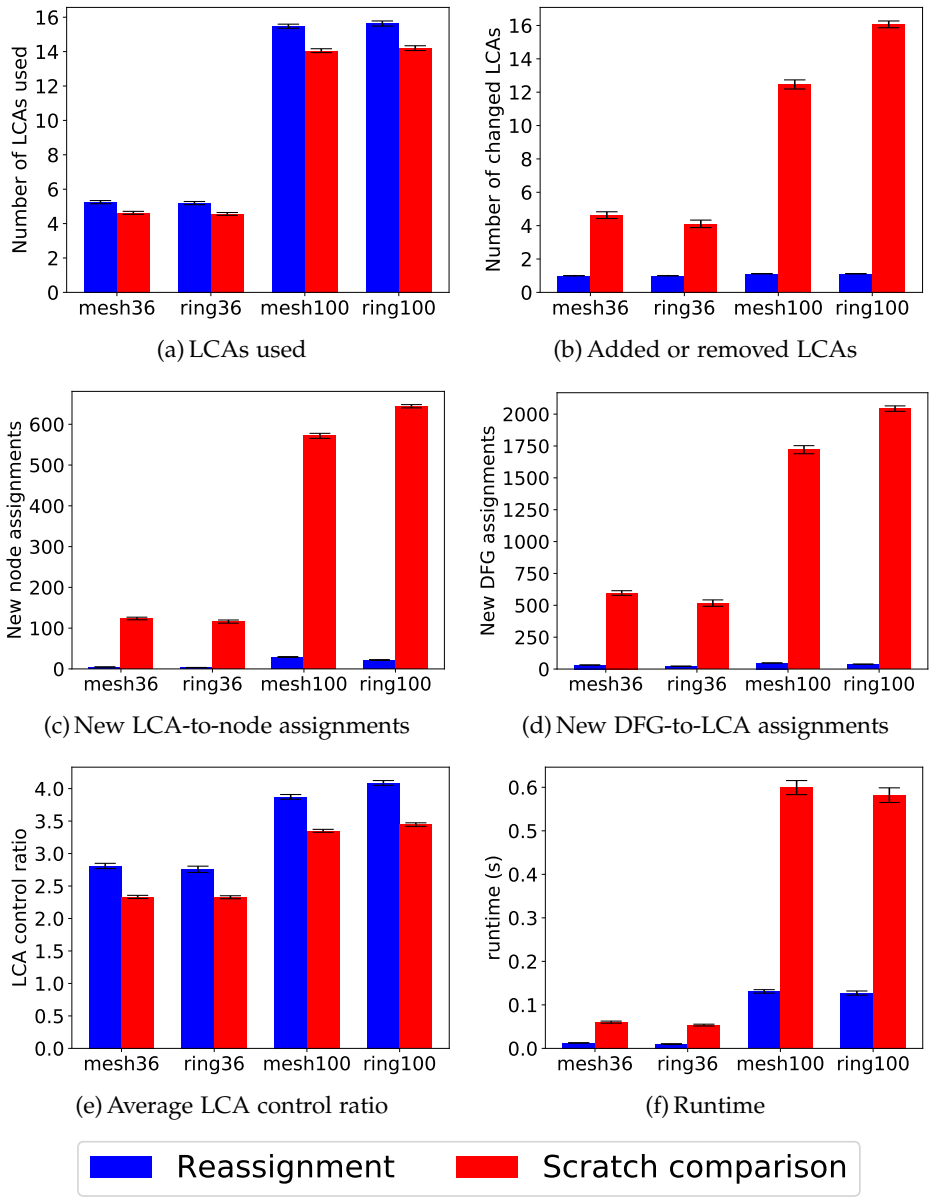
Figure 6.3: Simulation networks (with potential hosts highlighted)

I decided to not consider CA host failures for this evaluation as handling these situations is mostly done by procedures already known from Section 3.5. Thus, I do not expect additional insight by considering this, but rather unwanted distortion of the data obtained for adding additional CAs and for low-load handling.

6.3.2 Simulation Results

In this section, I summarize the results obtained from the simulations runs as described in Section 6.3.1. Throughout all simulations, FlexCAPF has generated valid solutions with only one RCA and all DFGs satisfied without exceptions.

Figure 6.4 summarizes the most important aspects. First, Figure 6.4a displays the average number of LCAs used. It can be seen that the flexible reassignment is clearly competitive with the from-scratch comparison considering the number of used LCAs, using only very few LCAs more on average. This is a good result since the reassignment is technically disadvantaged as it has to build up on an already existing controller placement.



(a) LCAs used (b) Added or removed LCAs

(c) New LCA-to-node assignments (d) New DFG-to-LCA assignments

(e) Average LCA control ratio (f) Runtime

Figure 6.4: Evaluation: FlexCAPF reassignment vs. initial CA placement

Next, Figures 6.4b to 6.4d give an impression of the reconfiguration overhead caused by the newly calculated assignments. Figure 6.4b shows the average number of added or removed LCAs compared to the set of active LCAs of the

previous placement. Further, Figure 6.4c and Figure 6.4d depict the average number of new LCA-to-node and DFG-to-LCA assignments. As can be seen, the flexible reassignment significantly outperforms the from-scratch initial placement for all these metrics. Hence, as I hoped, the flexible reassignment saves a large amount of reconfiguration overhead caused by establishing new assignments in the network.

Figure 6.4e shows a downside of the flexible reassignment. The plot illustrates the average *LCA control ratio* in the networks, i.e. the average number of LCAs a node is controlled by, which is visibly higher for the flexible reassignment than for the initial CA placement. I recognize that this is a downside of flexible reassignment based on a previous placement as described in Section 6.1. Apparently, the approach of finding an LCA for a specific DFG pursued by the flexible reassignment (Algorithm 6.1) leads to nodes being assigned to multiple LCAs as usually not always the same LCAs are going to have resources available to satisfy DFGs incoming at the same node. It has to be recalled that the flexible reassignment does not rearrange DFGs between LCAs to possibly save LCA-to-node assignments. The global view of the initial controller placement, however, provides a smaller LCA control ratio as an LCA is assigned to as many nearby nodes and the DFGs originating from them as possible.

In total, this means that the flexible reassignment has to devote slightly more network resources for the network control compared to the from-scratch initial placement, taking away resources for processing DFGs in theory. While this does not strongly affect the number of used CAs in my current evaluation scenario, this might still be different for other evaluation scenarios where network control requires more resources or if there are fewer resources available in the network. Therefore, I will further investigate and try to improve this aspect in Section 6.3.3.

Table 6.2: Simulation runtime statistics

Network:	mesh36	ring36	mesh100	ring100
Average number of runs (total):	64.97	60.27	65.33	62.3
Average number of runs (HL):	33.2	30.9	37.37	35.87
Average number of runs (LL):	31.77	29.37	27.97	26.43
Average runtime (reass.):	0.013 s	0.01 s	0.131 s	0.127 s
Average runtime (HL):	0.007 s	0.006 s	0.038 s	0.04 s
Average runtime (LL):	0.019 s	0.014 s	0.255 s	0.245 s
Average runtime (scratch):	0.06 s	0.054 s	0.599 s	0.582 s

Finally, Figure 6.4f shows the runtimes of reassignment and scratch comparison. It can be seen that the flexible reassignment runs significantly faster than the initial from-scratch controller placement. Table 6.2 gives a more fine-grained overview of the number of reassignment runs and the average

runtimes recorded during the simulations for each topology. For the flexible reassignment, I have also included separate numbers for reassignments due to needing an additional LCA (HL) and due to low-load situations (LL). Especially for the HL case, where a quick reaction is particularly important, the flexible reassignment runs about an order of magnitude faster than the initial placement from scratch.

6.3.3 Optional DFG Rearrangement

In this section, I further discuss the higher LCA control ratio of the flexible reassignment observed in the previous section and make an attempt to reduce it. While it is a strength of flexible reassignment to reuse existing LCA-to-node assignment, it is simultaneously one of its weaknesses with regard to efficient resource usage. Once such an assignment is established, the flexible reassignment will steadily reuse it. In particular, if a node $v \in V$ requires additional LCA-to-node assignments to satisfy the DFGs originating from it at a certain point in time, the flexible reassignment will still be using these assignments at a later point in time when they might no longer be needed. In particular, the reassignment as presented so far does not modify any existing DFG-to-LCA assignments to reduce the number of existing LCA-to-node assignments. As a result, the LCA control ratio slightly increases over time and network resources are wasted. For this reason, I decided to make an attempt to counteract this trend by implementing an additional procedure REARRANGEDFGS (Algorithm 6.7) that rearranges DFGs between

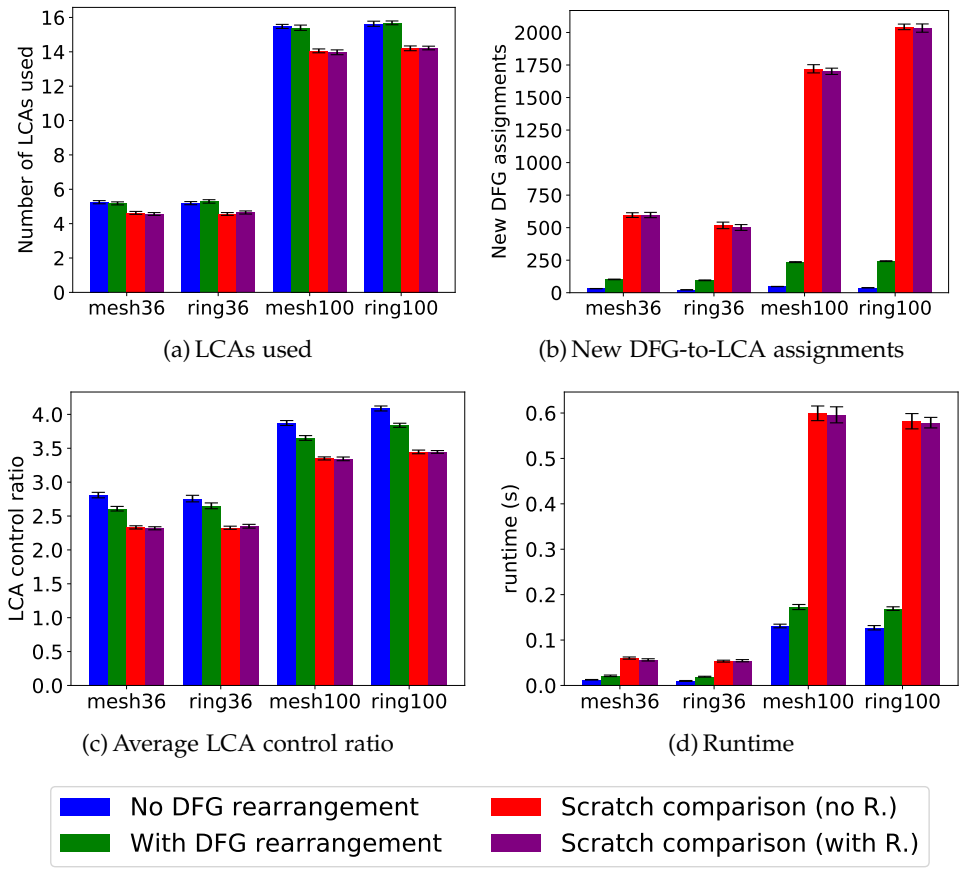
Algorithm 6.7 REARRANGEDFGS()

```
V_tmp = V
sort V_tmp descending by number of LCAs
for v in V_tmp do
  if |LCAs(v)| = 1 then
    break
  LCAs_tmp = LCAs
  sort LCAs_tmp ascending by number DFGs x satisfied with W_{x,v} = 1
  for c in LCAs_tmp do
    GiveAwayDFGs(c) = True, CanTakeDFGs(c) = True
  for c in LCAs_tmp do
    if GiveAwayDFGs(c) = False then
      continue
    F_tmp = {x ∈ F, Sat_{c,x} = 1, W_{x,v} = 1}
    sort F_tmp descending by p_flow // most demanding DFG first
    for x in F_tmp do
      for d in reversed(LCAs_tmp) do
        if CanTakeDFGs(d) = False then
          continue
        if CHECKDFGSAT(d, x) then
          REMDFGSAT(x), ADDDFGSAT(d, x)
          GiveAwayDFGs(d) = False, CanTakeDFGs(c) = False
          break
```

existing LCAs to reduce LCA-to-node assignments. The procedure is included in CPGREEDY and will thus always be called when the network configuration is changed.

For each node $v \in V$ with multiple LCAs, REARRANGEDFGS attempts to rearrange DFGs from LCAs that satisfy few DFGs originating from v to LCAs that already satisfy many DFGs originating from v. Because the implementation of REARRANGEDFGS follows a very similar strategy as REARRANGELCAS (Algorithm 6.5), I omit a more detailed description. However, it is important to recall that CPGREEDY already includes the CLEANUPLCACONTROLS routine, which removes all obsolete LCA-to-node assignments. So if REARRANGEDFGS successfully rearranges all DFGs originating from v from one LCA $c \in C$ to different LCAs, the corresponding LCA-to-node assignment between c and v will be removed afterwards.

To evaluate the effect of REARRANGEDFGS on the LCA control ratio and the other solution metrics, I ran additional simulations with the same random seeds as used in Section 6.3.2 to compare flexible reassignment without DFG rearrangement, flexible reassignment with DFG rearrangement and reassignment by initial controller placement from scratch. The relevant results can be seen in Figure 6.5.



(a) LCAs used

(b) New DFG-to-LCA assignments

(c) Average LCA control ratio

(d) Runtime

Figure 6.5: Evaluation: optional DFG rearrangement

Since adding DFG rearrangement causes reassignments to happen at different points in time, all plots in Figure 6.5 include both scratch comparisons, even though the results of both scratch comparisons did not show any significant differences. Similarly, the number of reassignments (total, HL and LL) did not change significantly either, for which reason I am omitting a detailed representation similar to Table 6.2.

First, Figure 6.5a shows the number of used LCAs and quickly reveals that rearrangeDFGs has no significant influence on this metric. The number of added or removed LCAs and new LCA-to-node assignments compared to the previous placement did not change either. The corresponding plots are omitted. Naturally, however, rearrangeDFGs causes a higher amount of new DFG-to-LCA assignments as can be seen in Figure 6.5b. This number is still way below the results obtained by the from scratch comparisons. Then, Figure 6.5c shows the targeted LCA control ratio. The use of DFG rearrangement did indeed result in a decrease, closing the gap to the from-scratch comparisons about half way. But in turn, Figure 6.5d shows that rearrangeDFGs results in an increased runtime compared to flexible reassignment without rearrangeDFGs. While the runtime is still below the one of the scratch comparison, the relative increase is clearly significant.

As a result, even though rearrangeDFGs brought a minor performance improvement, I decided not to activate the procedure in the remainder of the thesis because the relative runtime increase outweighs its benefit in my opinion.

6.4 Observations

In this chapter, I have shown how GreedyFCAPA, my fast heuristic FCAPP solution, can be extended into a flexible CA placement framework (FlexCAPF). My evaluation has shown that FlexCAPF is able to adapt a given placement very quickly and significantly faster than performing reassignment by calculating a new CA placement from scratch. Meanwhile, the loss in solution quality is minimal and reconfiguration overhead is vastly reduced. Especially for new incoming DFGs, FlexCAPF is able to react within very few milliseconds. As a result, the framework can possibly be used in a real-world implementation during network operation. I will come back to investigating this later in Chapter 10.

But more generally, FlexCAPF fulfills the two criteria that I listed in Section 1.2 and thus allows to answer the corresponding research question in the affirmative. However, there a lots of further aspects surrounding FCAPP that I have not yet considered but which I will will address in the following chapters.

7

Distributed Flow Processing-aware Control Application Placement

In the previous chapter, I have shown how FlexCAPF provides very convincing results for flexibly governing CA placement during network operation. However, FlexCAPF is a centralized algorithm; it relies on the fundamental assumption that all information about the network state is available in time for centralized decision-making. Given that so far one RCA was always sufficient in the previous evaluation parts, this information could potentially be available via this RCA, provided that a CA placement is already given. But providing all relevant information in real time, in particular information about new DFGs in the network, is hard to realize.

For this reason, this chapter is devoted to investigating an alternative, distributed approach for tackling the same tasks as FlexCAPF. The result is a *Distributed flow processing-aware Control Application Placement Algorithm (DistCAPA)* that places and flexibly reassigns CAs just like FlexCAPF. To simplify the algorithm, DistCAPA considers only LCAs, omitting the additional coordination layer provided by RCAs unlike the previous FCAPP solution approaches. DistCAPA has originally been created together with Dimitrij Pauls over the course of his bachelor thesis [102] under my supervision and has subsequently been considerably extended and refined by myself.

In the remainder of this chapter, I first state the modeling assumptions in Section 7.1, in particular about the information available for each node in the network. I then proceed by describing DistCAPA in Section 7.2 and by presenting evaluation results in Section 7.3.

7.1 Modeling Assumptions

In this section, I explain the fundamental modeling assumptions regarding information availability and algorithm execution for DistCAPA.

7.1.1 Information Availability

As I elaborated in Section 2.2.2, distributed algorithms are usually unaware of the entire system's state during their execution. Each network node should only be aware of its own local state and any additional information needs to be obtained via communication with other nodes from the system. So in contrast to FlexCAPF, it is essential to define for each node $v \in V$ a suitable subset of the entire FCAPP input parameters (Table 5.1 and Table 6.1) which is available at v without communication effort.

For defining these subsets, I decided to divide the input parameters into *static* information and *fluctuating* information. Static information comprises all parameters which I consider to be static over the course of a simulation run. Analogously, fluctuating information comprises all parameters that can change (possibly very frequently) over the course of a simulation. First, static information includes the parameters for the requirements of LCA control. I also consider all information about the network deployment as static, i.e. the sets of nodes, potential hosts and backhaul links and their fixed attributes, assuming an error-free operation of all network devices and network links. On the contrary, I consider all information about DFGs in the network as fluctuating, because DFGs are assumed to change frequently over time as described in Chapter 6.

With this partitioning of input parameters, I assume that all parameters related to static information are known by all network nodes, while all parameters related to fluctuating information are only known by directly affected network nodes. For example, each node $v \in V$ is only aware of the DFGs that either originate from it or are satisfied by it (provided that v serves as an LCA). Table 7.1 and Table 7.2 provide a complete list of both types of parameters.

Table 7.1: DistCAPA input parameters available to all network nodes

V	set of nodes in the network
$C \subseteq V$	set of potential hosts in the network
E	set of all network links with $E \subseteq V \times V$
$p_{\text{node}}(c)$	processing power for each node $c \in C$
$b_{\text{cap}}(v, w)$	maximum data rate for each link $(v, w) \in E$
$l_{\text{cap}}(v, w)$	latency of each link $(v, w) \in E$
b_{LCA}	data rate required from an LCA-to-node routing path
l_{LCA}	maximum acceptable LCA-to-node round trip latency
p_{LCA}	operations per control packet required for LCA control

It should be noted that not all static information is required in DistCAPA. For example, no node $v \in V$ ever uses the knowledge about $p_{\text{node}}(c), c \in C$ unless $v = c$. However, since I do not consider memory for storing this information a problem, bounding the availability of static information to the portion that a node really requires would not bring any benefit.

Table 7.2: DistCAPA input parameters only available to $v \in V$

F_v	set of DFGs with $W_{x,v} = 1$ or $\text{Sat}_{v,x} = 1$
V_x	set of nodes DFG $x \in F_v$ originates from
$b_{\text{flow}}(x)$	data rate required by each flow of DFG $x \in F_v$
$l_{\text{flow}}(x)$	maximum acceptable round trip latency for DFG $x \in F_v$
$p_{\text{flow}}(x)$	operations per packet required for processing DFG $x \in F_v$

But apart from the input parameters, it is also necessary to define where information about the placement decisions is know. Certainly, all such information has to be considered fluctuating. Table 7.3 summarizes which nodes are aware of which placement decision during the execution of DistCAPA.

Table 7.3: DistCAPA: availability of information about placement decisions

$\text{LCA}_{c,v}$	only c and v know whether $c \in C$ is an LCA for $v \in V$
isLCA_c	only c and all $v \in V$ with $\text{LCA}_{c,v} = 1$ know if $c \in C$ is an LCA
$\text{Sat}_{c,x}$	only c and all $v \in V$ with $W_{x,v} = 1$ know if $c \in C$ satisfies $x \in F$
$f_{c,u,v,w}$	only c and u are aware of the $(v, w) \in E$ used for the routing path from LCA $c \in C$ to node $u \in V$
$p^{\text{LCA}}_{c,v}$	only c is aware of the processing capacity reserved at LCA $c \in C$ for controlling node $v \in V$
$p^{\text{DFG}}_{c,x}$	only c is aware of the processing capacity reserved at LCA $c \in C$ for processing DFG $x \in F$

All necessary information that is not available to a certain node according to the tables above has to be retrieved via communication during the execution of DistCAPA.

7.1.2 Execution Model

In contrast to centralized algorithms, which are assumed to be executed on a single machine, distributed algorithms require the definition of an execution model to describe the execution on multiple distributed machines. The model that I adopt here is based on the widely employed CONGEST model [103].

The execution of DistCAPA is a sequence of synchronous rounds in which each node performs the following steps:

1. receive an arbitrary amount of messages from other nodes,
2. perform an arbitrary amount of computation,
3. send an arbitrary amount of messages to other nodes.

Every message sent is assumed to be received in the subsequent round, independent of the distance or number of hops between sender and receiver. In addition, nodes and links are assumed to be fault-free, i.e. messages are always sent and received correctly and nodes never crash. Based on this execution model, I will describe DistCAPA in the following section.

7.2 Distributed FCAPP Algorithm

DistCAPA can be divided into three key procedures:

1. node control,
2. DFG satisfaction,
3. LCA reassignment.

In the following parts, I first describe each procedure in more detail and then describe the main procedure executed on every network node during every round. To prevent repetitions in the following, I anticipate that all ties occurring during sorting nodes or DFGs in procedures of DistCAPA are resolved by choosing the node or DFG with the lowest ID.

7.2.1 Node Control

The first goal of each network node is node control, either by finding an LCA that is able to control it or by becoming an LCA itself. The procedure for node control has been inspired by an existing distributed algorithm for solving the capacitated Facility Location Problem (FLP) [62]. However, there is an important difference between node control in FCAPP and facility assignment in FLP. While in the common FLP scenario, the set of nodes consists of two disjoint sets of facilities and clients, the potential hosts in FCAPP can take both the role of an LCA or the role of a normal node looking for an LCA. Ignoring this problem would lead to the chaotic scenario of potential hosts sending and receiving *Node Control* requests at the same time. The best way to illustrate this problem is to look at an example of two neighboring potential hosts $c, d \in C$. Assuming that all potential hosts always look for an LCA and are ready to be an LCA at the same time, both c and d would send node control requests to each other, then confirm to each other and eventually both c and d would become LCAs with c controlling d and d controlling c. Of course, this behavior is not desirable.

DistCAPA resolves this using two different states, the first state representing the search for an LCA and the second state representing the collecting of control requests, and randomization. Each potential host starts in searching state and waits for a certain number of rounds, chosen uniformly at random from $\{0, ..., s_{\text{init}}\}$, before switching to collecting state. Following this, the two states are again switched continuously after a number of rounds chosen

uniformly at random after each switch from I_{switch} until a host is either controlled or has become an LCA.

