UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Scaling, Placement, and Routing for Pliable Virtualized Composed Services

Dissertation

by
Sevil Dräxler, née Mehraghdam

accepted by the
Faculty of Electrical Engineering, Computer Science, and Mathematics
Paderborn University

in partial fulfillment of the requirements for the degree of
Doctor rerum naturalium (Dr. rer. nat.)

Referees:
Prof. Dr. Holger Karl, Paderborn University, Germany
Prof. Dr. Giuseppe Bianchi, University of Roma Tor Vergata, Italy

Submission date: 28.05.2019
Examination date: 22.07.2019

Abstract

Next-generation networks are currently being shaped by the softwarization of service components and the virtualization of resources. New approaches are being developed to control the compute, storage, and networking resources and to create, compose, and orchestrate different application components and network functions for offering services. The resource demands and the topology of virtualized composed services are, however, still fixed and pre-defined using rigid and inaccurate descriptors, usually created manually in current approaches. This jeopardizes the correctness of decisions for resource management and service orchestration and can easily result in over- or under-utilization of resources.

The aim of this dissertation is to address this issue by introducing virtualized composed services, which have flexible structures and load-adaptive resource demands. In particular, different types of pliable virtualized composed services are described that consist of components, which (i) can be defined with a *partial order* and can be composed in different ways according to the availability of resources and the load, (ii) have their resource demands specified as a function of load and can be *scaled horizontally and vertically* depending on the load, or (iii) are developed in different *deployment versions*, each using different sets of virtual and physical resources, resulting in different characteristics, e.g., in terms of cost and performance.

For pliable virtualized composed services, scaling, placement, and routing approaches are presented. These approaches can be used in modern service management and orchestration frameworks, to adapt the resource allocation and the topology of services to the requirements of service users, service providers, and network operators. Simulation-based analyses show the feasibility of defining services in a flexible and adaptable way, unveiling a new degree of freedom in service management and orchestration decisions.

Zusammenfassung

Die Struktur zukünftiger Netze wird insbesondere durch softwarebasierte Dienstkomponenten und Virtualisierung von Ressourcen bestimmt werden. Aktuell werden neue Ansätze und Verfahren entwickelt, um Verarbeitungs-, Speicher- und Netzressourcen zu kontrollieren und um damit Dienste anzubieten und Anwendungskomponenten zu erstellen, zusammenzustellen und zu orchestrieren. Demgegenüber steht eine bisher feste, vordefinierte, manuelle und damit ungenaue Beschreibung von Ressourcenanforderungen und den Topologien von virtualisierten Dienstkomponenten. Dies gefährdet die Korrektheit der Entscheidungen für Ressourcenverwaltung und Dienstorchestrierung und führt leicht zu einer zu hohen oder zu geringen Auslastung der Ressourcen.

Um dieses Problem anzugehen, werden in dieser Arbeit virtualisierte zusammengestellte Dienste vorgestellt, die über flexible Strukturen und lastadaptive Ressourcenanforderungen verfügen. Insbesondere werden verschiedene Arten von formbaren virtualisierten zusammengestellten Diensten vorgestellt, die (i) mit einer *partiellen Reihenfolge* definiert sind und auf unterschiedliche Weise in Abhängigkeit der verfügbaren Ressourcen und der Last zusammengestellt werden können, (ii) Ressourcenanforderungen als Funktion über die Last spezifizieren und automatisch lastabhängig *horizontal und vertikal skaliert* werden können, oder (iii) so entwickelt sind, dass sie über verschiedene *Einsatzimplementierungen* verfügen, die jeweils spezifische virtuelle oder physikalische Ressourcen nutzen und so unterschiedliche Charakteristiken in Bezug auf Kosten oder Leistungsfähigkeit bieten.

Ansätze für Skalierung, Platzierung und Lenkung von formbaren virtualisierten zusammengestellten Diensten werden präsentiert, die in aktuellen Implementierungsmodellen für Dienstverwaltung und Orchestrierung verwendet werden können, um Ressourcenzuweisungen und Diensttopologien an die Anforderungen der Dienstnutzer, Dienstanbieter und Netzbetreiber anzupassen. Simulationsbasierte Analysen zeigen die Umsetzbarkeit einer flexiblen und anpassbaren Definition von Diensten und veranschauliche damit einen neuen Freiheitsgrad bei der Ressourcenverwaltung und für Orchestrierungsentscheidungen.