As a starting point for the node control procedure, the search distance d_{search} of each $v \in V$ is initialized as the minimal distance (in number of hops) from v to its nearest potential host (ignoring itself in case v is a potential host) – information known from the static information. To explain the procedure, I start with a high-level description of its communication phases:

Phase 1: Nodes in searching state without an LCA send control requests to the potential hosts within their search distance (*Node Control* or *Node Control Urgent*, their difference will be elaborated below).

Phase 2: Potential hosts that receive control requests check whether or not the requesting node can be controlled. If yes, then there are two cases. If the respective host is already an LCA it responds with *Connect For Free*, otherwise it responds with *Lock*. If the requesting node cannot be controlled, e.g. because of the host's state or because of missing resources, it responds with *Node Control Reject*.

Phase 3: Each searching node collects its positive replies and selects one of the sending potential hosts, preferring *Connect For Free* over *Lock* replies. Correspondingly, the node sends an acceptance message *Connect For Free Accept* or *Lock Accept* to the selected host and waits for a reply to it. If no positive reply was received, the search distance is increased and new potential hosts are contacted with control requests.

Phase 4: LCAs that receive one or more *Connect For Free Accept* messages or potential hosts that receive one or more *Lock Accept* messages once more check whether or not the requesting nodes can be controlled (for each node individually). This is necessary because additional work could have been accepted in the meantime (see Phase 5). If the check is successful, the corresponding node is accepted and notified (*Node Control Accept*), otherwise it is rejected (*Node Control Reject*).

Phase 5: If a searching node receives a *Node Control Accept* reply to its acceptance message, it stores the sender as its LCA and is done with node control. Otherwise, it goes back to Phase 3 and contacts the next best host from which it already received a positive initial reply (*Connect For Free* or *Lock*). This means that the number of rounds that may pass between *Connect For Free* and *Connect For Free Accept / Lock* and *Lock Accept* is not constant, which explains why the second resource check at a host is required. If there is no such host, the search distance is increased and new potential hosts are contacted with control requests as already seen in Phase 3.

To give an overview, Figure 7.1 summarizes the message exchange that can occur between a searching node $v \in V$ and a potential host (or LCA) within the search distance of v.

To provide more details of the node control procedure, I show the procedure run by a node in searching state during every round in Algorithm 7.1. Ad-

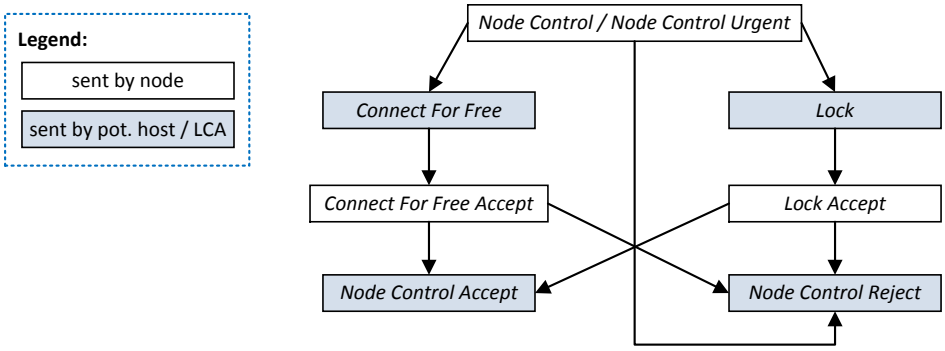


Figure 7.1: Node control: possible messages between a node and a potential host (or LCA)

ditionally, Algorithm 7.2 shows how the messages sent by a node $v \in V$ are processed at potential host (or LCA) $c \in C$.

Algorithm 7.1 Node control procedure of a searching node $v \in V$

if $C_{\text{requested}} = C_{\text{rejected}}$ **and** $|C_{\textbf{requested}}| > 0$ **then**
  d_{search} += 1
N = getNeighborhood(d_{search})
if ControlConformationSent = False **then**
  if $|C_{\text{free}}| > 0$ **then**
    $c = C_{\text{free}}$.pop(getClosestCandidate(C_{free}))
    send *Connect For Free Accept* **to** v
    ControlConformationSent = True
  else if $|C_{\text{lock}}| > 0$ **then**
    $c = C_{\text{lock}}$.pop(getClosestCandidate(C_{lock}))
    send *Lock Accept* **to** v
    ControlConformationSent = True
$N_{\text{remaining}} = (N \cap C) - C_{\text{rejected}}$
if $|N_{\text{remaining}}| > 0$ **then**
  for c **in** $N \cap C$ **if not** c **in** $C_{\text{requested}}$ **do**
    send *Node Control* **to** c
    $C_{\text{requested}}$.append(c)
else
  if self.id **in** C **then**
    self.isLCA = True, self.state = *collecting*, addLCAControl(self.id)
  else
    for c **in** N **if not** c **in** C_{urgent} **do**
      send *Node Control Urgent* **to** c
      C_{urgent}.append(c)

In Algorithm 7.1, node $v \in V$ first increases its search distance d_{search} if each potential host to which a control request has been sent has rejected this request. Next, an acceptance message (Phase 3) is sent if (1) v is not currently waiting for a reply to an acceptance message and (2) there are hosts with positive initial replies available. As described for Phase 3 above, v prefers already

active LCAs that replied with *Connect For Free* (C_{free}) over potential hosts that are not yet LCAs (C_{lock}). In both cases, the closest candidate is selected for sending the corresponding acceptance message.

Finally, the last part of Algorithm 7.1 shows how the control requests are sent, including the difference between *Node Control* or *Node Control Urgent* messages on the side of a searching node. By default, v sends *Node Control* to all potential hosts in its search distance it has not yet contacted. However, v deviates from this default behavior if all potential hosts in its *current* search distance have rejected controlling it. It is very important to note that this can only occur if the search distance was already increased at the beginning of the round *and* this increase did not result in any new potential hosts to contact, i.e. v is rather isolated in the given network. Therefore, v finds itself in an urgent need for an LCA. If that case occurs, which can be seen as a kind of emergency case for preventing uncontrolled nodes, v will become an LCA itself if $v \in C$. Otherwise, it will contact the potential hosts in its search distance again by sending *Node Control Urgent*.

Algorithm 7.2 Node control message processing at host $c \in C$

```
receive Node Control from v or receive Node Control Urgent from v
if self.state = searching and message = Node Control then
  send Node Control Reject to v
else
  if checkLCAcontrol(v) = True then
    if self.isLCA = True then
      send Connect For Free to v
    else
      send Lock to v
  else
    send Node Control Reject to v
```

```
receive Connect For Free Accept from v or receive Lock Accept from v
V_accept.append(v)
```

```
sort V_accept ascending by shortest path
for v in V_accept do
  if checkLCAcontrol(v) = True then
    addLCAcontrol(v)
    send Node Control Accept to v
  else
    send Node Control Reject to v
```

Algorithm 7.2 includes three code fragments. The first part shows how a potential host $c \in C$ handles incoming control requests and reveals the difference between *Node Control* or *Node Control Urgent* messages on a host's side. First, c rejects a control request if it is currently in searching state. However, this only happens for *Node Control* requests. If this is not the case, c checks if is has sufficient resources to control the requesting node. This check is done based on the shortest path from c to the node, which is known from

the static information. Then, c responds correspondingly as already described above (Phase 3). In a nutshell, this means that a *Node Control Urgent* request circumvents the searching state of a potential host and is only rejected if there are insufficient resources. Therefore, this alternative type of control request prevents nodes from being uncontrolled because of too many surrounding potential hosts in searching state.

The second and third part of Algorithm 7.2 show how a potential host treats acceptance messages. During every round, c first stores all nodes that sent such a message and then successively performs the second control check before notifying the corresponding nodes as elaborated further above (Phases 4 and 5), starting with the closest node first.

To conclude the node control procedure, any searching node is in the worst case going to send one *Node Control* and one *Node Control Urgent* message to every potential host in the network. Therefore, the procedure terminates eventually and will result in a complete control structure unless the potential hosts in the network do not have sufficient resources to achieve this.

7.2.2 DFG Satisfaction

Once a node $v \in V$ is controlled, i.e. it has found an LCA or has become an LCA itself, it starts to work towards satisfying its DFGs from F_v. However, it is important to remember that according to the problem statement of FCAPP, every DFG must be satisfied by at most one LCA. If every node would contact LCAs for all of their DFGs, this could hence create a problematic situation for DFGs that are originating from more than one network node. Therefore, for each DFG $x \in F$ originating from nodes $V_x \subseteq V$, only the node from V_x with the lowest ID is allowed to contact LCAs for satisfying x. Since V_x is assumed to be known by every $v \in V_x$ (see Table 7.2) this approach completely avoids additional communication effort between the nodes of such a DFG.

For describing the DFG satisfaction procedure of DistCAPA, I will refer to the node with the lowest ID from $V_x, x \in F$ as the *main node* of x in the following. As in the previous section, I first provide a high-level description based on the message exchange that can occur between a DFG's main node and an LCA (or potential host), depicted in Figure 7.2.

Since the goal of this procedure – finding an LCA for taking a certain task – is fundamentally the same as for the node control procedure, the possible message exchange is also quite similar. However, the procedure as such differs in several aspects. Given a DFG $x \in F$ with main node $v \in V_x$, the procedure aimed towards satisfying x is as follows:

Phase 1: As soon as v is controlled, it contacts all of its LCAs with a request to satisfy x (*Flow Control*), including all required information about x.

Phase 2: An LCA c receiving a request for x checks whether or not it is able to satisfy x. Since it is possible that x is originating from more nodes – not necessarily controlled by c – this check also includes whether or not c can also control the $w \in V_x$ it does not yet control. According to the check's result, c responds positively (*Flow Control Possible*) or negatively (*Flow Control Impossible*).

Phase 3: v collects all positive replies, sorts the senders by shortest distance to itself, and sends a confirmation request (*Flow Confirmation Request*) to the closest candidate and waits for a reply.

Phase 4: Similar to the node control procedure, it is possible that an LCA is no longer able to satisfy x. Therefore, an LCA c receiving a flow confirmation request for x rechecks if satisfying x is still possible. If not, c rejects immediately (*Flow Control Reject*); otherwise it stores x as a potential DFG to be accepted. Whether or not c will actually accept x is decided in a separate procedure that I will describe further below. For now, it is only important to state that if c eventually accepts x, c will send an acceptance message to v and all other $w \in V_x$ (*Flow Control Accept*) and save the DFG satisfaction assignment accordingly.

Phase 5: v receives the reply to the confirmation request. In case of acceptance, x is ultimately satisfied and no further action from v is required regarding x. All $w \in V_x$, $w \neq v$ that receive an acceptance message from c for x and were not yet controlled by c will add c as an LCA.
In case of a rejection, v will go back to Phase 3 if it has remaining LCAs that initially replied with *Flow Control Possible*. Otherwise, all LCAs of v have rejected to satisfy x and thus v starts to send *Flow Control* to other LCAs and potential hosts that are not yet serving as LCAs, essentially repeating Phases 1 to 5 until x is satisfied or until every $c \in C$ has rejected to satisfy x.

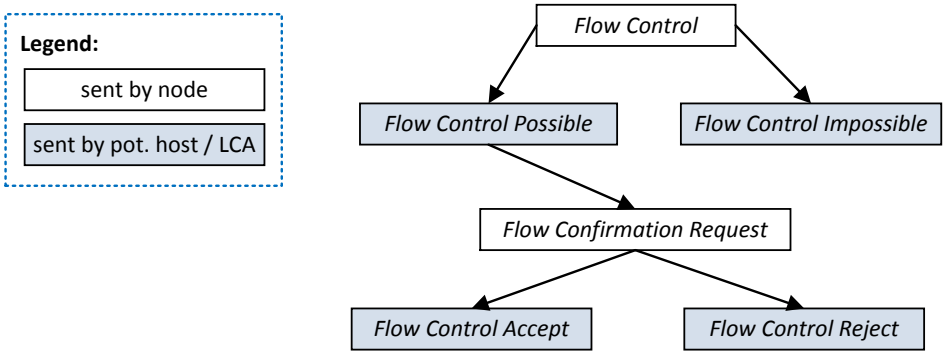


Figure 7.2: DFG satisfaction: possible messages between a node and a potential host (or LCA)

In the latter case, v uses an individual search distance for x that is initialized as the distance (number of hops) to the furthest LCA of v. Exactly as for the node control procedure, this distance is increased once all potential hosts in the current search distance have rejected satisfying x. Because of the procedure's similarity to the node control procedure in Algorithm 7.1, I skip a detailed presentation of the DFG satisfaction procedure running on each node.

Instead, I will elaborate on the procedure that each host $c \in C$ employs to satisfy DFGs (see Phase 4). This procedure (acceptDFGs) is executed during every round by each $c \in C$, with an exception that I will elaborate on in Section 7.2.4. The procedure is shown in Algorithm 7.3.

Algorithm 7.3 acceptDFGs() at host $c \in C$

```
sort F_pot ascending by average distance to all v ∈ V_x, x ∈ F_pot
for x in F_pot do
  if checkDFGsat(x) = True then
    addDFGsat(x)
    for v in V_x do
      send Flow Control Accept to v
  else
    send Flow Control Reject to mainNode(x)
```

After $c \in C$ has first gathered the DFGs that it could potentially satisfy (F_{pot}) instead of accepting them right away (see description of Phase 4 above), the purpose of acceptDFGs is to accept these DFGs in a more structured order. The idea behind this is that satisfying DFGs originating from nodes close to c is preferable, so that other DFGs that can eventually not be satisfied by c can be more easily satisfied by other LCAs. Of course, this is only relevant for the DFGs c is asked to confirm within one execution round. c starts by sorting F_{pot} by average distance to the nodes they are originating from. Then, c successively rechecks whether the current $x \in F_{\text{pot}}$ can be satisfied and if yes, finally assigns it to itself. Rechecking the satisfaction constraints for every DFG is necessary because the resources of c diminish with every previously satisfied DFG. For each DFG, c then sends out the corresponding messages as earlier described for Phase 4.

7.2.3 LCA Reassignment

After additional LCAs have been added during the DFG satisfaction procedure, another crucial task is to deactivate no longer needed LCAs in case of low-load situations. However, due to the myopic view of each network node, DistCAPA cannot consider the load of the entire network like FlexCAPF (Section 6.2.2) did, at least not without significant communication. Therefore, DistCAPA follows a different approach based on a threshold parameter $L_{\text{lowload}} \in (0, 1]$, set globally for all $v \in V$, and a relative load metric for all LCAs:

$$\text{load}(c) = \frac{p_{\text{node}}(c) - p_{\text{rem}}(c)}{p_{\text{node}}(c)} = 1 - \frac{p_{\text{rem}}(c)}{p_{\text{node}}(c)}.$$

During every execution round of DistCAPA, each LCA c checks whether or not its load is above or below L_{lowload}. But in addition, I introduce a parameter h_{lowload} to ensure that c is not currently requested to control a node or to satisfy a DFG. So in total, c regularly checks for the following conditions:

1. $\text{load}(c) \leq L_{\text{lowload}}$,
2. no control request (*Node Control* or *Node Control Urgent*) has been received in the past h_{lowload} rounds,
3. no DFG satisfaction request (*Flow Control*) has been received in the past h_{lowload} rounds.

If all three conditions are fulfilled, c could handover its work to another active LCA. However, having too many LCAs trying to handover their work at the same time, possibly to each other, is not desirable. Assuming c starts a handover attempt and receives a handover request by another LCA d, one possibility could be that c aborts its own handover attempt to take the work of d, e.g. if $\text{load}(c) > \text{load}(d)$. But this can be problematic, for example, if another LCA is already preparing to take the work from c. For this reason, I have decided to not interrupt handover attempts in such a way. Instead, the number of LCAs starting handover attempts concurrently is limited via randomization as in Section 7.2.1. Thus, if the three conditions are fulfilled, a handover is only attempted with a probability of $p_{\text{LL}} \in [0, 1]$.

Given an LCA $c \in C$, a handover attempt, i.e. the procedure for finding another LCA to take the work of c, then works as follows:

Phase 1: c sends a take over request (*Take Over*) including information about its controlled nodes and satisfied DFGs to all other $d \in C$. Simultaneously, c stops to accept any node control or DFG satisfaction requests over the course of the handover attempt.

Phase 2: After receiving a *Take Over* request, each $d \in C$, $d \neq c$ immediately responds with a negative reply (*Take Over Impossible*) if it is either not an LCA or currently attempting a handover itself. But if d is an LCA and not attempting a handover, it checks whether or not it has the available resources to control all nodes currently controlled by c and satisfy all DFGs currently satisfied by c. According to the result, d either sends a positive (*Take Over Possible*) or a negative (*Take Over Impossible*) reply back to c.

Phase 3: c collects the replies and aborts its handover attempt if all replies have been negative. Otherwise, c sends a confirmation request (*Take over Confirmation*) to the closest LCA $d \in C$ that has sent a positive reply.

Phase 4: After receiving a *Take over Confirmation* request, d now rechecks if taking over the work of c is still possible, which might not be the case if d has accepted additional work in the meantime or started a handover attempt itself. If this second check is successful, d then takes the work of c and notifies c (*Take Over Accept*). Further, d notifies all nodes that were previously controlled by c (*Took Over Work*) that it took over from c. Otherwise, d rejects the handover (*Take Over Reject*).

Phase 5: c now either receives a *Take Over Accept* or a *Take Over Reject* reply by d. In the former case, c has succeeded handing over its work and deactivates itself as an LCA. Simultaneously, each $v \in V$ that has so far been controlled by c receives the *Took Over Work* message by d. As a result, each such v will remove c as one of its LCAs, add d (if v was not

already controlled by d) and forward all of its DFGs that were previously satisfied by c to d in the following. In the later case (d rejected c), c will repeat Phase 3 with the next nearest LCA. The handover attempt ends, once an LCA accepts the handover or all $d \in C$, $d \neq c$ have rejected.

By way of illustration, Figure 7.3 summarizes the possible message exchange between an LCA attempting a handover and another target LCA.

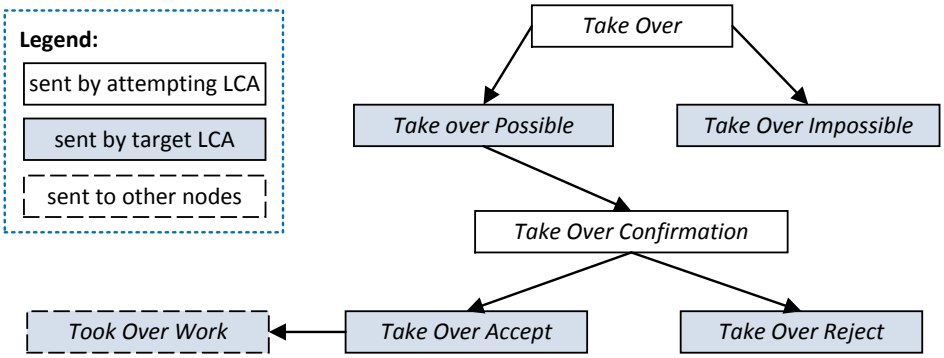


Figure 7.3: LCA reassignment: possible message exchange

Unsurprisingly, the choice of the L_{lowload} parameter has a high influence of the performance of DistCAPA regarding number of used LCAs and will play a major role in the evaluation in Section 7.3.

7.2.4 Node Main Procedure

After describing all three key procedures of DistCAPA in the previous section, I explain how these and other procedures are executed within the main procedure of each node $v \in V$, which is run during every round of DistCAPA. This main procedure is shown in Algorithm 7.4.

During every execution round of DistCAPA a node $v \in V$ first calls the processMessages procedure. This procedure analyzes all received messages and stores the corresponding information adequately, so that this information can later be used within other processes. For example, as seen in Section 7.2.1, a received *Connect For Free* message will cause the message's sender to be stored in C_{free}. Next, v evaluates starting a handover attempt (Section 7.2.3) if and only if v currently is an LCA. Afterwards, if and only if $v \in C$, v will take care of adding accepted nodes (as seen in Algorithm 7.2) and accepting DFGs (as seen in Algorithm 7.3). However, the latter only happens if v did not receive any node control requests in the last h_{sat} rounds, which is the exception that I mentioned in Section 7.2.2. Similar to the $n_{\min}$ value of FlexCAPF, this constraint has the purpose to ensure that nodes do not remain uncontrolled, because potential hosts prematurely devote all of their resources to satisfying DFGs. However, since node control requests no longer appear once that all network nodes are controlled, this constraint is only in effect during initial placement situations, but not during dynamic network operation.

Algorithm 7.4 main procedure of every node $v \in V$

```
processMessages()
// determine if I will attempt to handover my work to another LCA
if self.isLCA = True then
  if self.attemptHandover = False and checkLowLoadConditions() = True
  then
    if random([0,1]) < p_LL then
      self.attemptHandover = True
  if self.attemptHandover = True then
    attemptHandover()
// add confirmed nodes and DFGs
if self.id in C then
  acceptNodes()
  if r_lastControlRequest ≤ self.rounds − h_sat then
    acceptDFGs()
// if I am already controlled, take care of DFG satisfaction
if |self.LCAs| > 0 or self.isLCA = True then
  DFGsatisfaction()
  cleanupLCAcontrols()
else // else make sure that I get controlled
  if self.id in C then
    updateState()
  if self.state = searching then
    nodeControl()
self.rounds += 1
```

Finally, v either takes care of DFG satisfaction (Section 7.2.2) if v is an LCA or has already found an LCA or works towards being controlled (Section 7.2.1) otherwise. In the former case, the DFG satisfaction procedure is followed by cleanupLCAcontrols, which removes no longer required LCA control assignments and which works very similar as the procedure with the same name that was already described in Section 3.5. Accordingly, an LCA $c \in C$ is only removed if and only if (1) c does not satisfy any DFG originating from v and (2) v will still be controlled without having c as an LCA. If an LCA is found to be no longer needed, v will send it a corresponding message (*No Deal*) to let it know that it can remove v from its controlled nodes. In the latter case, if $v \in C$, v first updates its state (collecting or searching) as described in Section 7.2.1. If v is in searching state (which is always the case if $v \notin C$), it will then execute the node control procedure. At last, the main procedure is ended by increasing the round counter.

7.3 Evaluation

In this section, I compare the performance of DistCAPA against the initial placement and continuous reassignment abilities of FlexCAPF. To do this, I

used the same network instances as in Section 5.4 and Section 6.3. Further, for FlexCAPF the RCA requirements have been removed, i.e. the data rate and processing capacity requirements for RCA coordination were set to zero and the RCA latency requirement has been set to a value exceeding the sum of all link delays.

DistCAPA has been implemented using the ComplexNetworkSim package for Python [104]. As before, all evaluation runs have been conducted with different 64-bit random seeds, using the same seed for executing FlexCAPF and DistCAPA on the same network instance. All calculations are executed in single-threaded mode on Intel® Xeon® E5-2695 v3 CPUs running at 2.30 GHz. All plots contain confidence intervals at a 95% confidence level.

During the whole evaluation, I choose the parameters of DistCAPA as follows:

- the upper bound for the number of rounds after which a node $c \in C$ initially changes from searching state to collecting state is set to $s_{\text{init}} = 10$,
- the set from which a potential host $c \in C$ uniformly at random chooses a number of rounds to wait before again switching its state is $I_{\text{switch}} = \{10, 11, ..., 20\}$,
- the number of rounds that must have passed since the last received node control request before accepting DFGs is set to $h_{\text{sat}} = 10$,
- the number of rounds that must have passed since the last received node control or DFG satisfaction request before starting a handover attempt is set to $h_{\text{lowload}} = 10$,
- the probability for a node $c \in C$ to start a handover attempt after fulfilling all conditions is set to $p_{\text{LL}} = 0.1$.

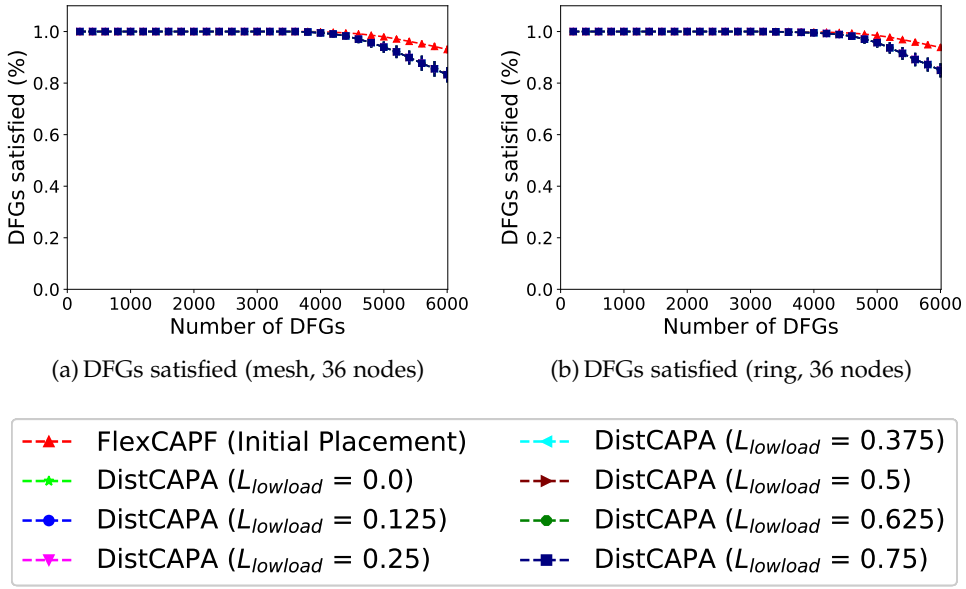
All of these values have been determined during initial evaluation runs. Only the low load threshold parameter L_{lowload} will be varied in the following.

7.3.1 Initial Placement Evaluation

In this part of the evaluation, I compare FlexCAPF against DistCAPA regarding initial CA placement, i.e. for networks without any given CA placement, using networks with 36 or 100 nodes and mesh or ring topology and fixed network states. For DistCAPA, I employ different L_{lowload} values, i.e. 0.0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75.

Since DistCAPA is primarily designed for dynamic network operation, setting the right execution conditions for an initial placement is no straightforward task. After experimenting with several fixed limits for the number of rounds, I concluded that low limits would always bring poor performance results for larger instances while high limits would unnecessarily increase the runtime for smaller instances. For this reason, I decided to successively execute 10 rounds for each instance until (1) the number of controlled nodes did not increase, (2) the number of LCAs did not decrease and (3) the number of satisfied DFGs did not increase over the course of 100 additional execution

rounds. These 100 additional execution rounds have been subtracted in the execution statistics that I am presenting later in this section.



(a) DFGs satisfied (mesh, 36 nodes)

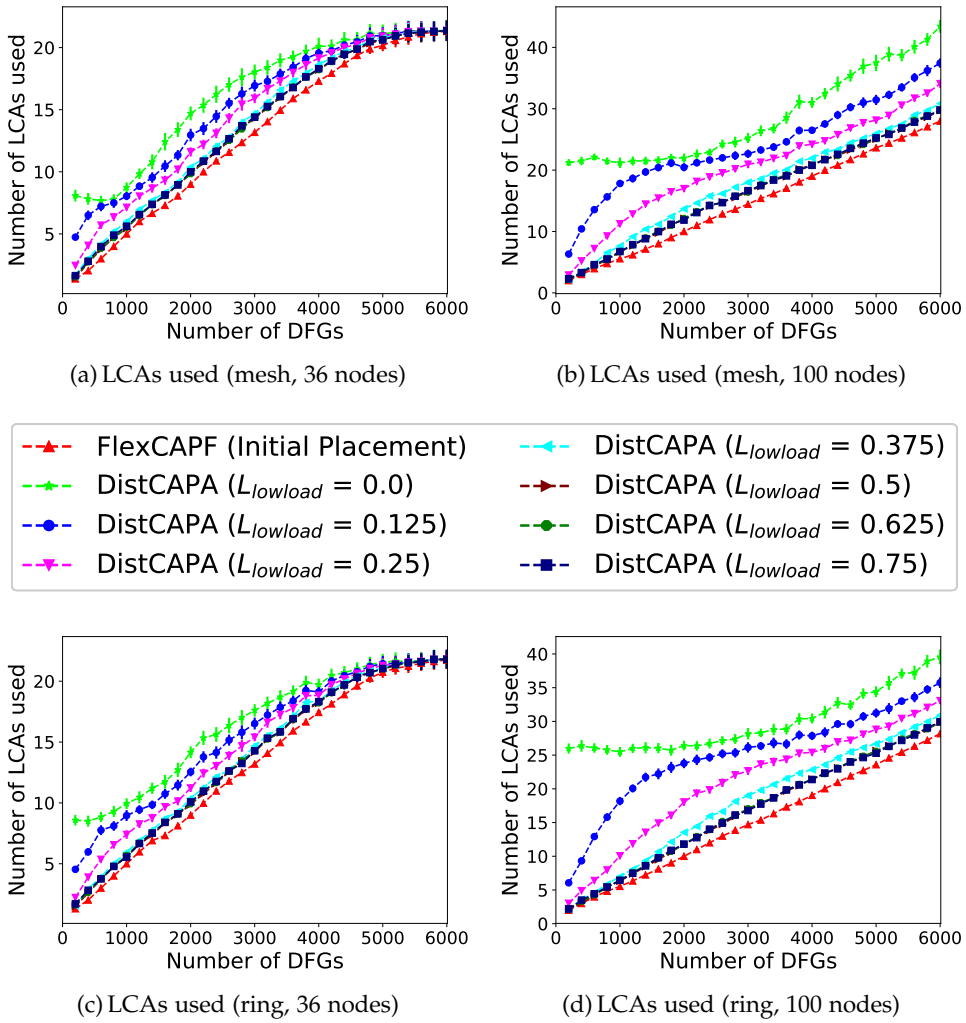
(b) DFGs satisfied (ring, 36 nodes)

Figure 7.4: Evaluation: DFGs satisfied by FlexCAPF and DistCAPA

During this evaluation part, no instance remained unsolved, i.e. DistCAPA produced a complete control structure for all instances. Figure 7.4 shows the percentage of DFGs satisfied for the 36-node networks. In the 100-node networks, all DFGs were satisfied, so these plots were omitted. It can be seen that DistCAPA performs equally for all L_{lowload} values – in fact, the plot lines of DistCAPA for all L_{lowload} values overlap perfectly. But this is not surprising as the L_{lowload} value has no direct influence on the node control and DFG satisfaction procedures. DistCAPA performs slightly worse than FlexCAPF but surprisingly well given its disadvantage in information availability. Only once the network resources are getting scarce, DistCAPA starts to lose some ground but manages to satisfy all DFGs before, just like FlexCAPF. Finally, in this metric, there is no significant difference between the mesh and the ring topologies.

Next, Figure 7.5 shows the number of used LCAs for all four network types. The corresponding plots contain lots of interesting observations. First, which comes as no surprise, DistCAPA with $L_{\text{lowload}} = 0.0$ performs terribly in this metric, since LCAs are never shut down once they are in place. Then, the number of used LCAs visibly decreases with increasing L_{lowload} up to $L_{\text{lowload}} = 0.5$; no further improvements can be seen for $L_{\text{lowload}} \in \{0.625, 0.75\}$. Thinking about it, this is very logical, as finding an LCA to take over work that would need more than half of its resources is naturally very hard. Beforehand, the curves for $L_{\text{lowload}} \in \{0.125, 0.25, 0.375\}$ show an interesting behavior which can be seen best for $L_{\text{lowload}} = 0.125$ in the 100-node topologies. For

both mesh and ring topologies, the curve first raises very quickly before flattening between 1000 and 2000 DFGs. Obviously, $L_{\text{lowload}} = 0.125$ still allows DistCAPA to reassign several LCAs for low networks loads, but with increasing load the number of LCAs whose load falls below that threshold diminishes very quickly.



(a) LCAs used (mesh, 36 nodes)

(b) LCAs used (mesh, 100 nodes)

(c) LCAs used (ring, 36 nodes)

(d) LCAs used (ring, 100 nodes)

Figure 7.5: Evaluation: number of LCAs used by FlexCAPF and DistCAPA

Last but not least, Table 7.4 lists the average number of execution rounds for this evaluation part as described at the beginning of this section. First, it can be observed that the instances with 100 nodes required more execution rounds than the networks with 36 nodes. This is of course not surprising because a larger network results in a higher maximum search distance and/or a larger number of candidates to contact for a certain task. Therefore, the number of rounds needed for the individual procedures of DistCAPA increases. The same reasoning can be applied to the observation that the ring topologies needed slightly more execution rounds than the mesh topologies of the

same size. Since the distance between nodes is always measured in number of hops, a ring topology with fewer links compared to a mesh topology generally increases the search distance accordingly. Finally, the number of execution rounds also grows with increasing L_{lowload} but seems to stagnate at $L_{\text{lowload}} = 0.5$. This observation corresponds to the observation for the number of used LCAs based on L_{lowload} for the previous plot: the lower the L_{lowload} value, the fewer LCAs are able to attempt a handover. For values above 0.5, however, the chance for a successful handover does not increase any further.

Table 7.4: DistCAPA initial placement: execution statistics

Topology:	mesh36	ring36	mesh100	ring100
Needed rounds ($L_{\text{lowload}} = 0.0$):	37.91	39.03	33.53	33.8
Needed rounds ($L_{\text{lowload}} = 0.125$):	48.18	50.27	58.15	58.06
Needed rounds ($L_{\text{lowload}} = 0.25$):	52.49	55.39	65.73	67.83
Needed rounds ($L_{\text{lowload}} = 0.375$):	57.16	58.93	80.29	84.77
Needed rounds ($L_{\text{lowload}} = 0.5$):	58.42	61.0	89.23	99.95
Needed rounds ($L_{\text{lowload}} = 0.625$):	57.57	58.81	88.89	95.26
Needed rounds ($L_{\text{lowload}} = 0.75$):	57.46	59.33	91.65	99.59
Needed rounds (all):	52.74	54.68	72.5	77.04

7.3.2 Dynamic Network Simulation

In this evaluation part, I provide results for DistCAPA obtained from dynamic network simulations with changing network load as already performed for FlexCAPF in Section 6.3. Due to the observations in the previous section, I decided to set DistCAPA's low-load threshold parameter to $L_{\text{lowload}} = 0.5$.

The evaluation scenario for the simulations are very similar to Section 6.3. However, the simulations performed for this section only cover networks with 36 nodes and a simulated time of 2 hours instead of 48 hours due to the higher execution time required for emulating the distributed execution of DistCAPA. Accordingly, I have scaled the load curve shown in Figure 6.2 from 24 hours to one hour to retain the effect of changing network load over time. Still, incoming DFGs are generated using a non-stationary Poisson process with $\lambda = |V| \cdot \text{loadlevel}(t)$, simulated using the thinning method [99], where $\text{loadlevel}(t)$, t being the current simulated time in seconds, denotes the value of the scaled load curve. As in Section 6.3, the duration of a DFG is determined using an exponential variable with an expected DFG duration of 50 seconds, giving an approximated expected number of $50 \cdot |V| \cdot \text{loadlevel}(t^*)$ DFGs in the network at time t^*.

One crucial aspect of this evaluation part is that performing a dynamic simulation with DistCAPA requires to somehow relate the concept of the generated DFG inter-arrival times of the Poisson process for DFG generation

to the synchronous round concept of DistCAPA. I decided to do this by assuming an execution time of $t_{\text{round}} = 0.005$ seconds (5 milliseconds) for every execution round of DistCAPA. For every DFG arrival time t_{curr} generated by the Poisson process, I then predetermine the subsequent DFG arrival time t_{next} and calculate the number of rounds r_{curr} to be executed on the network state following the DFG arrival at t_{curr} by

$$r_{\text{curr}} = \left\lfloor \frac{t_{\text{next}} - t_{\text{curr}}}{t_{\text{round}}} + 0.5 \right\rfloor.$$

This formula corresponds to rounding the inter-arrival time following to t_{curr} divided by t_{round} to the nearest integer value. Based on $|V| = 36$ in this evaluation part, this leads to an expected number of executed rounds of

$$\mathrm{E}(r_{\text{curr}})(t) = (|V| \cdot \text{loadlevel}(t) \cdot t_{\text{round}})^{-1} = \frac{50}{9} \cdot \text{loadlevel}(t)^{-1},$$

which I have depicted in Figure 7.6.

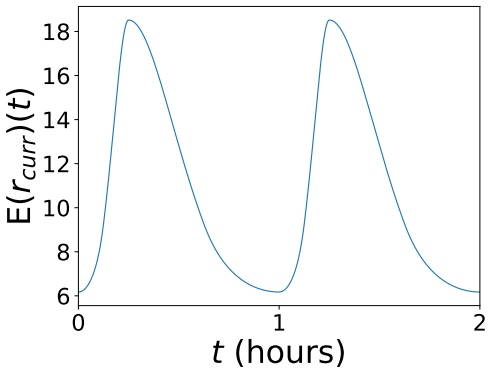
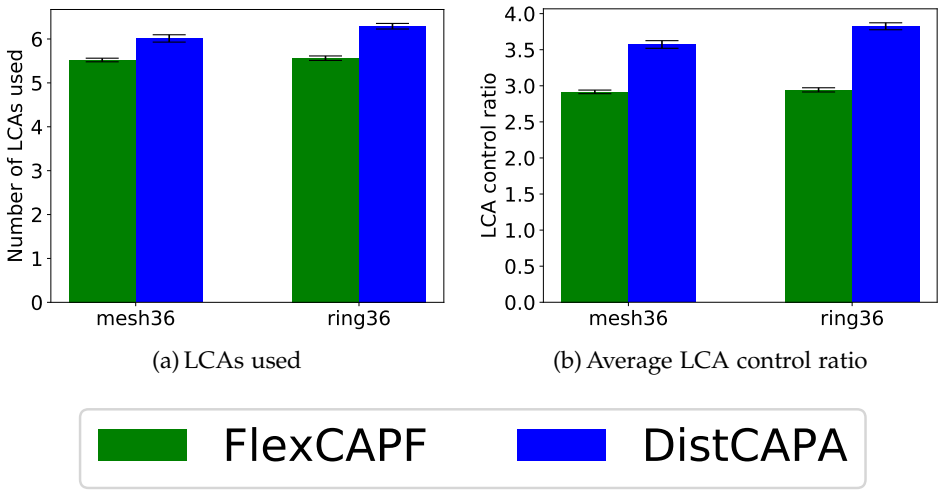


Figure 7.6: Evaluation: expected number of executed rounds

For both DistCAPA and FlexCAPF, the DFG generation is started at $t = -3600$, while the algorithms start to be executed after passing $t = -60$. Thus, the 2 hour network monitoring starting at $t = 0$ is not influenced by a warm-up period.

The results of the simulations can be seen in Figure 7.7 and Table 7.5. Both algorithms satisfied all DFGs during all simulations, for which reason there is no illustration of DFG satisfaction. First, Figure 7.7a shows the number of used LCAs by FlexCAPF and DistCAPA. It can be observed that the difference between FlexCAPF and DistCAPA is similar to the one already seen in Figure 7.5. This reveals that DistCAPA adapts well to changing network load over time, just like FlexCAPF. Then, Figure 7.7b depicts the average control ratio, i.e. number of LCAs per node, established by both algorithms. It can be seen that the placements issued by DistCAPA feature a higher LCA control ratio compared to FlexCAPF. This can be explained by the main node concept used for satisfying DFGs originating from multiple

nodes, through which nodes other than a DFG's main node are sometimes assigned additional LCAs. But the centralized view of FlexCAPF allows to prefer LCAs that already control all or most nodes a DFG is originating from.



(a) LCAs used

(b) Average LCA control ratio

Figure 7.7: Evaluation: FlexCAPF reassignment vs. DistCAPA reassignment

Table 7.5 provides the reassignment statistics of FlexCAPF and DistCAPA. The first aspect that can be noticed is that DistCAPA performed far more reassignments, i.e. changes in the set of LCAs, changed LCA-to-node assignments or changed DFG-to-LCA assignments, than FlexCAPF, which is why providing these metrics in form of a plot as in Section 6.3 would have been very misleading. The reason for this is simple to explain: while FlexCAPF is often able to prevent a reassignment because of its centralized view, DistCAPA consistently operates with a myopic view of the network and willingly approves reassignments if that allows, for example, satisfying an additional DFG. But in particular, the LCA reassignment certainly has a high influence on this statistic. As described in Section 6.2.2, the LCA reassignment of FlexCAPF heavily uses the information about the resource consumption of *all* LCAs in the network and an LCA typically only gives up its work if the resulting network load situation is rather stable subsequently, i.e. no other LCA will end up being very close to its load limit. For DistCAPA, however, LCA reassignment is handled only based on the load of a *single* LCA. Once an LCA attempts a handover, it will give up its work right away once it finds another LCA that is able to take it. But the LCA taking the work could be almost out of resources subsequently, causing a new LCA to be required in its environment soon after.

Another notable observation relates to the differences between mesh and ring topology. Obviously, FlexCAPF changes fewer LCA-to-node assignments and DFG-to-LCA assignments in the ring topology, while DistCAPA affects considerably more changes in it. Again, this is caused by the difference between a centralized and a distributed algorithm. For FlexCAPF, a ring topology implies fewer paths compared to a mesh topology and hence fewer

options to make placement decisions. For DistCAPA, on the contrary, a ring topology instead of a mesh topology means longer paths and hence less information within a given search distances.

Table 7.5: FlexCAPF vs. DistCAPA: reassignment statistics

	FlexCAPF		DistCAPA	
Network:	mesh36	ring36	mesh36	ring36
Reassignments:	19.93	19.73	813.8	1041.9
LCA reassignments:	20.5	20.47	882.23	1134.33
Node reassignments:	148.17	150.77	3477.67	4623.7
DFG reassignments:	829.73	604.0	3314.97	4116.7

7.4 Observations

In this chapter, I have shown a distributed approach to solve FCAPP. The outcome, DistCAPA, turned out to provide very good performance results coming close to the solution quality of my centralized solution FlexCAPF. In particular, DistCAPA was capable of flexibly adapting a given CA placement in reaction to changing network load. So just like FlexCAPF, DistCAPA fulfills the two initial criteria for an FCAPP solution from Section 1.2.

However, because of its disadvantage in information availability, DistCAPA caused significantly more reconfiguration overhead compared to FlexCAPF. Further, the behavior of a centralized algorithm can commonly be better controlled and its execution is typically more convenient. Due to these reasons and due to the fact that my wired backhaul scenario is not a classic scenario where a distributed algorithm would be strictly required, I will consider DistCAPA as a proof of concept solution for FCAPP and continue to further analyze FCAPP based on FlexCAPF in the remainder of my thesis.

8

Flow Processing-aware Control Application Placement with Backbone Extension

All work on FCAPP that I presented so far used the assumption that all backhaul nodes are connected to the backbone network and that this connection provides unlimited data rate and zero latency (see Section 3.2). Of course, this assumption is a simplification and not realistic, but this has no effect on the model as long as the DFGs are not forwarded to the backbone network. In a mobile access network, however, the backbone network provides access to services and constitutes the gateway to the global internet (see Section 2.1.1), which is why it is reasonable to consider scenarios where DFGs need to be forwarded to it.

Further, in contrast to my initial working assumption, only a few nodes within the backhaul network provide the connection to the backbone network in real-world deployments [28]. Therefore, data traffic needs to be routed to these nodes to access the backbone network. These nodes are commonly realized as Traffic Aggregation Points (TAPs) [105]. The main functionality of a TAP is to receive data traffic from many nodes and forward it to the backbone network [106].

As a result, this chapter rectifies my initial assumption for the backbone connection and I describe how FCAPP can be extended employing TAPs to take forwarding DFGs to the backbone network into account. The considered extension has initially been created together with Rasha Al-Naseri over the course of her master thesis [107] under my supervision and has later been considerably refined by myself. After describing the extended problem statement in Section 8.1, I present updated versions of OPT_{ps} and FlexCAPF that take this extension into account in Sections 8.2 and 8.3. Finally, I provide evaluation results for both approaches in Section 8.4 to measure the effect of the backbone extension.

8.1 Problem Statement

Extending FCAPP to incorporate possible backbone connections using TAPs is a rather straightforward task. Compared to the initial FCAPP problem statement from Section 3.2, an additional set $T \subseteq V$ is defined as a first step, which denotes the nodes serving as TAPs and thus provide access to the backbone network. To keep the backbone extension model reasonably simple, the connection to the backbone network with a TAP $t \in T$ as start and end is assumed to require a constant latency of l_{BB} seconds. For illustration, Figure 8.1 depicts an exemplary FCAPP scenario with TAPs providing access to the backbone network.

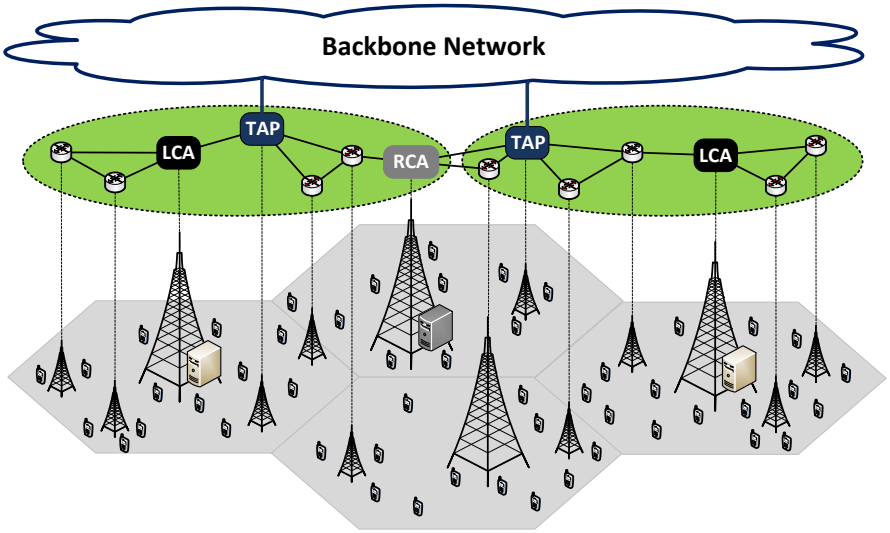


Figure 8.1: FCAPP scenario with TAPs providing access to the backbone network

Regarding the DFGs, it is assumed that not every DFG requires forwarding to the backbone network, i.e. to a TAP. The set of DFGs in need of such a backbone connection is denoted as $F_{\text{BB}} \subseteq F$. So in total, a rather small number of additional input parameters is required, which are summarized in Table 8.1.