In loving memory of my grandfather, Ababa

Acknowledgements

I would like to thank my advisor Prof. Dr. Holger Karl for guiding, encouraging, and criticizing me throughout the research that has led to this dissertation. I have always enjoyed our discussions and being challenged by his questions. I am thankful and fortunate that I had the chance to work with him and learn from him.

My sincere thanks also go to Dr. Florentin Neumann for introducing me to the Computer Networks research group in Paderborn University. I have had close collaborations and interesting discussions with my colleagues from this research group and the research projects where I was involved, which have contributed to shaping this dissertation. I am especially grateful to Dr. Zoltán Ádám Mann, Stefan Schneider, Manuel Peuster, and Haitham Afifi for the joint work and the fruitful discussions that have enriched my dissertation.

I gratefully acknowledge the funding I have received for my research from the International Graduate School "Dynamic Intelligent Systems" in Paderborn University, the German Research Foundation (DFG) through the Collaborative Research Center "On-The-Fly Computing" (SFB 901), as well as the SONATA and 5G-PICTURE projects within the European Union's Research and Innovation Programme.

I am deeply thankful to Faezeh for leading me to Paderborn and helping me change my life for the better. The journey leading to this dissertation could not have started without her.

I would also like to thank Martin for his love, his enthusiasm to hear about my work, his patience at the times I was frustrated, and the inspiring discussions that helped me move forward.

I cannot thank Maman, Baba, and Khazar enough for always believing in me, having my back, and giving me the courage and strength to free myself from limitations and obstacles. This work would not have been possible without their support.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

ADC	Application Delivery Controller
BNG	Broadband Network Gateway
CDN	Content Delivery Network
DPI	Deep Packet Inspector
EBNF	Extended Backus-Naur Form
ETSI	European Telecommunications Standards Institute
FPGA	Field-Programmable Gate Array
FW	Firewall
GPU	Graphics Processing Unit
IDS	Intrusion Detection System
IETF	Internet Engineering Task Force
LB	Load Balancer
LI	Lawful Interception
LRP	Location-Routing Problem
MANO	Management and Orchestration
MILP	Mixed-Integer Linear Program
MIP	Mixed-Integer Program
MIQCP	Mixed-Integer Quadratically Constrained Program
MSE	Mean Squared Error
NAT	Network Address Translator
NF	Network Function
NFV	Network Function Virtualization
PGW	Packet Data Network Gateway

PNF	Physical Network Function
PR	Polynomial Regression
ROID	Ratio of Outgoing to Incoming Data Rate
SCND	Supply Chain Network Design
SDN	Software-Defined Networking
SFC	Service Function Chaining
SLA	Service-Level Agreement
SVR	Support Vector Regression
TiS	Time in System
vCPU	Virtual CPU
VCS	Virtualized Composed Service
VIM	Virtualized Infrastructure Manager
VM	Virtual Machine
VNE	Virtual Network Embedding
VNF	Virtual Network Function
VNFFG	Virtual Network Function Forwarding Graph
VNMP	Virtual Network Mapping Problem
WOC	Web Optimization Controller

1 Introduction

Services like video streaming, on-line gaming, mobile connectivity, etc., consist of various hardware- and software-based components. These components are hosted on top an of infrastructure managed by a *network operator*. As shown in Figure 1.1, the network operator has access to compute, storage, and networking resources (owned by itself or provided by external *infrastructure operators*) and is responsible for the management and orchestration of multiple services from different *service providers*.

Using the available resources, the hardware- and software-based components provide Network Functions (NFs) [1] and application-specific capabilities. The intended functionality of the overall service with the intended performance is applied to the network flows by routing these flows through the right components, deployed using the most suitable technology in appropriate locations. Different components can modify the traversing flows in different ways. For example, a Deep Packet Inspector (DPI) can split the incoming flows over different branches according to the type of the inspected packets, each branch receiving a fraction of the data rate of the incoming flow. Firewalls can drop certain packets, resulting in flows with a lower data rate than incoming flows. A video optimizer can change