Table 8.1: Additional input parameters for FCAPP with backbone extension

$T \subseteq V$	set of TAPs that provide a backbone connection
$F_{\text{BB}} \subseteq F$	set of DFGs requiring forwarding to the backbone network
l_{BB}	delay time in the backbone network

Each DFG $x \in F_{BB}$ is defined just like every other $y \in F \setminus F_{BB}$, but additionally requires being forwarded from its LCA to a TAP. As a result, such a DFG's path delay is increased by the corresponding link delay and by the backbone delay l_{BB}. So in summary, a DFG $x \in F_{BB}$ is said to be *satisfied* by an LCA $c \in C$ if and only if

(1) c controls all nodes v with $W_{x,v} = 1$,
(2) the routing paths from all nodes v with $W_{x,v} = 1$ to c have sufficient resources to each provide a data rate of $b_{\text{flow}}(x)$,
(3) the routing path from c to a TAP $t \in T$ has sufficient resources to provide a data rate of $b_{\text{flow}}(x) \cdot \sum_{v \in V} W_{x,v}$ and
(4) c has sufficient processing capacity to ensure a round trip latency of $l_{\text{flow}}(x)$, taking into account the backbone delay l_{BB} and the path delay of the routing paths from (2) and (3).

It is crucial to emphasize that the round trip latency for a DFG $x \in F_{BB}$ in the network not only includes the time needed for processing x at c and the routing paths delays as seen before, but also the additional backbone delay l_{BB}. Therefore, the time available for handling the required $p_{\text{flow}}(x)$ operations per packet is reduced by both path delays and l_{BB}. Depending on the choice of l_{BB}, this can lead to a substantial increase of required processing capacity per DFG $x \in F_{BB}$ to achieve the required round trip latency $l_{\text{flow}}(x)$.

8.2 Optimization Model with Backbone Extension

In this section, I elaborate how OPT_{ps} from Section 5.2 is extended to incorporate the backbone extension. The resulting optimization model remains an MIQCP and is further referenced as OPT_{BB}. Before I present the additional constraints that need to be ensured, Table 8.2 lists the additional variables used to store the decisions about the LCA-to-TAP assignments.

Table 8.2: Additional MIQCP variables for backbone extension

$\text{TAP}_{t,x} \in \{0,1\}$	determines if $t \in T$ is the TAP of DFGs $x \in F_{\text{BB}}$
$\text{hasTAP}_x \in \{0,1\}$	determines if DFG $x \in F_{\text{BB}}$ has a TAP connection
$h_{c,t,v,w,x} \in \{0,1\}$	determines if $(v,w) \in E$ is included in the routing path from LCA $c \in C$ to TAP $t \in T$ for $x \in F_{\text{BB}}$

As a start, routing path constraints are required for the connections between LCAs and TAPs similar to those already introduced for LCA-to-node and DFG-to-LCA assignments (5.1–5.6).

However, the corresponding h variables require an additional index compared to the previously used routing variables f and g to establish the connection between a routing path and a DFG requiring backbone connection $x \in F_{\text{BB}}$. This increases the number of generated variables and constraints substantially compared to OPT_{ps}. Consequently, constraints (8.1–8.3) guarantee for every LCA-to-TAP routing path that starts at an LCA, ends at a TAP and that ingress and egress are balanced.

$$\sum_{(c,w)\in E} h_{c,t,c,w,x} = \mathrm{Sat}_{c,x} \cdot TAP_{t,x}, \quad \forall c \in C, t \in T, x \in F_{\mathrm{BB}}, c \neq t \tag{8.1}$$

$$\sum_{(v,t)\in E} h_{c,t,v,t,x} = \mathrm{Sat}_{c,x} \cdot \mathrm{TAP}_{t,x}, \quad \forall c \in C, t \in T, x \in F_{\mathrm{BB}}, c \neq t \tag{8.2}$$

$$\sum_{(u,v)\in E} h_{c,t,u,v,x} = \sum_{(v,w)\in E} h_{c,t,v,w,x}, \quad \begin{array}{c}\forall c \in C,\ t \in T,\ v \in V, x \in F_{\mathrm{BB}},\\ c \neq t \neq v\end{array} \tag{8.3}$$

Next, each DFG must be assigned to at most one TAP (8.4) and the corresponding decision variable needs to be set (8.5).

$$\sum_{t\in T} \mathrm{TAP}_{t,x} \leq 1, \quad \forall x \in F_{\mathrm{BB}} \tag{8.4}$$

$$\mathrm{hasTAP}_x = \sum_{t\in T} \mathrm{TAP}_{t,x}, \quad \forall x \in F_{\mathrm{BB}} \tag{8.5}$$

As for the DFG satisfaction, it is now important to distinguish between DFGs that do and ones that do not require a backbone connection. DFGs $x \in F_{\mathrm{BB}}$ are only determined as satisfied if they are assigned to an LCA *and* to a TAP (8.6). It should be noted that in this case, the $\mathrm{Sat}_{c,x}$ variables technically violate their initial definition and are only used for the DFG-to-LCA part of the DFG satisfaction. For all remaining DFGs $x \in F \setminus F_{\mathrm{BB}}$, the satisfaction constraint remains unchanged (8.7).

$$\mathrm{isSat}_x = \sum_{c\in C} \mathrm{Sat}_{c,x} \cdot \mathrm{hasTAP}_x, \quad \forall x \in F_{\mathrm{BB}} \tag{8.6}$$

$$\mathrm{isSat}_x = \sum_{c\in C} \mathrm{Sat}_{c,x}, \quad \forall x \in F \setminus F_{\mathrm{BB}} \tag{8.7}$$

Then, the backhaul link usage for LCA-to-TAP assignments has to be incorporated into the constraints governing the link capacity (8.8).

$$\begin{aligned}\sum_{c\in C, d\in V} f_{c,d,v,w} \cdot \left(b_{\mathrm{LCA}} + \sum_{x\in F} W_{x,d} \cdot \mathrm{Sat}_{c,x} \cdot b_{\mathrm{flow}}(x)\right) + \sum_{c,d\in C} g_{c,d,v,w} \cdot b_{\mathrm{RCA}}\\ + \sum_{c\in C, t\in T, x\in F_{\mathrm{BB}}} h_{c,t,v,w,x} \cdot b_{\mathrm{flow}}(x) \leq b_{\mathrm{cap}}(v,w), \quad \forall (v,w) \in E\end{aligned} \tag{8.8}$$

Similarly, the delay of the LCA-to-TAP routing paths and the backbone delay l_{BB} have to be included into the constraints making sure that the required round trip latency is met for each DFG $x \in F_{\mathrm{BB}}$ (8.9). Because it is required

to also iterate over T, this again results in additional constraints for the optimization model. Again, nothing changes for the DFGs $x \in F \setminus F_{\text{BB}}$ (8.10).

$$p_{c,x}^{\text{DFG}} \cdot \Bigg(l_{\text{flow}}(x) - \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \Big(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v) \Big) - l_{\text{BB}} - \sum_{(v,w)\in E} h_{c,t,v,w,x} \cdot \Big(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v) \Big) \Bigg) \geq \text{Sat}_{c,x} \cdot \text{TAP}_{t,x} \cdot p_{\text{flow}}(x), \quad \forall c \in C, d \in V, t \in T, x \in F_{\text{BB}}, W_{x,d} = 1 \tag{8.9}$$

$$p_{c,x}^{\text{DFG}} \cdot \Bigg(l_{\text{flow}}(x) - \sum_{(v,w)\in E} f_{c,d,v,w} \cdot \Big(l_{\text{cap}}(v,w) + l_{\text{cap}}(w,v) \Big) \Bigg) \geq \text{Sat}_{c,x} \cdot p_{\text{flow}}(x), \quad \forall c \in C, d \in V, x \in F \setminus F_{\text{BB}}, W_{x,d} = 1 \tag{8.10}$$

Finally, it is necessary to add constraints to prevent loops for the LCA-to-TAP connections (8.11–8.12), analogously to the already present loop prevention constraints in OPT_{ps} (5.22–5.25).

$$\sum_{(v,c)\in E} h_{c,t,v,c,x} = 0, \quad \forall c \in C, t \in T, x \in F_{\text{BB}}, c \neq t \tag{8.11}$$

$$\sum_{(t,w)\in E} h_{c,t,t,w,x} = 0, \quad \forall c \in C, t \in T, x \in F_{\text{BB}}, c \neq t \tag{8.12}$$

All further constraints, as well as the objective function, can be adopted unmodified from OPT_{ps}.

8.3 FlexCAPF with Backbone Extension

For extending FlexCAPF to also consider DFGs that require a backbone connection, mainly two aspects have to be covered:

1. When evaluating whether or not an LCA $c \in C$ can satisfy a DFG $f \in F_{\text{BB}}$, it has to be checked simultaneously whether or not an appropriate routing path can be found from c to a TAP.
2. For every DFG $f \in F_{\text{BB}}$, the TAP connection has to be taken into account when checking if the DFG satisfaction constraints are fulfilled and when establishing the corresponding assignment.

To accomplish the first aspect, every occurrence of a DFG satisfaction check (checkDFGsat) followed by adding the corresponding DFG-to-LCA assignment if the check is successful (addDFGsat) – for example in Algorithm 3.5 or Algorithm 6.1 – is replaced by the code fragment shown in Algorithm 8.1.

Algorithm 8.1 Code fragment for an LCA c and a DFG f

```
if f in F_BB then
  TAPpath = findTAP(f, c)
  if path is not None then
    addDFGsat(v, f,TAPpath)
else
  if checkDFGsat(v, f) = True then
    addDFGsat(v, f,None)
```

The included procedure findTAP is presented in Algorithm 8.2. The benefit of creating routing paths from LCAs to TAPs for each DFG individually is that for each DFG a new, best routing path can be determined. Therefore, the procedure first creates weights for each network link, 1 if the link has enough remaining capacity to transfer all required data to a TAP, $|E| + 1$ otherwise. Then, I use the Dijkstra algorithm [108, 109] with these weights to determine the shortest path to each TAP. The paths are then sorted by length and handed over to checkDFGsat. findTAP finally returns the shortest path for which the DFG satisfaction check succeeded, otherwise it returns None.

Algorithm 8.2 findTAP(f, c)

```
for e in E do
  if b_rem(e) ≥ b_flow(f) · Σ_{v∈V} W_{f,v} then
    weight(e) = 1
  else
    weight(e) = |E| + 1
for t in TAPs do
  paths = {dijkstraPath(source=c, target=t,weight=weight) for t in T}
sort paths by length of each path
for p in paths do
  if checkDFGsat(c, f, p) = True then
    return p
return None
```

The second aspect from above has been dealt with by extending the procedures checkDFGsat and addDFGsat. The extended version of checkDFGsat, whose presentation I had omitted in previous chapters, can be seen in Algorithm 8.3. The addDFGsat procedure has been modified analogously, thus I skip a separate representation.

As already mentioned in previous chapters, checkDFGsat is responsible for checking whether or not satisfying a DFG $f \in F$ is possible for an LCA $c \in C$. For each node that f is originating from, the procedure first calculates the link delays of the available LCA-to-node routing paths and additionally takes the LCA-to-TAP path delay and l_{BB} into account if $f \in F_{BB}$. It then checks if the given paths fulfill the latency requirements of f in accordance with constraints (8.9) and (8.10) from OPT_{BB}. While doing so, it also counts how many times

each link $e \in E$ is used and utilizes this to ensure that all used links provide enough data rate compliant with constraint (8.8) from OPT_{BB}.

Algorithm 8.3 checkDFGsat$(f, c, \text{TAPpath})$

```
count = [0 for e in E]
for v in {v in V if W_{f,v} = 1} do
  if TAPpath is not None then // therefore f in F_BB
    path = join(TAPpath,LCApath(c, v))
    delay = 2 · Σ_{e∈path} l_cap(e) + l_BB
  else
    path = LCApath(c, v)
    delay = 2 · Σ_{e∈path} l_cap(e)
  for e in path do
    count[e] += 1
  if delay + p_flow(f)/p_rem(c)) > l_flow(f) then // DFG latency constraint
    return False
for e in {e in E if count[e] > 0} do
  if b_rem(e) < count[e] ·b_flow(f) then // DFG link capacity constraint
    return False
return True
```

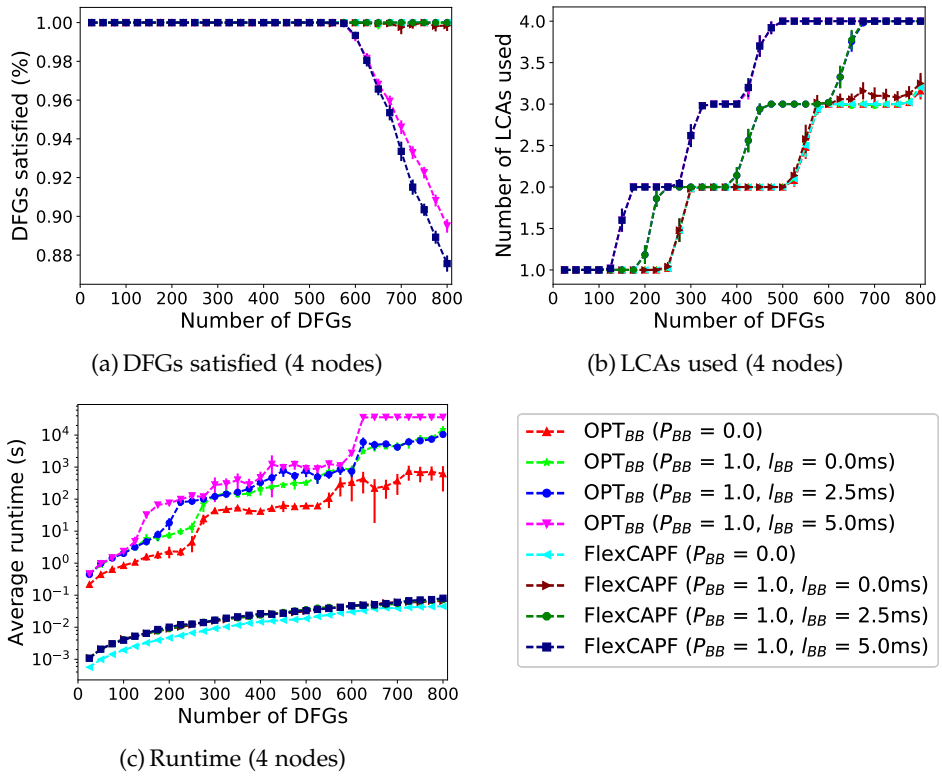
8.4 Evaluation

In this section, I analyze the effects of the presented backbone consideration on the results for FCAPP and on the performance of my solution approaches. The used evaluation scenario is based on the ones already used in previous chapters and only extended by the additional backbone parameters. For all instances, I initialize the set of TAPs T by generating a random sample of $\lceil 0.1 \cdot |V| \rceil$ unique nodes uniformly at random. The set of DFGs requiring backbone connection F_{BB} is initialized by adding every $f \in F$ to F_{BB} with a probability of P_{BB}. For the backbone delay l_{BB}, I consider values of 0.0, 2.5 and 5.0 milliseconds throughout this section. While 0.0 ms is mostly included to see what happens if the backbone connection comes without additional delay (apart from the additional LCA-to-TAP path delays), 2.5 ms and 5.0 ms are chosen as exemplary values, corresponding to a quarter or half of the maximum acceptable round trip latency of most DFGs in my evaluation scenario (see Table 3.3).

As in previous chapters, all evaluation runs have been conducted with different 64-bit random seeds, using the same seed for all parameter settings for FlexCAPF for each network instance. All calculations are executed in single-threaded mode on Intel® Xeon® E5-2695 v3 CPUs running at 2.30 GHz. All plots contain confidence intervals at a 95% confidence level. Unless otherwise noted, every instance was solved using only one RCA.

8.4.1 OPT_{BB} vs. FlexCAPF

In this first evaluation part, I present results obtained from OPT_{BB} for small networks and compare them with results obtained by FlexCAPF's initial placement. Probably due to the large increase of variables and constraints from OPT_{ps} to OPT_{BB}, the optimization model turned out to execute very slowly. I conducted evaluation runs for networks with 4 and 9 nodes and consistently used $P_{BB} = 1.0$, except for the comparison with $P_{BB} = 0.0$. However, it was impossible to solve 9-node networks for DFG counts that would have brought interesting aspects in reasonable time, which is why the results presented here are restricted to the 4-node networks. As in previous chapters, I limited the execution time for each instance to one hour and fed them with the results of FlexCAPF to converge faster, but with many DFGs, most instances with $P_{BB} = 1.0$ terminated without yielding a valid solution within the time limit. I then restarted those instances up to 800 DFGs with a time limit of 10 hours. The results are depicted in Figure 8.2.



(a) DFGs satisfied (4 nodes)

(b) LCAs used (4 nodes)

(c) Runtime (4 nodes)

Figure 8.2: Evaluation: OPT_{BB} vs. FlexCAPF (initial placement)

First, Figure 8.2a shows the percentage of DFGs satisfied. It can be seen that for the instances with $P_{BB} = 1.0$ and $l_{BB} = 5.0$ not all DFGs can be satisfied starting from 575 DFGs. But apart from that, a few other instances with $P_{BB} = 1.0$ and $l_{BB} \in \{0.0, 2.5\}$ left a few DFGs unsatisfied as a result from exhausted links adjacent to the chosen TAP. Then, it is possible to observe in Figure 8.2b, which shows the number of used LCAs, that FlexCAPF with

backbone extension continues to provide optimal or very close to optimal results, as was already seen for FlexCAPF without backbone extension in Chapter 5. At last, the runtimes are depicted in Figure 8.2c. Apart from the well-known runtime difference between optimization model and FlexCAPF, the plot reveals that for FlexCAPF the backbone extension seems to slightly increase execution time. I will also elaborate more on this is the following evaluation part.

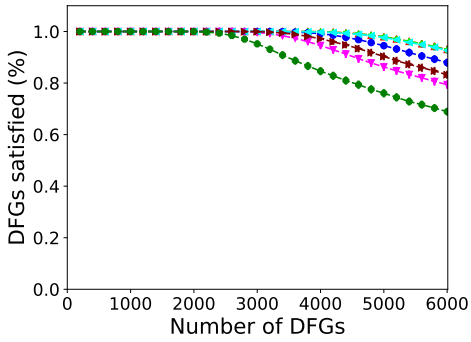
8.4.2 Initial Placement Evaluation

Following the results of OPT_{BB}, I now provide results for the initial placement of FlexCAPF for 36-node networks with mesh and ring topology. This time, I show results for both $P_{\text{BB}} = 0.5$ and $P_{\text{BB}} = 1.0$ as well as $P_{\text{BB}} = 0.0$ for comparison. The results are illustrated in Figure 8.3.

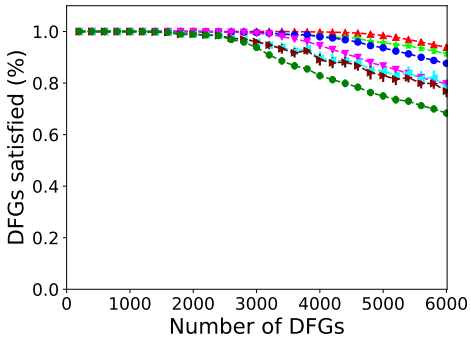
Figures 8.3a and 8.3b offer the percentage of DFGs satisfied for the mesh and ring topologies, along with several interesting observations. First of all, it can be seen that with higher P_{BB} and higher l_{BB}, fewer DFGs are satisfied. This comes as no surprise because the backbone delay l_{BB} is part of a DFG's required round trip latency and thus a DFG $f \in F_{\text{BB}}$ requires more processing capacity from its LCA. Then, for the mesh topologies, the instances with $P_{\text{BB}} \neq 0.0$ and $l_{\text{BB}} = 0.0$ provide the exact same results as the instances with $P_{\text{BB}} = 0.0$. This makes sense because the additional link delays are small and the mesh topologies provide vast link capacity. But in the ring topologies, where link capacity is more limited, there is a significant influence. This is most notable for the instances with $P_{\text{BB}} = 1.0$. The ones with $l_{\text{BB}} = 0.0$ leave almost as many unsatisfied as the ones with $l_{\text{BB}} = 2.5$. Obviously, the link capacities of the links connected to a TAP are exhausted with so many DFGs. The instances with $l_{\text{BB}} = 5.0$ perform similarly to the mesh topologies though, likely because their main bottleneck is still the processing capacity.

Next, Figures 8.3c and 8.3d depict the number of used LCAs. Neither plot reveals new aspects but confirms my conclusions from above. Instances with higher P_{BB} and l_{BB} require more LCAs, which is consistent with the lower DFG satisfaction rates seen before. In addition, Figure 8.3d reveals that the instances that performed worse for the ring topologies in terms of DFG satisfaction also use fewer LCAs in the ring topologies than in the mesh topologies. This verifies that DFGs cannot be satisfied because of missing link capacity, otherwise additional LCAs would be added to satisfy them. In fact, a few of these instances with $P_{\text{BB}} = 1.0$ also resulted in solutions with a second RCA, which is also a straightforward consequence of exhausted link capacities.

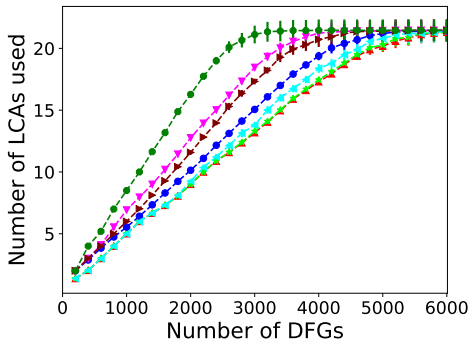
Finally, Figure 8.3e exhibits the runtime results and confirms the observations from Section 8.4.1. The more DFGs require a backbone connection, the higher the runtime of the instances. The runtime increase was practically the same for both mesh and ring topologies, which is why the plot for the latter has been omitted here.



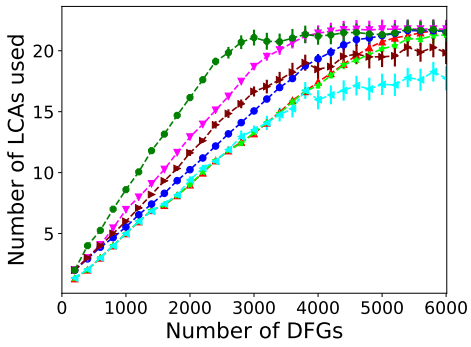
(a) DFGs satisfied (mesh, 36 nodes)



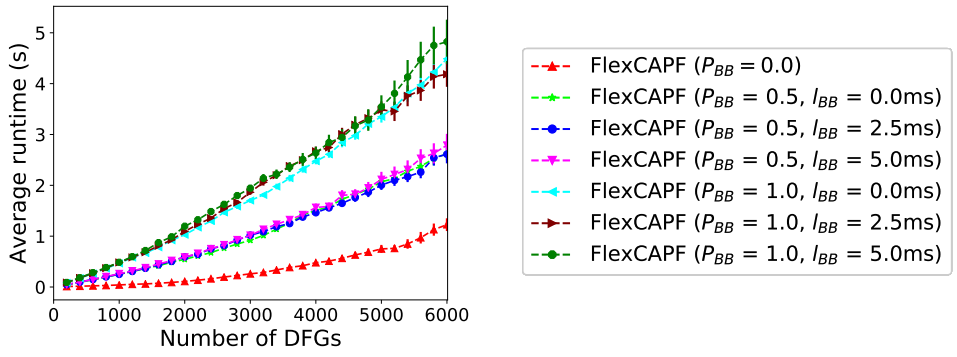
(b) DFGs satisfied (ring, 36 nodes)



(c) LCAs used (mesh, 36 nodes)



(d) LCAs used (ring, 36 nodes)



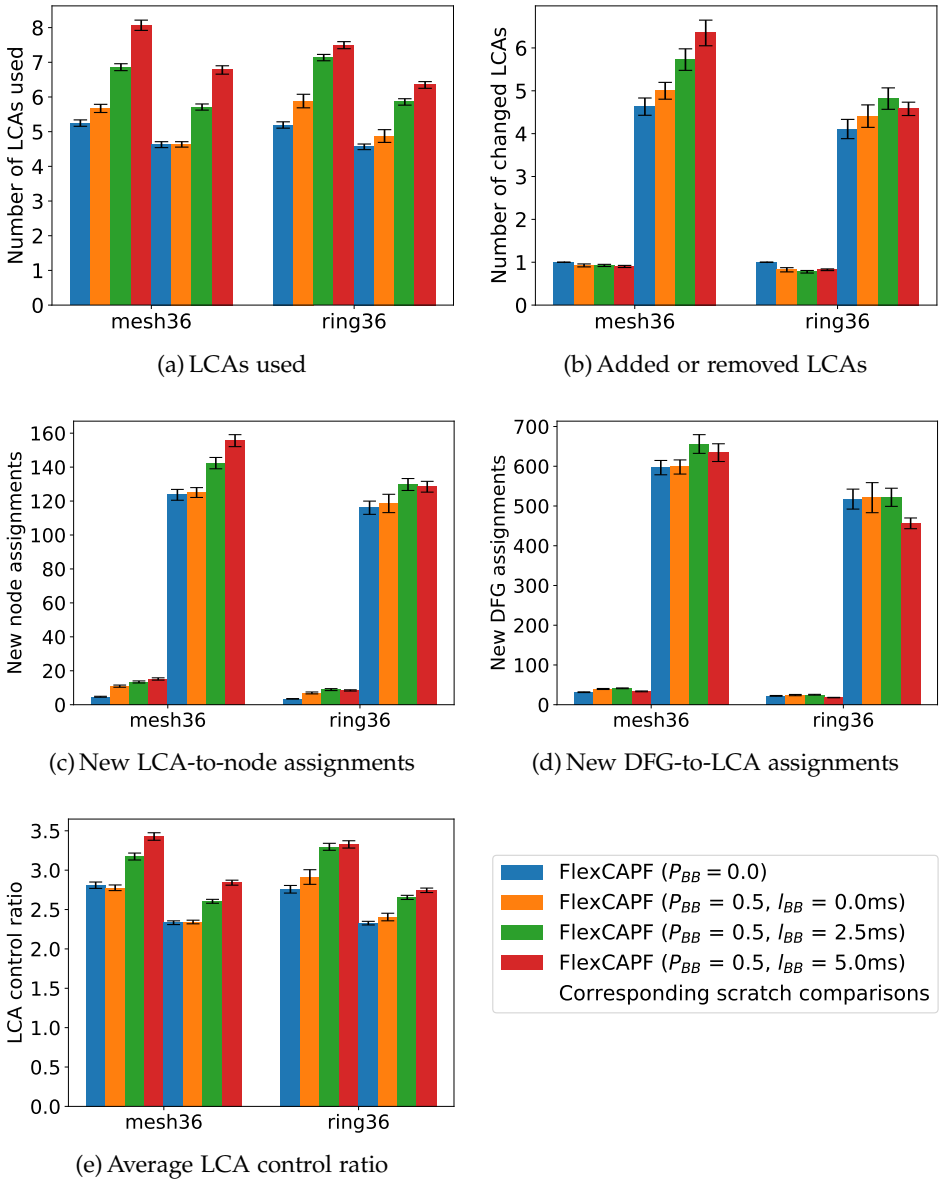
(e) Runtime (mesh, 36 nodes)

Figure 8.3: Evaluation: initial placement with different backbone parameters

8.4.3 Dynamic Network Simulation

In the last part of this evaluation, I provide results for FlexCAPF with backbone extension in a dynamic network simulation. The used simulation parameters are identical to the ones in previous chapters. I also included from-scratch comparisons, like in Section 6.3.2. For this evaluation part, I used instances with $P_{\text{BB}} = 0.5$ only, except for the $P_{\text{BB}} = 0.0$ comparison case of course. The results can be seen in Figure 8.4.

Since many aspects that can be observed in these plots have been covered in the previous two evaluation parts, I will focus on the reassignment aspects hereafter. Figure 8.4a shows the number of LCAs used. One aspect is that flexible reassignment apparently copes with the backbone extension slightly worse compared to the initial placement; the gap between reassignment and from-scratch comparison is larger for the instances with $P_{\mathrm{BB}} = 0.5$ compared to $P_{\mathrm{BB}} = 0.0$. A possible explanation for this minor effect could be that, since DFGs with backbone connection require more resources for satisfaction, reusing existing LCAs compared to placing new ones tailored for the current DFGs has a slightly bigger impact.



(a) LCAs used

(b) Added or removed LCAs

(c) New LCA-to-node assignments

(d) New DFG-to-LCA assignments

(e) Average LCA control ratio

Figure 8.4: Evaluation: FlexCAPF reassignment with different backbone parameters

Then, the number of added or removed LCAs and new DFG-to-LCA assignments compared to the previous placement (Figures 8.4b and 8.4d) appear unaffected by the backbone extension. However, the number of new LCA-to-node assignments is visibly higher for the instances with $P_{BB} = 0.5$. This can be explained similarly as before. With increased resource requirements of DFGs $x \in F_{BB}$, it is more likely that LCAs cannot satisfy new DFGs originating from their controlled nodes. Hence, new LCA-to-node assignments have to be created. This thought is confirmed by the average number of LCAs per node shown in Figure 8.4e, which reveals that the control ratio is higher for instances with $P_{BB} = 0.5$. At last, the relative runtime increase for the flexible reassignment was the same as for the initial placement, hence I skip the corresponding plot.

8.5 Observations

In this chapter, I have described how FCAPP can be extended to employ TAPs to forward DFGs to the backbone network. It turned out that depending on the chosen backbone parameters, this extension has a significant influence on the results obtained for FCAPP. Still, my solution approaches were able to handle the extension very well and, after all, the deviations between previously seen results could generally be explained by increased resource requirements of DFGs in need of a backbone connection.

However, whether or not DFGs need to be forwarded to the backbone network is very dependent on the type of considered DFG, or in other words, the scenario modeled using my concept of DFGs. This naturally raises the question how FCAPP would perform for evaluation scenarios completely different to the ones used so far. Therefore, I will tackle this question in the following chapter. As for the backbone extension, I will consider it to be a special use case that is now covered and whose influence is now known, so I will refrain from employing DFGs requiring a backbone connection in the remainder of the thesis.

9

CoMP-based Evaluation of Flow Processing-aware Control Application Placement

In the preceding chapters, I have studied multiple variations of FCAPP, I have presented various solution approaches for those and I have evaluated them using fixed networks and dynamic network simulations. But one significant aspect has remained unchanged so far: the DFG evaluation scenario introduced in Section 3.6.1 (that will be referred to as *generic scenario* in the following) has been used consistently throughout all evaluation parts in my thesis. Up to this point, doing this was convenient and allowed to analyze the effects of my considered modifications to the FCAPP problem statement (such as the backbone extension in the previous chapter) or to compare different solution approaches.

The evaluation results obtained in the previous chapter revealed how much the results for FCAPP could already be affected by an additional backbone delay parameter. This naturally raises the question how FCAPP, or in particular FlexCAPF, would perform for other types of DFGs, possibly very different from those used in the generic scenario. For this sake, Coordinated Multi-Point (CoMP) techniques present themselves as a base for an alternative evaluation scenario. In previous chapters, CoMP has been mentioned and been used as a motivation for my work several times, for example for the definition of DFGs in Section 3.2. As a result, this chapter introduces a new DFG evaluation scenario based on CoMP transmission/reception (Section 2.1.2) in Section 9.1 and then analyzes its influence on FCAPP in Section 9.2.

9.1 Scenario Description

There are a number of different techniques summarized under the umbrella term CoMP, achieving different gains and having different requirements for the backhaul network. For this CoMP evaluation scenario, I will focus on Joint Processing (JP) and Joint Scheduling (JS)/Joint Beamforming (JB), which I introduced in Section 2.1.2.

Each DFG is either of type JP or of type JS/JB, as shown in Table 9.1. I have extrapolated the data for these types based on a deliverable of the NGMN Alliance RAN evolution project *CoMP evaluation and enhancement* [110]. As already done in Section 3.6.1, I employ a factor $\mathrm{op}(x)$ as a base for choosing the $p_{\mathrm{flow}}(x)$ parameters for each DFG, which describes the operational overhead that arises during data processing. For all ranges listed in Table 9.1, the values have been chosen uniformly at random.

Table 9.1: CoMP scenario: DFG types

type	probability	b_{flow}	l_{flow}	$\mathrm{op}(x)$
JP	0.5	15 to 20 Mbit/s	2 to 4 ms	10^7
JS/JB	0.5	5 to 10 Mbit/s	2 to 4 ms	$5 \cdot 10^6$

Based on this, the processing capacity requested by DFG x is determined by

$$p_{\mathrm{flow}}(x) = \mathrm{op}(x) \cdot \sum_{v \in V} W_{f,v}.$$

Just as for the generic scenario, I pick random points in the grid and use the GreenTouch connectivity model [92] to assign DFGs to nodes. I then connect every DFG with up to three BSs providing the highest Signal to Noise Ratios (SNRs) (or chosen uniformly at random in the very unlikely case of a tie). This choice is compliant with existing CoMP studies [111, 112], where three cooperating nodes with the highest SNR give the best results. However, to add some variation to the number of data flows per DFG I decided to add the third best one if and only if its SNR surpasses a threshold of 0.0 dB.

To give an impression of the number of data flows per DFG produced by this evaluation scenario, I have generated 1000 DFGs for each network used in Section 9.2. Rounded to one decimal place, 32.2% of the DFGs had two data flows and 67.8% had three data flows, giving an average number of 2.678 data flows per DFG. This constitutes a significant difference compared to the generic scenario, for which an identical sample analysis gave a result of 1.346 data flows per DFG in Section 3.6.1. Table 9.2 summarizes this and other key differences between the new CoMP scenario and the generic scenario.

Table 9.2: Key differences between generic and CoMP scenario

	generic scenario	CoMP scenario
data flows per DFG (sample)	1.346	2.678
b_{flow} (expected value)	~ 3 Mbit/s	12.5 Mbit/s
l_{flow} (expected value)	14 ms	3 ms
p_{flow} (expected value/sample)	$\sim 1.35 \cdot 10^7$	$\sim 2 \cdot 10^7$

In total, the presented CoMP scenario appears to be substantially more challenging compared to the previously used generic scenario since the produced DFGs require more data rate, more processing capacity and shorter round trip delays. More precisely, with the processing demands p_{flow} being increased by a factor of approximately 1.5 and the maximum round trip latency l_{flow} being decreased by a factor of $\frac{14}{3}$, each CoMP DFG will require approximately $1.5 \cdot \frac{14}{3} = 7$ times more processing capacity from an LCA to be satisfied. But of course, this is just a rough estimate that ignores the link delays. In the following section, I will evaluate how the results of FCAPP are influenced by the CoMP scenario in practice.

9.2 Evaluation

In this section, I analyze the results of FCAPP based on the CoMP evaluation scenario and discuss the observed effect, focusing on possibly different behavior compared to the previously used generic scenario. First, I compare the results of OPT_{ps} and FlexCAPF in Section 9.2.1 to see if the different characteristics of the CoMP scenario have an influence on the gap between both solution approaches. Next, I evaluate FlexCAPF for larger networks with fixed state in Section 9.2.2. Last, I present the results of dynamic network simulation runs in Section 9.2.3.

Apart from the CoMP scenario for generating DFGs, I use the same evaluation parameters and topologies as described in Sections 3.6.1, 5.4.1 and 6.3.1 – except for a few differences that I will outline in the following subsections. As before, all calculations are executed in single-threaded mode on Intel® Xeon® E5-2695 v3 CPUs running at 2.30 GHz and all plots contain confidence intervals at a 95% confidence level.

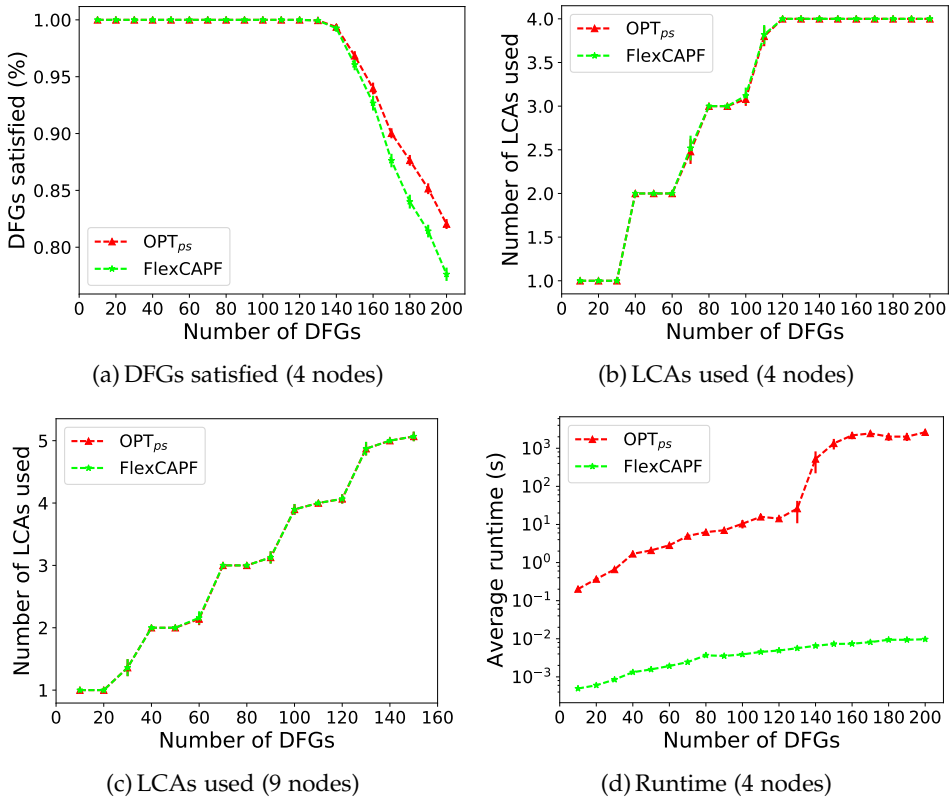
9.2.1 OPT_{ps} vs. FlexCAPF

To compare OPT_{ps} and FlexCAPF, I have generated mesh topologies with 4 and with 9 nodes, where each node is a potential host (to not unnecessarily reduce the solution space of the small networks). The instances are generated with multiples of 10 DFGs (i.e. 10, 20, 30,...) as long as they could still be solved within a 1-hour time limit. As in previous evaluation parts featuring optimization models, I enhanced OPT_{ps} by feeding it with the results of FlexCAPF for reducing the search space (as described in Section 3.6.2).

The results of these evaluation runs can be seen in Figure 9.1. Figure 9.1a shows the percentage of DFGs satisfied in the 4-node networks and allows to make two very interesting observations. First of all, it can be seen that the DFG satisfaction drops below 100% at around 120 DFGs, which confirms that satisfying CoMP DFGs requires way more network resources than satisfying the generic DFGs in previous chapters. As a reminder, both OPT_{ps} and $\text{GreedyFCAPA}_{\text{ps}}$ (which corresponds to the initial placement functionality of

FlexCAPF) were able to satisfy all generic DFGs for up to 1100 DFGs in the network in the corresponding evaluation in Section 5.4.2. This means that the number of DFGs satisfied dropped by a factor of $\frac{1100}{120} \approx 9.167$, i.e. by more than the factor of 7 estimated earlier. This difference can be explained by the previously neglected link delays on the one hand and with additionally longer routing paths for CoMP DFGs due to more data flows per DFG (all of which have to be routed to the same LCA) on the other hand.

Then, it can be noted that FlexCAPF continues to provide great results in direct comparison with OPT_{ps}, despite the vastly different DFG types. FlexCAPF reliably satisfies all DFGs as long as OPT_{ps} does so and then continues to satisfy as many DFGs as OPT_{ps} does before a small gap starts to appear at around 150 DFGs. At 200 DFGs, FlexCAPF still satisfies only 5% less DFGs than OPT_{ps}. Given that the new CoMP scenario includes around twice as many data flows per DFG than the former generic scenario, it is definitely a great result that FlexCAPF features such a stable behavior. The same plot for the 9-node networks has been omitted because all DFGs were satisfied.



(a) DFGs satisfied (4 nodes)

(b) LCAs used (4 nodes)

(c) LCAs used (9 nodes)

(d) Runtime (4 nodes)

Figure 9.1: Evaluation: OPT_{ps} vs. FlexCAPF (initial placement) for CoMP scenario

The pleasant conclusion about FlexCAPF from above is confirmed by Figures 9.1b and 9.1c, which depict the number of used LCAs in the 4-node and 9-node networks. As in previous chapters, FlexCAPF uses the optimal number of LCAs. Also, as for the corresponding evaluation part using the generic

DFG scenario, the number of used LCAs scales in a very obvious way with more DFGs in the network.

Last but not least, Figure 9.1d illustrates the execution times of OPT_{ps} and FlexCAPF for the 4-node networks. I have omitted the corresponding plot for the 9-node networks, as it would not bring any additional insight. Again, two interesting aspects can be observed. First, it can be seen that the execution time of OPT_{ps} increases drastically once no longer all DFGs can be satisfied – a similar correlation could already be seen in previous evaluation parts but not in such a clear manner. Apparently, the increased number of data flows per DFG causes more interdependencies for OPT_{ps} to deal with to satisfy as many DFGs as possible. The other observation concerns FlexCAPF. By comparing Figure 9.1d with Figure 5.3d (which shows the runtime of the corresponding evaluation using the generic scenario), it can be observed that both feature an almost identical curve for up to 200 DFGs. I conclude that the runtime of FlexCAPF is mainly affected by the number of DFGs in the network – not by the number of data flows included in all DFGs in the network. This is another pleasant result because it reveals again that FlexCAPF shows a stable behavior for more difficult DFG types.

9.2.2 Initial Placement in Larger Networks

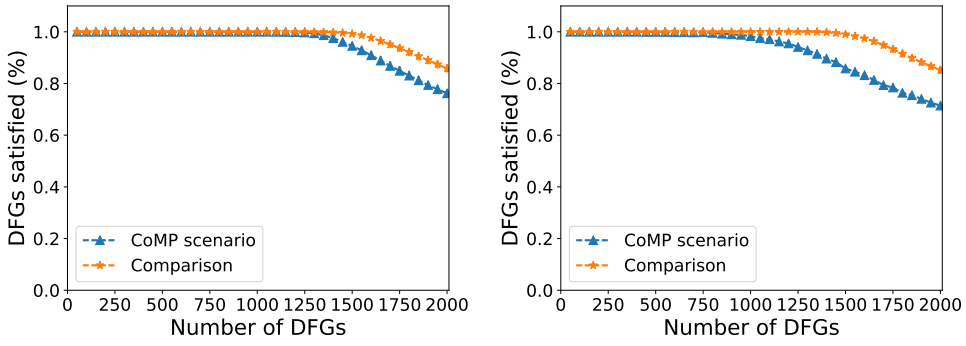
After looking at OPT_{ps} and FlexCAPF for smaller networks in the previous section, I am now evaluating FlexCAPF using the CoMP scenario in larger networks with fixed network state. To do this, I use the same 36-node and 100-node node networks with mesh or ring topology already used in previous chapters with multiples of 50 DFGs.

While it would be desirable to compare the CoMP scenario and the generic scenario directly, such a comparison is not reasonable because of the vast differences between the two scenarios. Instead, I decided to run every instance a second time, but considering each data flow of each DFG individually, i.e. each data flow of a DFG is considered as an individual DFG with only one data flow and can be assigned to different LCAs. The DFG's required processing $p_{\text{flow}}(f)$ is shared equally between the single data flows. The required data rate $b_{\text{flow}}(f)$ and the required round trip latency $l_{\text{flow}}(f)$, which already apply to each individual data flow of every DFG anyways, remain untouched.

The intention of doing so is twofold: on the one hand, I consider this a viable comparison case since it features one data flow per DFG (and is hence not that far away from the generic scenario) while maintaining the same theoretical resource requirements, which enables a direct comparison. Arguably, this comparison for the CoMP scenario is just as suitable to reveal behavioral differences compared to the generic scenario as any possible attempt to scale generic DFGs so that their resource requirements correspond more to those of CoMP DFGs. On the other hand, it is a perfect opportunity to determine the effect of my requirement that all data flows of a DFG have to be jointly

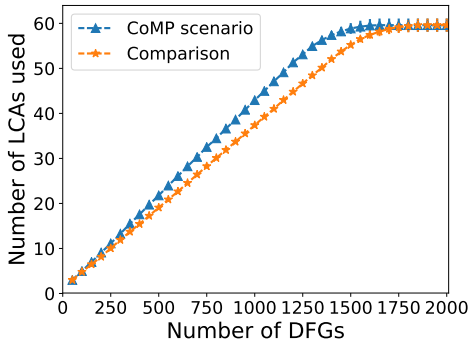
assigned to one LCA, i.e. to evaluate one of the major conceptual decisions of my thesis.

The most interesting results of this evaluation are illustrated in Figure 9.2. This time, I have omitted the results of the 36-node networks, because they exhibited the same behavior as the 100-node networks, albeit slightly less prominent. At first, Figures 9.2a and 9.2b show the percentage of DFGs satisfied in the network. It can be perceived that in both plots, the comparison scenario yields more satisfied DFGs than the default CoMP scenario. Moreover, while the comparison scenario shows no visible difference between mesh and ring topologies, there is a notable difference for the CoMP scenario that has fewer DFGs satisfied in the ring topology. In particular, some individual DFGs already remain unsatisfied with 750 DFGs in the network, while the same only occurs in the mesh networks at around 1250 DFGs.

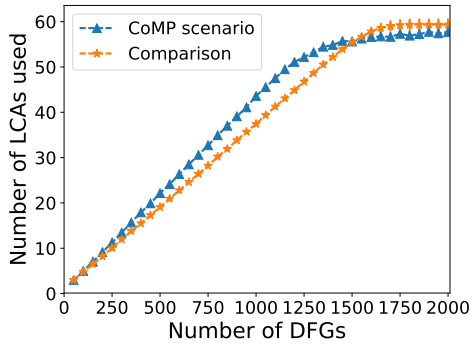


(a) DFGs satisfied (mesh, 100 nodes)

(b) DFGs satisfied (ring, 100 nodes)



(c) LCAs used (mesh, 100 nodes)



(d) LCAs used (ring, 100 nodes)

Figure 9.2: Evaluation: FlexCAPF initial placement for CoMP scenario

The first conclusion to be drawn from this is that the essential requirement of all data flows of a DFG to be jointly processed at the same LCA obviously has a major influence on the obtained results and generally makes it harder for DFGs to be satisfied. With exhausted network resources, some highly demanding CoMP DFGs can no longer be satisfied. But treated individually, some of them still can be satisfied, which is why the comparison satisfies

more DFGs. This effect is particularly noticeable in the ring topologies, where fewer backhaul links can lead to very long routing paths compared to a mesh network and especially compared to data flows being processed individually.

To illustrate this, Figure 9.3 depicts a 36-node ring topology and three nodes a DFG could originate from are highlighted. For this particular topology, these three nodes represent a worst case, so that any LCA satisfying the corresponding DFG would be at least five hops away from one of the nodes. As a result, the path delay increases and an LCA needs to devote more processing capacity to fulfill the required round trip latency of the DFG. In a mesh topology, however, all three nodes would be interconnected and commonly, an LCA significantly closer to all nodes can be found.

From an operator's perspective, this means that a decision has to be made between additional capital expenditure for additional backhaul links or additional operational costs for more required processing capacity and thus more active LCAs. In this regard, FlexCAPF constitutes a powerful tool as it allows to characterize the practical benefits of envisioned network extensions via simulation before making a costly, final deployment decision.

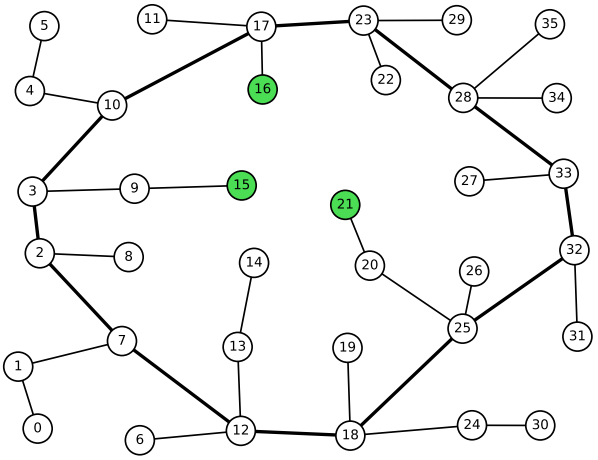


Figure 9.3: Ring topology: worst case example for a DFG originating from 3 nodes

Coming back to the evaluation results in Figure 9.2, Figures 9.2c and 9.2d show the number of LCAs used in the mesh and ring topologies. As expected, it can be seen that the CoMP scenario requires more LCAs than the comparison case. Naturally, the same explanation that I elaborated already extensively based on the DFG satisfaction plots applies here as well. However, it can additionally be noticed that with exhausted network resources in the ring topology, FlexCAPF even stops to utilize all LCAs for the CoMP scenario. Since this can only happen if FlexCAPF is not able to satisfy additional DFGs by doing so, this indicates that the higher data rate requirements of the CoMP scenario are exhausting the backhaul links for many DFGs – in contrast to the mesh topologies. At last, I have skipped runtime results for this evaluation part because they showed the same similarity to previous evaluation parts as I already described in Section 9.2.1.

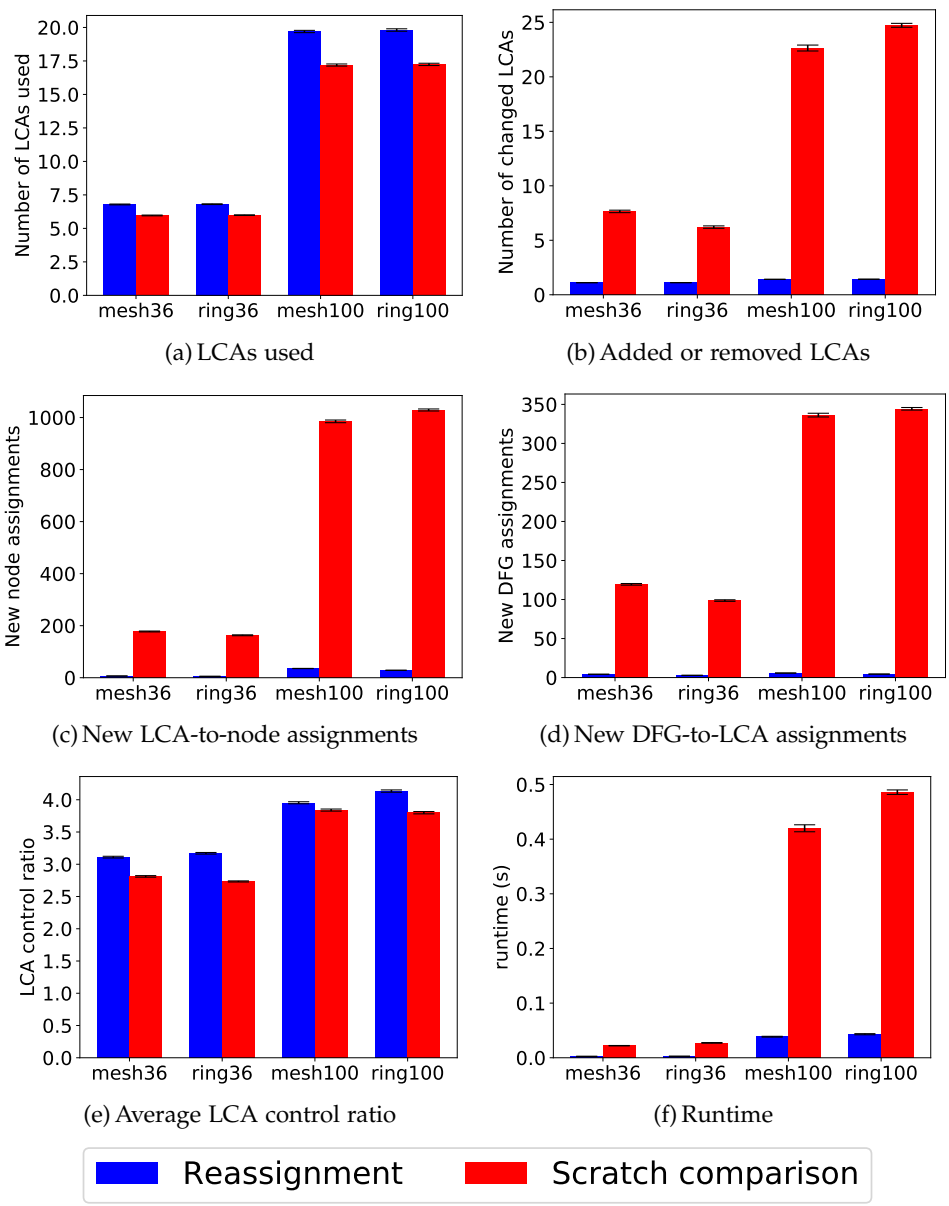
9.2.3 Dynamic Network Simulation

In this final evaluation part, I present results from dynamic network simulation runs for the CoMP scenario to determine the effect of the new scenario on the flexible reassignment of FlexCAPF. As in Section 6.3, I have simulated the network operation using FlexCAPF over the course of 48 simulated hours on the same four network topologies with 36 or 100 nodes and mesh or ring topology (Figure 6.3). Further, I again perform a from-scratch comparison on an empty copy of the current network each time the set of used CAs is modified in the course of the simulation. The remaining evaluation scenario is also identical to Section 6.3, except that the intensity of the non-stationary Poisson process to generate DFGs is set to $\lambda = 0.125 \cdot |V| \cdot \text{loadlevel}(t)$, i.e. scaled by a factor of 0.125 to compensate for the higher resource demand of the CoMP scenario.

The results of the simulation runs are depicted in Figure 9.4. As in all simulations all DFGs were consistently satisfied, there is no illustration of DFG satisfaction. Since I have already elaborated on the general reassignment characteristics in previous chapters, I will focus on the key differences from the reassignment results based on the generic scenario from Section 6.3.2 in the following.

First, Figures 9.4a and 9.4b display the average number of LCAs used and the average number of added or removed LCAs compared to the previous placement. For both plots there is no notable difference compared to the results from Section 6.3.2: the flexible reassignment uses only very few LCAs more on average and affects the set of active LCAs far less. Then, Figure 9.4c and Figure 9.4d show the average number of new LCA-to-node and DFG-to-LCA assignments. The behavior is again roughly the same as in Section 6.3.2. But it can be noticed that the absolute numbers of LCA-to-node assignments are higher, while the absolute numbers of DFG-to-LCA assignments are lower. This can, however, be explained since the CoMP scenario features fewer DFGs but with more data flows per DFG. Therefore, it is less likely that there is already an LCA controlling all nodes a new DFG is originating from, so that new LCA-to-node assignments need to be established.

This is also expected to result in more LCAs per node on average, which is confirmed by Figure 9.4e. The control ratio shown in this plot is indeed slightly bigger compared to the results from Section 6.3.2. Surprisingly, the gap between reassignment and from-scratch comparison is smaller in turn. A possible explanation for this is that even the from-scratch placement has to create new LCA-to-node assignments more often to satisfy DFGs due to more data flows per DFG. At last, the runtimes illustrated in Figure 9.4f are lower compared to those from Section 6.3.2 due to fewer DFGs in the network, but the relation between reassignment and from-scratch comparison also corresponds to the one already seen. In total, the flexible reassignment seems to exhibit a very stable behavior just as the initial placement in the previous section.



(a) LCAs used (b) Added or removed LCAs

(c) New LCA-to-node assignments (d) New DFG-to-LCA assignments

(e) Average LCA control ratio (f) Runtime

Figure 9.4: Evaluation: FlexCAPF reassignment with CoMP scenario

Table 9.3 gives a more fine-grained overview of the number of reassignment runs and the average runtimes recorded during the simulations for each topology, including separate numbers for reassignments due to needing an additional LCA (HL) and due to low-load situations (LL) for the flexible reassignment.

As a matter of fact, the table reveals a considerable difference compared to the numbers seen before in Section 6.3.2. Apparently, the CoMP scenario required significantly more reassignments, about 10 to 15 times the ones needed for the generic scenario. Likely, this is caused by the characteristic of the CoMP

scenario having few but highly demanding DFGs. As a result, every expired DFG frees and every new DFG requires considerably more resources than in the generic scenario, which can more easily cause a low-load reassignment, soon followed by adding an LCA again once new DFGs arrive in the network.

Table 9.3: Simulation runtime statistics

Network:	mesh36	ring36	mesh100	ring100
Average number of runs (total):	640.6	640.27	1022.53	967.2
Average number of runs (HL):	355.77	356.7	725.27	690.23
Average number of runs (LL):	284.83	283.57	297.27	276.97
Average runtime (reass.):	0.003 s	0.003 s	0.039 s	0.043 s
Average runtime (HL):	0.002 s	0.002 s	0.018 s	0.021 s
Average runtime (LL):	0.004 s	0.004 s	0.089 s	0.098 s
Average runtime (scratch):	0.022 s	0.027 s	0.42 s	0.486 s

But while this behavior is certainly not desirable, it is not a general flaw of FlexCAPF and can be corrected by adjusting the parameters of FlexCAPF appropriately to the characteristics of the CoMP scenario. In Section 6.2.2, I introduced a parameter $L_{\text{lowload}} \in (0, 1]$, which steers the intensity of the low-load handling. This parameter had been set to 0.9 in Section 6.3.2, provided good results for the generic scenario and was simply adopted for this evaluation part. But for a scenario like CoMP, where individual DFGs require more resources, a lower value would be more appropriate to avoid too aggressive short-term estimates that do not hold up in the long run. To confirm this hypothesis, I reran all simulations using $L_{\text{lowload}} = 0.8$. The relevant figures can be seen in Table 9.4.

Table 9.4: Simulation runtime statistics ($L_{\text{lowload}} = 0.8$)

Network:	mesh36	ring36	mesh100	ring100
Average number of runs (total):	180.47	180.57	123.43	114.87
Average number of runs (HL):	98.63	98.77	86.27	80.8
Average number of runs (LL):	81.83	81.8	37.17	34.07

Indeed, the number of reassignments dropped significantly, especially for the 100-node networks. This does not only confirm my assumption from above but also showcases the flexibility provided by the L_{lowload} parameter and the importance of adjusting it appropriately to the given evaluation scenario. Moreover, none of the metrics shown in Figure 9.4 changed for the worse. In contrast, the control ratio of the flexible reassignment even improved slightly, while there was no notable difference for all other plots (which is why I skip these plots here). The former is easy to explain since fewer reassignments result in fewer LCA-to-node assignments added due to instant need but eventually not needed in the long run.

9.3 Observations

In this chapter, I have introduced a new DFG evaluation scenario based on CoMP transmission/reception, after consistently using the same generic DFG types in all previous chapters. Doing this is an important step in this thesis to prove that that my developed solution approaches do not just work well for only one specific DFG scenario. The new scenario differs from the former one in basically all relevant aspects as the CoMP DFGs have significantly higher data rate and processing capacity requirements while requiring a lower maximum round trip latency. In addition, CoMP DFGs feature around twice as many data flows per DFG on average.

Using the CoMP scenario, I have extensively evaluated how FCAPP (in particular FlexCAPF) is influenced by the scenario's different characteristics, with very pleasant results. Overall, FlexCAPF proved to be very stable and continued to provide very good results compared to the reference optimization model and also for larger networks. While the flexible reassignment first revealed a non-desirable effect caused by the new scenario, this could be compensated for by adjusting an already existing parameter steering the reassignment intensity. In total, these results indicate that FlexCAPF is perfectly fit to handle all sorts of scenarios that can be expressed with my DFG concept.

But apart from evaluating the influence of the new DFG scenario, I also took the opportunity to evaluate the influence of my DFG concept as such in Section 9.2.2 by verifying what happens if the crucial requirement to jointly process all data flows of one DFG at one LCA is neglected. The results reveal that this key assumption of my work does indeed have a significant impact and that DFG satisfaction is massively simplified if this requirement is dropped.

10

SDN Testbed-based Evaluation of Flow Processing-aware Control Application Placement

All chapters of this thesis have included extensive evaluation results for the presented approaches. These results were obtained by using simulations, without real traffic and without taking into account possibly required reconfiguration effort in the network to affect the placement decisions of my approaches. Of course, such aspects cannot be neglected in a real deployment. In particular, for routing real traffic exactly as determined by my algorithms, it is necessary to modify the routing entries in the underlying network.

In this chapter, I show as a proof of concept how FlexCAPF can be implemented on top of a testbed with an emulated backhaul network that is based on the Software-Defined Networking (SDN) approach (Section 2.1.3). The underlying testbed has been developed just for this sake and features real traffic, a real SDN controller to modify routing entries in the emulated network and emulation of DFG processing. This work has been conducted together with Tarun Kumar Sarkar over the course of his master thesis [113] under my supervision.

In the remainder of this chapter, I first describe the structure and implementation of the testbed in Section 10.1 and then present and analyze the emulation results obtained from it in Section 10.2. As the technical details of the testbed setup are expansive and well documented in the aforementioned master thesis, I will skip most of these details here and keep the description of the testbed's implementation on a concise, high level.

10.1 Testbed Description

The testbed employed in this chapter is designed as a proof of concept with focus on the DFG concept by featuring real traffic that is routed according to the decisions of FlexCAPF and by emulating of DFG processing at the designated LCAs. However, the testbed is limited in other aspects and does, in

particular, not include actual implementations of RCAs or LCAs. While RCAs are not emulated at all, LCAs are emulated by running a simple application that receives the packets of a data flow of a DFG and sends it back to its source after emulating DFG processing (see Section 10.1.4).

The testbed setup can be divided into the following functional components (also illustrated in Figure 10.1):

- FlexCAPF
- FCAPP SDN controller
- Emulated backhaul network
- Emulation module

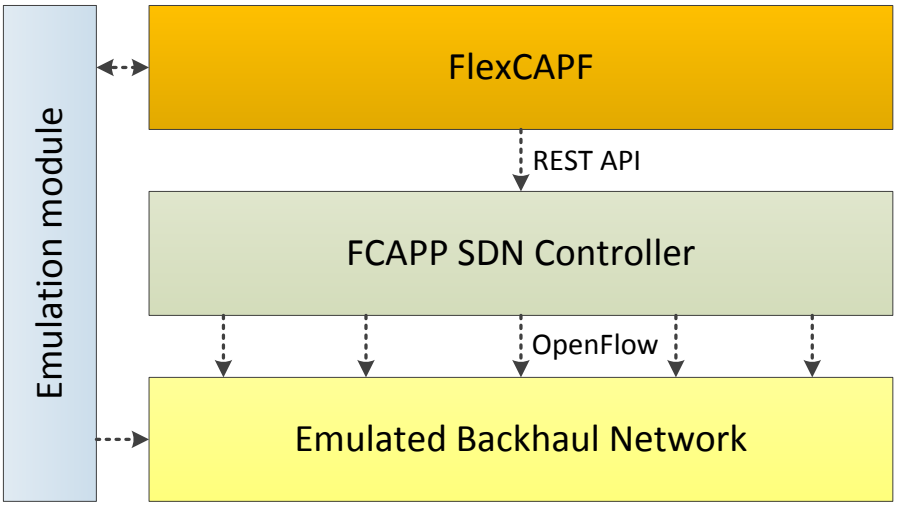


Figure 10.1: FCAPP testbed: functional overview

In the following, I first describe these components in Sections 10.1.2–10.1.4 and then briefly describe the testbed's hardware setup in Section 10.1.5.

10.1.1 FlexCAPF

The FlexCAPF version utilized within the testbed is essentially the FlexCAPF algorithm as described in Section 6.2 extended by additional data structures to store information about the emulated backhaul network (Section 10.1.3) and with additional procedures to communicate with the FCAPP SDN controller and the emulation module (Section 10.1.4).

To communicate with the FCAPP SDN controller, FlexCAPF accesses the controller's Representational State Transfer (REST) API (Section 10.1.2). On the one hand, this happens once at the beginning of an emulation to retrieve the information about the emulated backhaul network. On the other hand, FlexCAPF regularly accesses the controller's REST API during an emulation to affect the necessary changes to the routing entries according to its CA placement decisions.

10.1.2 FCAPP SDN Controller

The FCAPP SDN controller is based on the Ryu SDN controller framework [39], which has been extended by an additional REST API for accepting the requests of FlexCAPF. In accordance with the description of an SDN controller in Section 2.1.3, the FCAPP SDN controller fulfills mainly two purposes within the testbed:

1. providing FlexCAPF with an abstract view of the emulated backhaul network and
2. communicating with the SDN-enabled network devices within the emulated backhaul network and modifying their routing tables upon request by FlexCAPF.

Accordingly, the additional REST API includes functions to (1) send topology information to FlexCAPF and (2) add/remove forwarding entries in the flow table of a switch within the emulated backhaul network via OpenFlow (Section 2.1.3). The topology information required for (1) is retrieved by exploiting Ryu's already built-in topology discovery mechanism that is based on the OpenFlow Discovery Protocol (OFDP).

10.1.3 Emulated Backhaul Network

The backhaul network within the testbed setup is emulated using MaxiNet [114, 115]. MaxiNet is an emulation environment for SDN that allows to emulate larger networks by extending Mininet [116] to span the emulation across multiple physical machines. MaxiNet technically works as a frontend distributing an emulated network over a cluster of Mininet instances running on different physical machines, which are called *workers*.

10.1.4 Emulation Module

At last, the emulation module includes the following processes to start and steer an emulation run:

- Topology initialization,
- DFG request generation,
- DFG processing emulation,
- Traffic generation.

Topology initialization: At the start of an emulation, the emulated backhaul network is generated based on a topology description file of the same type as already used for storing the instances in previous chapters. Within the emulated network, each node is represented by an OpenFlow-enabled switch that has a host attached to it. The switch is needed to route traffic to and through a node, whereas the attached host is required to run user processes, e.g. for generating traffic with the corresponding node as source or for representing

DFG processing (as described further below). While links between switches are generated based on the parameters from the topology description file, the links between a switch and its host are set up without any constraints. It should be noted that it might not be necessary to emulate all network nodes as such a pair in practice, since a node where no DFG ever originates from does not need a host and leaf nodes where no traffic is routed through do not need a switch. But since it is not always possible to predict if this would be the case over the course of an emulation run, all nodes are homogeneously emulated like that nonetheless.

As last step, the topology initialization launches a socat process on every potential host. Socat [117] is a command line-based multipurpose relay utility that, among others, includes the functionality for bidirectional data transfer between two devices. During an emulation run, this process creates a child process for every data flow of any satisfied DFG that duplicates every received packet and sends it back to its source. This corresponds to the DFG round trip assumptions made in the previous chapters.

DFG request generation: During an emulation, DFG requests are generated using a Poisson process just as for the dynamic network simulations (e.g. in Section 6.3.1). These requests are directly sent to FlexCAPF and will trigger an execution of the algorithm to satisfy these DFGs. It is important to note that the DFG requests are only virtual objects including all relevant DFG parameters at this stage; the real traffic corresponding to their parameters is generated at a later stage.

DFG processing emulation: In theory, three steps are necessary for each individual packet of each data flow of each DFG to emulate DFG processing in the testbed: (1) receiving the packet, (2) executing some processing logic on the packet that corresponds to the processing requirements of its associated DFG and (3) sending the packet back to its source. As described earlier, steps (1) and (3) are already handled by socat. But for (2), implementing real data processing is not a feasible option for this testbed due to hardware constraints. Instead, DFG data processing is emulated by delaying each packet at an LCA according to the delay that real processing would take before it is sent back to its source. This is done by installing queues and filters at an LCA using the Linux Traffic Control functionality [118] with NetEm [119, 120].

The delay for each DFG can directly be retrieved from FlexCAPF based on the processing capacity assigned for a DFG. However, adopting these non-uniform delays without modification practically results in a separate queue and filter for each DFG. In an early development stage of the testbed, this approach caused severe performance issues due to high memory consumption with more and more generated DFGs over the course of emulation runs. Therefore, every delay is now rounded up to the nearest tenth of a millisecond and I will denote the resulting value as *delay bin* in the following. As a consequence, only one queue and one filter is needed at each LCA for each delay bin, which handle all DFGs with the corresponding delay bin. In total, this alternative

approach trades a tiny bit of emulation accuracy for a significant performance improvement.

During an emulation run, missing queues and filters that are not already in place at an LCA are always set up once FlexCAPF has finished its execution and accessed the REST API of the FCAPP SDN controller to change the corresponding routing entries (Section 10.1.1).

Traffic generation: Ensuing the execution of FlexCAPF, the modification of the flow entries in the emulated network and the setup of all missing queues and filters, the real data traffic is finally generated in the network. This is done using Iperf [121], which is a tool normally used for measuring the throughput of the network and can create Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) data streams. In particular, its UDP mode allows to generate data streams with a controlled data rate and for a specified duration, which is used in the testbed to generate each data flow of each DFG with the right data rate and the right duration. Accordingly, for each data flow of each DFG one Iperf process is started on the host in the emulated network that belongs to the source node of the data flow.

However, Iperf does not take any parameter to include user-specified information in packets. For this reason, the Iperf version used in the testbed has been customized to include a DFG's delay bin into each of its packets. This information is then retrieved by the earlier mentioned filters to put a packet into the correct queue, i.e. to apply the correct delay before sending it back to its source.

Last but not least, it is also necessary to account for the case that FlexCAPF reassigns a DFG from one LCA to another one. In such a case, the corresponding Iperf processes are killed and new Iperf processes are started as described above, but only for the remaining duration of the corresponding DFG.

10.1.5 Hardware Setup

The hardware used for the testbed consists of four physical machines (PC1–PC4), all with Intel® Core™2 Duo E8400 CPUs running at 3.00 GHz and 8 Gb RAM, which are interconnected by a 1 Gbit/s Ethernet switch. One of the machines (PC1) is used to run the MaxiNet frontend, FlexCAPF, the FCAPP SDN controller and all procedures from the emulation module. The other three machines (PC2–PC4) are used as MaxiNet workers and form the MaxiNet cluster that hosts the emulated backhaul network. For illustration, the testbed's hardware setup is depicted in Figure 10.2.

10.2 Evaluation

In this section, I present selected evaluation results obtained from conducting emulation experiments on the testbed. Additional extensive evaluation results,

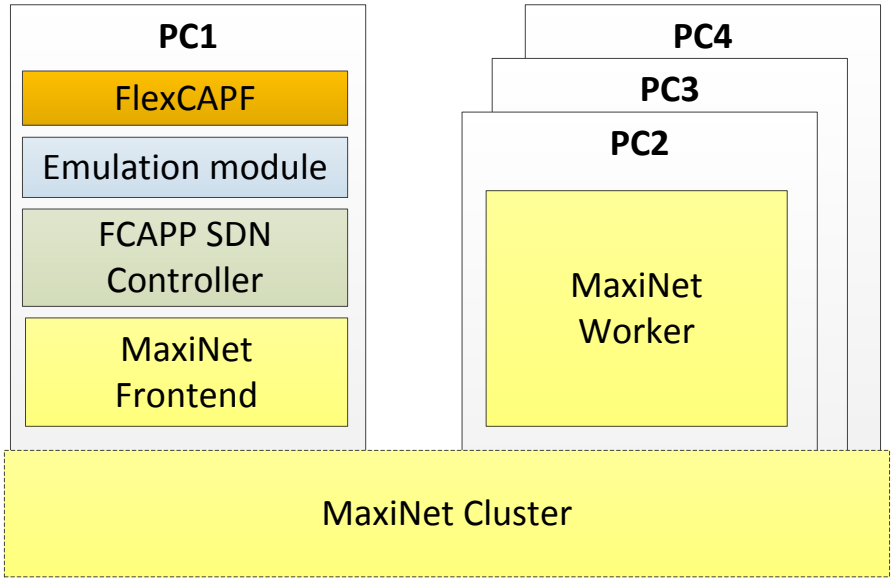


Figure 10.2: FCAPP testbed: hardware setup

including proof of concept testing results for the correctness of emulated DFG processing, traffic generation and traffic routing can be found in [113].

I first elaborate on the evaluation scenario in Section 10.2.1 and then present evaluation results in Section 10.2.2.

10.2.1 Evaluation Scenario

As already stated in Section 10.1.4, incoming DFG requests are generated using a non-stationary Poisson process as initially introduced for the dynamic network simulations in Section 6.3.1. Due to the hardware limitations of the testbed, the intensity of the Poisson process has been scaled (analogous to Section 9.2.3) by a factor of 0.2 for the generic scenario and by a factor of 0.02 for the CoMP scenario. Further, the employed load curve has been scaled from 24 hours to one hour as already done for the evaluation of DistCAPA in Section 7.3.2.

Each emulation experiment is started at $t = -1800.0$ before starting monitoring for 2 hours of *system time* at $t = 0$. The synchronization with the *emulation time* t generated by the Poisson process is achieved by forcing the DFG request generation script running the Poisson process to sleep after every run of FlexCAPF if and only if the emulation time is ahead of the system time that has passed since $t = 0$ was reached.

For the emulation experiments presented here, the emulated backhaul network has been initialized based on the topology description file of the 36-node network with mesh topology that was already used in Section 6.3. But to compensate for the reduced number of generated DFGs, the backhaul link

capacities and the available processing capacity per potential host have both been scaled by a factor of 0.2, giving a link capacity of $b_{\text{cap}}(v, w) = 0.5$ Gbit/s for all links $(v, w) \in E$ and a processing capacity of $p_{\text{node}} = 40$ GFLOPS for all potential hosts $c \in C$. At last, the L_{lowload} parameter steering the intensity of the low-load handling of FlexCAPF has been set to 0.9 for the generic scenario and to 0.8 for the CoMP scenario according to the assessments in previous chapters.

10.2.2 Evaluation Results

Based on the evaluation scenario described in the previous section, I have conducted two emulation experiments, one using the generic DFG scenario and one using the CoMP DFG scenario. As for previous evaluations, I have extracted the data from FlexCAPF after each CA reassignment, i.e. each time the set of used CAs was modified during the course of the emulation.

First, Figures 10.3 and 10.4 show the number of LCAs used over time for both experiments. For illustration, the plots also contain the loadlevel curve used for the non-stationary Poisson process, which I have scaled by the maximum number of LCAs observed each to fit the given value range.

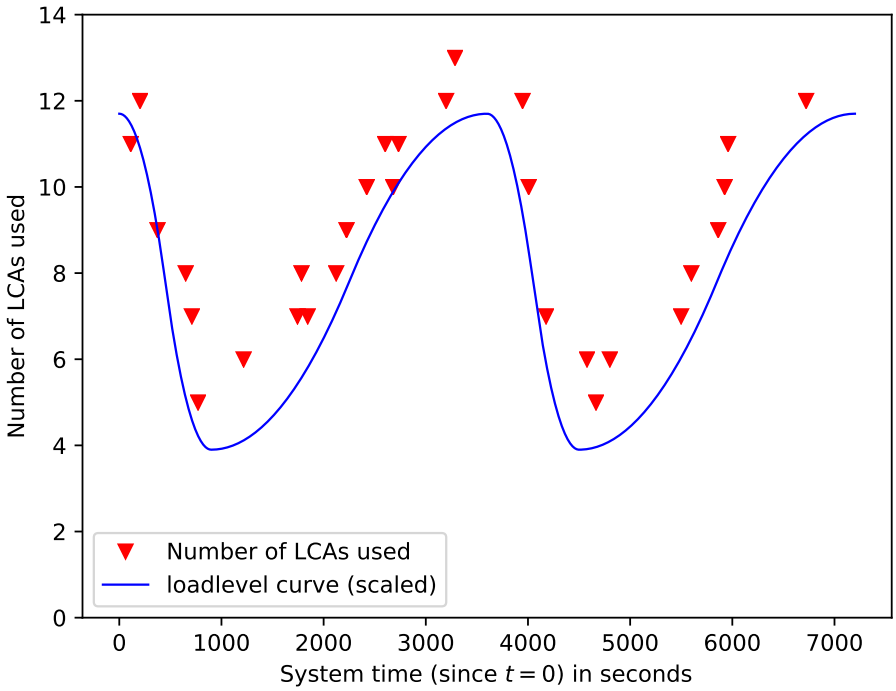


Figure 10.3: Emulation results: LCAs used (generic scenario)

In total, the results perfectly fit those already seen for the corresponding simulations before. Both plots show that the number of LCAs is flexibly adapted according to the current network load even with using real traffic and working in real time. In accordance with the results from Section 9.2.3,

it can also be seen that the CoMP scenario causes more CA reassignments compared to the generic scenario. Additionally, the CoMP results also show more fluctuation, which can be explained by the relatively low number of DFGs in the network compared to the generic scenario, so that individual new or expiring DFGs have a larger impact.

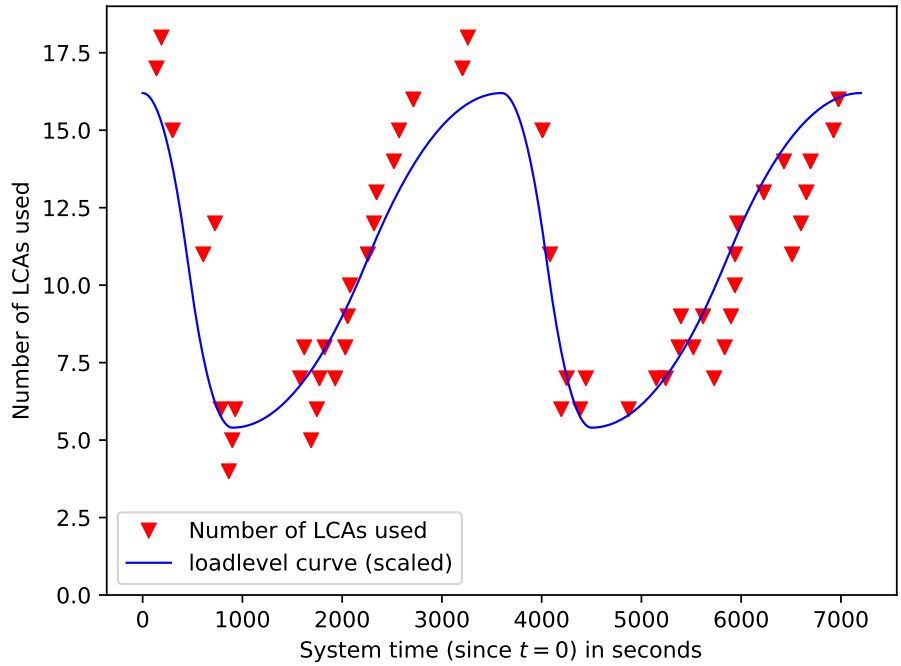


Figure 10.4: Emulation results: LCAs used (CoMP scenario)

Next, Figures 10.5 and 10.6 illustrate results of a runtime analysis performed for both experiments. In these plots, the runtime of FlexCAPF is treated separately from the time needed for reconfiguring the emulated network accordingly.

The first eye-catching aspect to note is that the network reconfiguration time is significantly higher than the algorithm runtime. Still, it generally remains below one second, which can be considered as a reasonable time range. In particular, it has to be kept in mind that the entire network reconfiguration in the testbed is done by a single FCAPP SDN controller running one one physical machine. In a real-world deployment with actual CA realization, this reconfiguration work would be handled by multiple LCAs and one or multiple RCAs in parallel, each running on their individual physical machines.

Another notable observation to be made is that the network reconfiguration time also seems to scale with the current network load. While the correlation is not as obvious as for the number of LCAs, it is still clearly visible. The explanation for this is rather simple, since more DFGs in the network naturally provide more possibilities for necessary reconfiguration work. But in addition, it should not be neglected that this might also be influenced by the fact that a higher load in the emulated network consumes more testbed hardware

resources, so that there are less available resources left for executing the needed reconfiguration work.

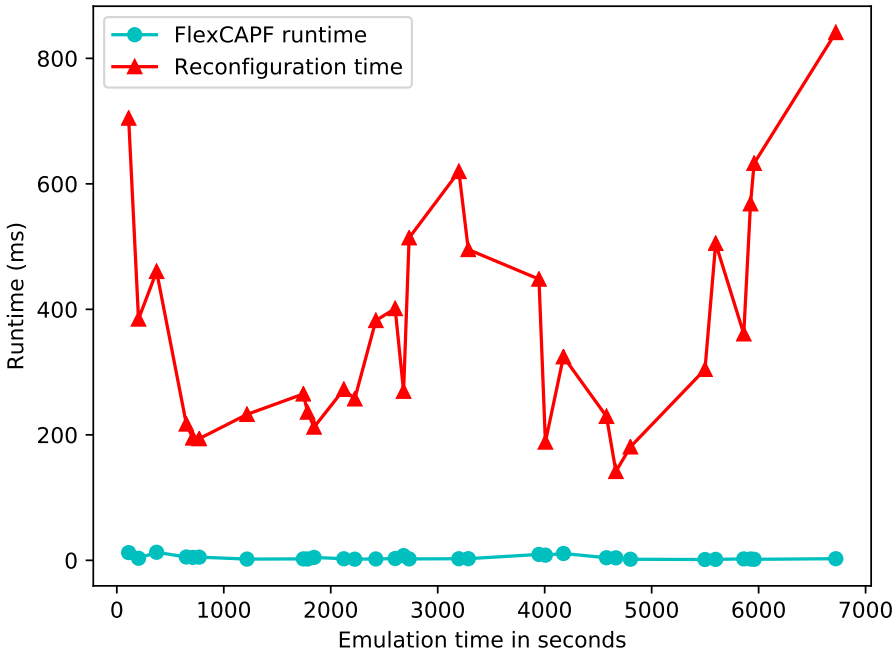


Figure 10.5: Emulation results: runtime analysis (generic scenario)

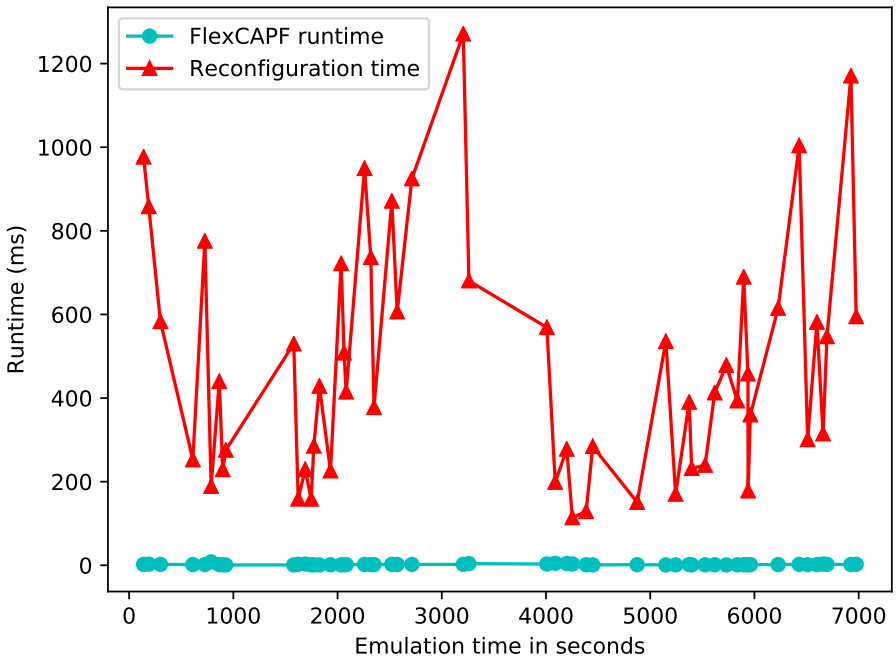


Figure 10.6: Emulation results: runtime analysis (CoMP scenario)

10.3 Observations

In this chapter, I have shown how FlexCAPF can be implemented on top an SDN-based emulated backhaul network. The underlying testbed features real traffic, which is routed in compliance with the placement decisions of FlexCAPF. This is realized via a real SDN controller that modifies the routing entries accordingly during network operation. Further, the testbed emulates DFG processing by delaying each packet according to the processing delay that would be caused by real data processing.

All of this serves as a proof of concept and shows that FlexCAPF, and more generally FCAPP, can be realized in such a real-world deployment and at the same time flexibly adapt CA placement to the current network load as previously seen from simulation results.

11

Conclusion and Future Research Directions

In Section 1.2, I introduced my key research question: Is it feasible to

(1) efficiently decide CA placement considering all latency, data rate and processing capacity constraints and
(2) do this within the order of seconds to milliseconds to flexibly adapt placement decisions during network operation in reaction to traffic load changes to maintain near-optimal network performance.

In this final chapter, I first summarize my work in Section 11.1 and conclude it based on the question above. Afterwards, I outline future research directions in Section 11.2.

11.1 Summary and Conclusion

In this thesis, I have investigated the problem of placing Control Applications (CAs) within the backhaul network of a mobile access network, which I introduced as *Flow processing-aware Control Application Placement Problem (FCAPP)*. To express the requirements of coordination mechanisms that should be executed on CAs, I have also introduced the concept of Data Flow Groups (DFGs) that are considered and whose resource demands are attempted to be satisfied by FCAPP.

I have described several variations of FCAPP and developed solution approaches for all of them. In Chapter 3, I have provided a first formalization of FCAPP based on equal-share processing scheduling. I presented an optimization model as reference solution and a multi-layer greedy heuristic (GreedyFCAPA) to solve FCAPP fast and efficiently. Even though equal-share scheduling later turned out to be replaced my a more efficient scheduling approach, this first variation of FCAPP provided valuable insights into the nature of FCAPP and allowed to prove that FCAPP is NP-hard. I then attempted to improve the solution quality compared to GreedyFCAPA by assessing Genetic Algorithms (GAs) in Chapter 4, which succeeded, but at the cost of a

significant increase in execution time. I then looked for other improvement opportunities and investigated FCAPP with proportional-share scheduling as a more elaborate processing scheduling approach in Chapter 5. This turned out to be a very good choice since the corresponding optimization model resulted in vastly improved solution quality compared to equal-share scheduling, while the modified version of GreedyFCAPA not only turned out to run faster with proportional-share scheduling but also provided near-optimal results compared to the optimization model.

At this point, I concluded that GreedyFCAPA with proportional-share scheduling is able to efficiently decide CA placement considering all latency, data rate and processing capacity constraints, so I focused on flexible reassignment of CAs in reaction to changing network load over time. To do this, I built up on GreedyFCAPA to create a flexible placement framework (FlexCAPF) that is able to place and flexibly reassign CAs during network operation. Evaluation by means of network simulation showed that FlexCAPF efficiently reassigns CAs in reaction to changing network load and minimizes needed reconfiguration work by taking into account the previous placement. FlexCAPF also ran significantly faster (within the order of milliseconds) with only a small decrease in solution quality compared to a new CA placement from scratch (which completely ignores the previous placement). As a result, FlexCAPF represents a solution for FCAPP that allows to answer both parts of my key research question in the affirmative.

In addition, there were several other aspects of FCAPP that I decided to tackle and investigate. In Chapter 7, I dropped the assumption that an FCAPP solution approach can be executed logically centralized and presented a distributed algorithm for FCAPP (DistCAPA). Evaluation results showed that DistCAPA places and flexibly reassigns CAs during network operation just like FlexCAPF and with only a marginal decrease in solution quality. Then, I presented an extended variation of FCAPP that appropriately takes the backbone network into account in Chapter 8. This extension enables FlexCAPF to deal with the possibility that DFGs require a backbone connection and gave a lot of additional, interesting insight. The latter led me to exchange the generic DFG evaluation scenario used so far for a very different one, based on CoMP transmission and reception. In Chapter 9, I presented this DFG scenario and studied its effect on the results of FlexCAPF. The results proved that FlexCAPF had no problems to cope with the new scenario by showing a very stable behavior. Finally, I provided a proof of concept for a real-world implementation of FlexCAPF in Chapter 10 by showing how FlexCAPF can be implemented on top of an SDN-based emulated backhaul network. The results of emulation experiments revealed that FlexCAPF is still able to flexibly adapt CA placements according to the current network load even with using real traffic and working in real time. Further, the combined algorithm runtime and network reconfiguration time still generally lied within an acceptable range of below one second.

In total, my work represents a unique contribution for understanding and solving the problem of placing CAs in the backhaul network of mobile access networks. Due to the DFG concept, it is possible to assess various types of coordination mechanisms with my solution approaches. In particular, I have developed two very powerful solution approaches for placing and flexibly reassigning CAs, FlexCAPF and DistCAPA, where FlexCAPF should always be preferred if a logically centralized execution is possible. But, moreover, the application field of my approaches is not limited to live network operation as it is also possible to use them to characterize the benefits of potential network extensions via simulations before making a final deployment decision. Overall, my work constitutes a valuable assistance for efficiently realizing coordination mechanisms via virtualized control applications in future mobile access networks – thus increasing their resource efficiency and reducing their operational costs.

11.2 Future Research Directions

For future research, I propose the following directions for further investigation:

Refinement of modeling assumptions: Over the course of my work, I made several modeling assumptions to limit the scope of my work. I have later dropped some of these assumptions, e.g. logically centralized execution or a simplified backbone connection, to investigate in the corresponding directions. But other assumptions that might be worth to investigate were kept throughout my work. An example for this is assuming link delay to be independent of the network load and ignoring possible queuing delays. Extending FCAPP by a more realistic queuing model might bring additional interesting insight and improve the model's accuracy.

Investigate other application scenarios: For the major part of my work, I consider a generic DFG evaluation scenario. Even though I later introduced the CoMP evaluation scenario and studied FCAPP for this very different scenario, other possible application scenarios with again very different attributes might give new insight for FCAPP and motivate additional improvements for my solution approaches.

Assess other algorithmic concepts for FCAPP: In Chapter 4, I assessed GAs for FCAPP, since GAs are known to often provide good heuristic results for difficult optimization problems. But the same reasoning also applies to other widely employed concepts, such as simulated annealing [122] or particle swarm optimization [123]. Even though the presented GAs gave satisfactory results for FCAPP, it would of course be possible that other concepts yield better results with less execution time.

Hybrid FCAPP solution scheme: The aforementioned algorithmic concepts all share the characteristic of storing intermediate solutions while trying to improve solution quality until a termination criterion is reached. This could be exploited to combine such a solution approach with FlexCAPF into a hybrid solution scheme that is able to provide fast solutions if needed but continuously optimizes the solution in the background. For example, it would be possible to define a certain trade-off between operational benefit and reconfiguration effort and to affect a reassignment if such a solution is found – independent of immediate load changes.

Leverage prediction techniques for FCAPP: All of the presented approaches for CA reassignment work in reaction to changing network load. However, one very promising research direction would be to attempt anticipatory reassignment according to predicted load changes obtained from integrated prediction techniques [124]. A detailed analysis of the trade-off between additional resource consumption against reduced reconfiguration time would certainly bring interesting results.

FCAPP prototype enhancements: The testbed presented in Chapter 10 represents a first proof of concept for realizing FCAPP in a real-world deployment. But as stated in the same chapter, the scope of the existing testbed is limited and does not include actual implementations of CAs or real DFG processing. Extending the testbed into a prototype with actual CA implementations and real applications for DFG processing would require additional hardware resources and a lot of additional effort but would definitely constitute a valuable contribution for the investigation of FCAPP.

Integrate FCAPP into an existing NFV platform: Another fundamental milestone for realizing FCAPP in a real-world deployment is the integration of an FCAPP solution approach into an existing NFV platform. A very promising orchestration platform for attacking this task is given by the SONATA service platform [125]. This platform is capable of managing and orchestrating VNFs (e.g. CAs) and also allows custom placement algorithms to be integrated as so-called *function- or service-specific managers*, which only requires minor extensions of an existing placement algorithm (e.g. FlexCAPF) to enable communication with the used messaging system. Overall, such an integration would already correspond to a real-world deployment of FCAPP.

Bibliography

[1] Cisco. Cisco visual networking index: Global mobile data traffic forecast update, 2013–2018 white paper. *Cisco Public Information*, 2014.

[2] Cisco. Cisco visual networking index: Global mobile data traffic forecast update, 2014–2019 white paper. *Cisco Public Information*, 2015.

[3] Cisco. Cisco visual networking index: Global mobile data traffic forecast update, 2016–2021 white paper. *Cisco Public Information*, 2017.

[4] Prakash Bhat, Satoshi Nagata, Luis Campoy, Ignacio Berberana, Thomas Derham, Guangyi Liu, Xiaodong Shen, Pingping Zong, and Jin Yang. LTE-Advanced: An operator perspective. *IEEE Communications Magazine*, pages 104–114, 2012.

[5] Arash Asadi, Vincenzo Sciancalepore, and Vincenzo Mancuso. On the efficient utilization of radio resources in extremely dense wireless networks. *IEEE Communications Magazine*, 53(1):126–132, 2015.

[6] Huaning Niu, Clara Li, Apostolos Papathanassiou, and Geng Wu. RAN architecture options and performance for 5G network evolution. In *Wireless Communications and Networking Conference Workshops (WCNCW)*. IEEE, 2014.

[7] Alex Galis, Stuart Clayman, Lefteris Mamatas, Javier Rubio Loyola, Antonio Manzalini, Slawomir Kuklinski, Joan Serrat, and Theodore Zahariadis. Softwarization of future networks and services-programmable enabled networks as next generation software defined networks. In *Proceedings of IEEE Software Defined Networks for Future Networks and Services (SDN4FNS)*, pages 1–7. IEEE, 2013.

[8] Li Erran Li, Zhuoqing Morley Mao, and Jennifer Rexford. Toward software-defined cellular networks. In *European Workshop on Software Defined Networking (EWSDN)*, pages 7–12. IEEE, 2012.

[9] Aditya Gudipati, Daniel Perry, Li Erran Li, and Sachin Katti. SoftRAN: Software Defined Radio Access Network. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 25–30. ACM, 2013.

[10] Hassan Ali-Ahmad, Claudio Cicconetti, Antonio de la Oliva, Martin Dräxler, Rohit Gupta, Vincenzo Mancuso, Laurent Roullet, and Vincenzo Sciancalepore. CROWD: An SDN Approach for DenseNets. In *Proceedings of the 2nd European Workshop on Software Defined Networks (EWSDN)*, 2013.

[11] Hassan Ali-Ahmad, Claudio Cicconetti, Antonio de la Oliva, Vincenzo Mancuso, Malla Reddy Sama, Pierrick Seite, and Sivasothy Shanmugalingam. An SDN-based Network Architecture for Extremely Dense Wireless Networks. In *Proceedings of IEEE Software Defined Networks for Future Networks and Services (IEEE SDN4FNS)*, 2013.

[12] S. Auroux and H. Karl. Flow processing-aware Controller Placement in Wireless DenseNets. In *Proceedings of the 25th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. IEEE, 2014.

[13] S. Auroux and H. Karl. Efficient Flow Processing-aware Controller Placement in Future Wireless Networks. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2015.

[14] S. Auroux, M. Dräxler, A. Morelli, and V. Mancuso. Dynamic Network Reconfiguration in Wireless DenseNets with the CROWD SDN Architecture. In *European Conference on Networks and Communications (EuCNC)*, 2015.

[15] S. Auroux and H. Karl. Flexible reassignment of flow processing-aware controllers in future wireless networks. In *Proceedings of the 26th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. IEEE, 2015.

[16] S. Auroux, D. Parruca, and H. Karl. Joint real-time scheduling and interference coordination for wireless factory automation. In *Proceedings of the 27th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC)*. IEEE, 2016.

[17] S. Auroux, S. Scholz, and H. Karl. Assessing Genetic Algorithms for Placing Flow Processing-aware Control Applications. In *Proceedings of European Wireless (EW)*, 2017.

[18] I. Aktas, J. Ansari, S. Auroux, D. Parruca, M. Guirao, and B. Holfeld. A Coordination Architecture for Wireless Industrial Automation. In *Proceedings of European Wireless (EW)*, 2017.

[19] H. Afifi, S. Auroux, and H. Karl. Network Function Virtualization for Wireless Acoustic Sensor Networks: An interference-aware placement and routing approach. In *Submitted to IEEE International Conference on Computer Communications (INFOCOM)*, 2018.

[20] H. Afifi, S. Auroux, and H. Karl. MARVELO: Wireless Virtual Network Embedding for Overlay Graphs with Loops. In *Submitted to IEEE Wireless Communications and Networking Conference (WCNC)*, 2018.

[21] S. Auroux and H. Karl. Distributed Placement of Virtualized Control Applications in Mobile Backhaul Networks. In *Submitted to IEEE Wireless Communications and Networking Conference (WCNC)*, 2018.

[22] Erik Dahlman, Stefan Parkvall, and Johan Skold. *4G: LTE/LTE-Advanced for Mobile Broadband*. Academic press, 2013.

[23] Van-Giang Nguyen, Truong-Xuan Do, and YoungHan Kim. SDN and Virtualization-Based LTE Mobile Network Architectures: A Comprehensive Survey. *Wireless Personal Communications*, 86(3):1401–1438, 2016.

[24] Orawan Tipmongkolsilp, Said Zaghloul, and Admela Jukan. The evolution of cellular backhaul technologies: Current issues and future trends. *IEEE Communications Surveys & Tutorials*, 13(1):97–113, 2011.

[25] Wei-Tao Shaw, Shing-Wa Wong, Ning Cheng, Koussalya Balasubramanian, Xiaoqing Zhu, Martin Maier, and Leonid G Kazovsky. Hybrid architecture and integrated routing in a scalable optical–wireless access network. *Journal of Lightwave Technology*, 25(11):3443–3451, 2007.

[26] GreenTouch. GreenTouch Green Meter Research Study: Reducing the Net Energy Consumption in Communications Networks by up to 90% by 2020. Technical report, GreenTouch, 2013.

[27] Andreas F Molisch. *Wireless communications*, volume 34. John Wiley & Sons, 2012.

[28] Michael Kende. The digital handshake: Connecting internet backbones. *CommLaw Conspectus: Journal of Communications Law and Technology Policy (1993-2015)*, 11(1):45–70, 2003.

[29] Abdelbaset S Hamza, Shady S Khalifa, Haitham S Hamza, and Khaled Elsayed. A survey on inter-cell interference coordination techniques in ofdma-based cellular networks. *IEEE Communications Surveys & Tutorials*, 15(4):1642–1670, 2013.

[30] Lindbom Lars, Love Robert, Krishnamurthy Sandeep, Yao Chunhai, Miki Nobuhiko, and Chanddrasekhar Vikram. Enhanced Inter-Cell Interference Coordination For Heterogeneous Networks in LTE-Advanced: A Survey. *Texas: Cornell University Library (http://arxiv.org/abs/1112.1344)*, 2011.

[31] Shipon Ali. On the Evolution of Coordinated Multi-Point (CoMP) Transmission in LTE-Advanced. *International Journal of Future Generation Communication and Networking*, 7(4):91–102, 2014.

[32] Qian Zhang, Chenyang Yang, and Andreas F Molisch. Cooperative downlink transmission mode selection under limited-capacity backhaul. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC)*, pages 1082–1087, 2012.

[33] Daewon Lee, Hanbyul Seo, Bruno Clerckx, Eric Hardouin, David Mazzarese, Satoshi Nagata, and Krishna Sayana. Coordinated multipoint transmission and reception in LTE-advanced: deployment scenarios and operational challenges. *IEEE Communications Magazine*, 50(2):148–155, 2012.

[34] Third Generation Partnership Project (3GPP). 3GPP TR 36.819 V11.1.0: Coordinated multi-point operation for LTE physical layer aspects (Release 11), 2011.

[35] Huawei Technologies. Huawei Response to ACMA's paper: Towards 2020 – Future spectrum requirements for mobile broadband, 2011.

[36] Open Networking Foundation (ONF). SDN Architecture 1.0 Overview. `https://www.opennetworking.org/`, 2013.

[37] Open Networking Foundation (ONF). Software-Defined Networking: The New Norm for Networks. *ONF White Paper*, 2:2–6, 2012.

[38] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[39] Ryu SDN Framework. `http://osrg.github.io/ryu/`.

[40] 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. *IEEE Communications Surveys & Tutorials*, 18(1):236–262, 2016.

[41] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[42] Margaret Chiosi, Don Clarke, Peter Willis, Andy Reid, James Feger, Michael Bugenhagen, Waqar Khan, Michael Fargano, Chunfeng Cui, Hui Deng, et al. Network functions virtualisation: An introduction, benefits, enablers, challenges and call for action. In *SDN and OpenFlow World Congress*, pages 22–24, 2012.

[43] European Telecommunications Standards Institute (ETSI). GS NFV 002-V1.1.1 - Network Functions Virtualization (NFV): Architechtural Framework. `http://www.etsi.org/`, 2013.

[44] Margaret Chiosi et al. Network Functions Virtualisation (NFV) - Update White Paper. In *SDN and OpenFlow World Congress*, 2013.

[45] Agoston E. Eiben and James E. Smith. *Introduction to evolutionary computing*. Springer, 2003.

[46] Pedro Larrañaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[47] Mantas Paulinas and Andrius Ušinskas. A survey of genetic algorithms applications for image enhancement and segmentation. *Information Technology and control*, 36(3), 2015.

[48] Theodore W. Manikas, Kaveh Ashenayi, and Roger L. Wainwright. Genetic algorithms for autonomous robot navigation. *Instrumentation & Measurement Magazine, IEEE*, 10(6):26–31, 2007.

[49] John H. Holland. Genetic algorithms and the optimal allocation of trials. *SIAM Journal on Computing*, 2(2):88–105, 1973.

[50] D. B. Hibbert. Hybrid genetic algorithms. *Data Handling in Science and Technology*, 23:55–68, 2003.

[51] Fatos Xhafa, Christian Sánchez, and Leonard Barolli. Genetic algorithms for efficient placement of router nodes in wireless mesh networks. In *Proceedings of 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 465–472. IEEE, 2010.

[52] Arpit Tripathi, Pulkit Gupta, Aditya Trivedi, and Rahul Kala. Wireless sensor node placement using hybrid genetic programming and genetic algorithms. *International Journal of Intelligent Information Technologies (IJIIT)*, 7(2):63–83, 2011.

[53] Andreas Fischer, Juan Felipe Botero, Michael Till Beck, Hermann De Meer, and Xavier Hesselbach. Virtual network embedding: A survey. *IEEE Communications Surveys & Tutorials*, 15(4):1888–1906, 2013.

[54] Xiuming Mi, Xiaolin Chang, Jiqiang Liu, Longmei Sun, and Bin Xing. Embedding virtual infrastructure based on genetic algorithm. In *13th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 239–244. IEEE, 2012.

[55] Nancy A Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[56] Gerard Tel. *Introduction to distributed algorithms*. Cambridge university press, 2000.

[57] Gerard J Foschini and Zoran Miljanic. A simple distributed autonomous power control algorithm and its convergence. *IEEE transactions on vehicular Technology*, 42(4):641–646, 1993.

[58] Martin Kubisch, Holger Karl, Adam Wolisz, Lizhi Charlie Zhong, and Jan Rabaey. Distributed algorithms for transmission power control in wireless sensor networks. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC)*, volume 1, pages 558–563. IEEE, 2003.

[59] Martine Labbé. Facility location: models, methods and applications. In *Operations Research and Decision Aid Methodologies in Traffic and Transportation Management*, pages 264–285. Springer, 1998.

[60] Thomas Moscibroda and Rogert Wattenhofer. Facility location: distributed approximation. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 108–117. ACM, 2005.

[61] Nikolaos Laoutaris, Georgios Smaragdakis, Konstantinos Oikonomou, Ioannis Stavrakakis, and Azer Bestavros. Distributed placement of service facilities in large-scale networks. In *International Conference on Computer Communications (INFOCOM)*, pages 2144–2152. IEEE, 2007.

[62] Matthias Keller, Stefan Pawlik, Peter Pietrzk, and Holger Karl. A local heuristic for latency-optimized distributed cloud deployment. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC)*, pages 429–434. IEEE, 2013.

[63] Juliver Gil Herrera and Juan Felipe Botero. Resource allocation in NFV: A comprehensive survey. *IEEE Transactions on Network and Service Management*, 13(3):518–532, 2016.

[64] Hendrik Moens and Filip De Turck. VNF-P: A model for efficient placement of virtualized network functions. In *10th International Conference on Network and Service Management (CNSM)*, pages 418–423. IEEE, 2014.

[65] Sevil Mehraghdam, Matthias Keller, and Holger Karl. Specifying and placing chains of virtual network functions. In *3rd International Conference on Cloud Networking (CloudNet)*. IEEE, 2014.

[66] Marco Savi, Massimo Tornatore, and Giacomo Verticale. Impact of processing costs on service chain placement in network functions virtualization. In *IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 191–197. IEEE, 2015.

[67] Long Qu, Chadi Assi, and Khaled Shaban. Delay-aware scheduling and resource optimization with network function virtualization. *IEEE Transactions on Communications*, 64(9):3746–3758, 2016.

[68] Ming Xia, Meral Shirazipour, Ying Zhang, Howard Green, and Attila Takacs. Network function placement for NFV chaining in packet/optical datacenters. *Journal of Lightwave Technology*, 33(8):1565–1570, 2015.

[69] Milad Ghaznavi, Aimal Khan, Nashid Shahriar, Khalid Alsubhi, Reaz Ahmed, and Raouf Boutaba. Elastic virtual network function placement. In *IEEE 4th International Conference on Cloud Networking (CloudNet)*, pages 255–260. IEEE, 2015.

[70] Sevil Dräxler, Holger Karl, and Zoltán Ádám Mann. Joint Optimization of Scaling and Placement of Virtual Network Services. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 365–370. IEEE, 2017.

[71] Zoltán Ádám Mann. Allocation of Virtual Machines in Cloud Data Centers—A Survey of Problem Models and Optimization Algorithms. *ACM Computing Surveys (CSUR)*, 48(1):11, 2015.

[72] Xiaoqiao Meng, Vasileios Pappas, and Li Zhang. Improving the scalability of data center networks with traffic-aware virtual machine placement.

In *International Conference on Computer Communications (INFOCOM)*, pages 1–9. IEEE, 2010.

[73] Aaron Gember, Anand Krishnamurthy, Saul St John, Robert Grandl, Xiaoyang Gao, Ashok Anand, Theophilus Benson, Aditya Akella, and Vyas Sekar. Stratos: A network-aware orchestration layer for middleboxes in the cloud. Technical report, Technical Report, 2013.

[74] Mansoor Alicherry and TV Lakshman. Network aware resource allocation in distributed clouds. In *International Conference on Computer Communications (INFOCOM)*, pages 963–971. IEEE, 2012.

[75] Matthias Keller and Holger Karl. Response time-optimized distributed cloud resource allocation. In *Proceedings of the ACM SIGCOMM workshop on Distributed cloud computing*, pages 47–52. ACM, 2014.

[76] Mina Sedaghat, Francisco Hernandez-Rodriguez, and Erik Elmroth. A virtual machine re-packing approach to the horizontal vs. vertical elasticity trade-off for cloud autoscaling. In *Proceedings of the ACM Cloud and Autonomic Computing Conference*, page 6. ACM, 2013.

[77] Zhen Xiao, Weijia Song, and Qi Chen. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE transactions on parallel and distributed systems*, 24(6):1107–1117, 2013.

[78] Brandon Heller, Rob Sherwood, and Nick McKeown. The controller placement problem. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 7–12. ACM, 2012.

[79] Minzhe Guo and Pallab Bhattacharya. Controller placement for improving resilience of software-defined networks. In *4th IEEE Int. Conference on Networking and Distributed Computing (ICNDC)*, pages 23–27, 2013.

[80] Yannan Hu, Wendong Wang, Xiangyang Gong, Xirong Que, and Shiduan Cheng. On reliability-optimized controller placement for software-defined networks. *Communications, China*, 11(2):38–54, 2014.

[81] Md. Faizul Bari, Arup Raton Roy, Shihabur Rahman Chowdhury, Qi Zhang, Mohamed Faten Zhani, Reaz Ahmed, and Raouf Boutaba. Dynamic Controller Provisioning in Software Defined Networks. In *9th IEEE/ACM/IFIP International Conference on Network and Service Management (CNSM)*, 2013.

[82] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella. Towards an elastic distributed SDN controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 7–12. ACM, 2013.

[83] Valliappan Annamalai, Sandeep KS Gupta, and Loren Schwiebert. On tree-based convergecasting in wireless sensor networks. In *Proceedings of IEEE Wireless Communications and Networking Conference (WCNC)*, volume 3, pages 1942–1947. IEEE, 2003.

[84] Sarma Upadhyayula, Valliappan Annamalai, and Sandeep KS Gupta. A low-latency and energy-efficient algorithm for convergecast in wireless sensor networks. In *Global Telecommunications Conference (GLOBECOM)*, volume 6, pages 3525–3530. IEEE, 2003.

[85] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 19–24. ACM, 2012.

[86] Mehrdad Moradi, Wenfei Wu, Li Erran Li, and Zhuoqing Morley Mao. SoftMoW: Recursive and Reconfigurable Cellular WAN Architecture. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 377–390. ACM, 2014.

[87] Thorsten Biermann. *Dealing with backhaul network limitations in coordinated multi-point deployments*. PhD thesis, Universität Paderborn, 2012.

[88] Samuel Burer and Anureet Saxena. The MILP road to MIQCP. *Mixed Integer Nonlinear Programming*, pages 373–405, 2012.

[89] M.R. Garey and D.S. Johnson. Computers and intractability: A guide to the theory of np-completeness. *WH Freeman & Co., San Francisco*, 1979.

[90] William E Hart. Python optimization modeling objects (pyomo). In *Operations Research and Cyber-Infrastructure*, pages 3–19. Springer, 2009.

[91] Gurobi Optimization. `http://www.gurobi.com/`.

[92] Oliver Blume, Anton Ambrosy, Michael Wilhelm, and Ulrich Barth. Energy Efficiency of LTE networks under traffic loads of 2020. In *Proceedings of the Tenth International Symposium on Wireless Communication Systems (ISWCS)*, pages 1–5. VDE, 2013.

[93] Rajiv Ramaswami, Kumar Sivarajan, and Galen Sasaki. *Optical networks: a practical perspective*. Morgan Kaufmann, 2009.

[94] Swante Scholz. A genetic algorithm for flow processing-aware controller placement. Bachelor thesis, Paderborn University, 2015.

[95] Brad L Miller and David E Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems*, 9(3):193–212, 1995.

[96] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, 2012.

[97] Navid Ghazisaidi, Martin Maier, and Chadi M Assi. Fiber-wireless (FiWi) access networks: A survey. *IEEE Communications Magazine*, 47(2):160–167, 2009.

[98] I-Fen Chao and Maria C Yuang. Toward wireless backhaul using circuit emulation over optical packet-switched metro WDM ring network. *Journal of Lightwave Technology*, 31(18):3032–3042, 2013.

[99] Averill M. Law and W. David Kelton. *Simulation modeling and analysis*, volume 2. McGraw-Hill New York, 1991.

[100] Ying Zhang and Ake Årvidsson. Understanding the characteristics of cellular data traffic. *ACM SIGCOMM Computer Communication Review*, 42(4):461–466, 2012.

[101] Arnold O. Allen. *Probability, statistics, and queueing theory*. Academic Press, 2014.

[102] Dimitrij Pauls. Distributed flexible reassignment for flow processing-aware controller placement. Bachelor thesis, Paderborn University, 2016.

[103] David Peleg. *Distributed computing: a locality-sensitive approach*. SIAM, 2000.

[104] ComplexNetworkSim Python package. `https://pythonhosted.org/ComplexNetworkSim/`.

[105] Andreas Baumgartner, Varun S Reddy, and Thomas Bauschert. Combined Virtual Mobile Core Network Function Placement and Topology Optimization with Latency Bounds. In *Fourth European Workshop on Software Defined Networks (EWSDN)*, pages 97–102. IEEE, 2015.

[106] Nicolas Pujet and David Rodrian. Network configuration optimization, April 2011. US Patent 7,924,734.

[107] Rasha Al-Naseri. A realistic mobile network model in the context of flow processing-aware controller placement. Master thesis, Paderborn University, 2016.

[108] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[109] NetworkX v1.10 Python package – documentation. `https://networkx.github.io/documentation/networkx-1.10/`.

[110] NGMN Alliance. RAN Evolution Project - CoMP Evaluation and Enhancement. `https://www.ngmn.org/uploads/media/NGMN_RANEV_D3_CoMP_Evaluation_and_Enhancement_v2.0.pdf`.

[111] 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. *IEEE Communications Magazine*, 49(2):102–111, 2011.

[112] Rikke Apelfröjd and Mikael Sternad. Design and measurement-based evaluations of coherent JT CoMP: a study of precoding, user grouping and resource allocation using predicted CSI. *EURASIP Journal on Wireless Communications and Networking*, 2014(1):1–20, 2014.

[113] Tarun Kumar Sarkar. SDN testbed-based evaluation of flow processing-aware controller placement. Master thesis, Paderborn University, 2017.

[114] MaxiNet. https://maxinet.github.io/.

[115] Philip Wette, Martin Dräxler, Arne Schwabe, Felix Wallaschek, Mohammad Hassan Zahraee, and Holger Karl. Maxinet: Distributed Emulation of Software-Defined Networks. In *IFIP Networking Conference*, pages 1–9. IEEE, 2014.

[116] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[117] socat – SOcket CAT (man page). https://linux.die.net/man/1/socat.

[118] Linux Traffic Control (man page). http://man7.org/linux/man-pages/man8/tc.8.html.

[119] NetEm – Network Emulator (man page). http://man7.org/linux/man-pages/man8/tc-netem.8.html.

[120] Stephen Hemminger et al. Network emulation with NetEm. In *linux.conf.au*, pages 18–23, 2005.

[121] Iperf. https://iperf.fr/.

[122] Peter JM Van Laarhoven and Emile HL Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

[123] James Kennedy. Particle swarm optimization. In *Encyclopedia of machine learning*, pages 760–766. Springer, 2011.

[124] Rob J. Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2014.

[125] Sevil Dräxler, Holger Karl, Manuel Peuster, Hadi Razzaghi Kouchaksaraei, Michael Bredel, Johannes Lessmann, Thomas Soenen, Wouter Tavernier, Sharon Mendel-Brin, and George Xilouris. Sonata: Service programming and orchestration for virtualized software networks. In *International Conference on Communications (ICC) Workshops*, pages 973–978. IEEE, 2017.