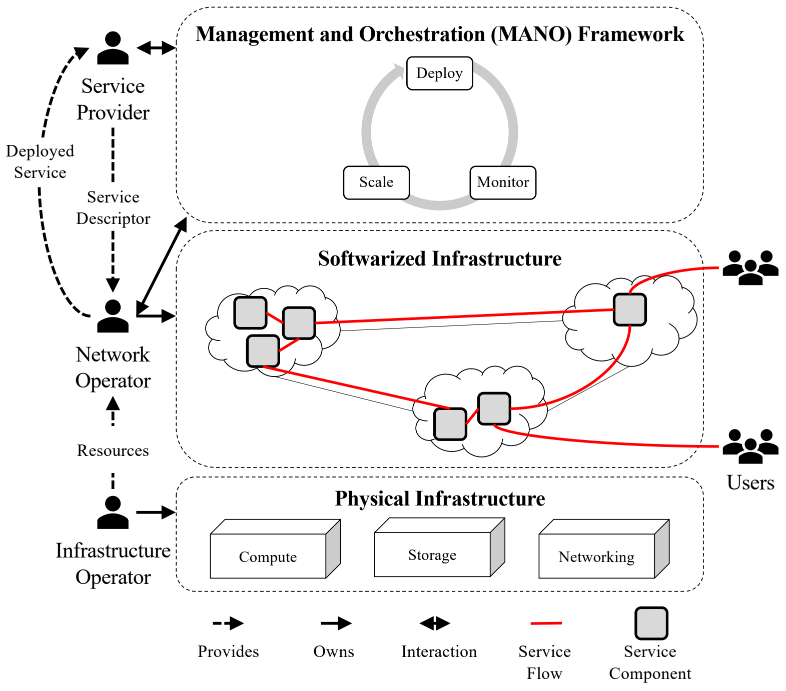


Figure 1.1: Different stakeholders in a service management and orchestration scenario

the encoding of the video for upscaling or downscaling, which can result in a higher or lower data rate compared to the original flow data rate, respectively.

In this dissertation, I refer to a composition of hardware- and software-based NFs and application-specific components as a **Virtualized Composed Service (VCS)** and to the included building blocks of it as **service components**.

The functionalities of the service components were traditionally provided using closed, proprietary elements, e.g., middleboxes offering business- or operations-related functionalities (like load balancing, deep packet inspection, HTTP header enrichment, etc.), or data bases, video optimizers, etc., deployed locally at the service provider premises. In such a setup, modifying the way service components are connected together, changing the placement of functions in the network, and scaling the service require complex modifications, such as changing the network topology or installing new equipment in the network.

The variety of services and the requirements of the involved stakeholders (shown in Figure 1.1), e.g., in terms of service latency, data rate, resilience, reliability, availability, mobility, and energy efficiency [2] are continuously increasing. To fulfill these requirements, next-generation networks need an extremely higher flexibility [3] in how the services can be defined, deployed, managed and modified, compared to the rigid ways of service management and orchestration in traditional networks.

Network softwarization [4, 5] promises such a transition, which will allow services and their components to be designed, developed, and tested efficiently and their deployment and life-cycle management steps (including on-boarding, starting, updating, stopping, etc. [6]) to be performed with no or limited human intervention. In softwarized networks, the NFs and the application-specific components are provisioned as virtual instances, using different virtualization technologies, depending on the requirements of the service and the capabilities of the infrastructure. This is facilitated by concepts like Network Function Virtualization (NFV) [7], Software-Defined Networking (SDN) [8], and Service Function Chaining (SFC) [9]. These key enablers [10] complement the developments in the areas of cloud [11] and interconnected cloud computing [12] as well as fog [13] and mobile edge computing [14].

To provide large capacities to host service components, where and when they are needed, a powerful and comprehensive **heterogeneous infrastructure** is needed that can offer compute, storage, and connectivity using different technologies, distributed over a large-scale network. Such an infrastructure may span different administrative domains of different network operators, offering different geographical and technological options for deploying services of different service providers.

In these large-scale networks, several **instances** (i.e., concrete instantiations) of service components are deployed, e.g., to serve the requirements of different *users*, possibly residing in different geographical locations. The required number of instances for service components as well as the type and the number of required compute, storage, and networking resources depend on the load that the service is expected to handle. These requirements are typically specified by the service provider. Similarly, the service provider defines the overall structure of the VCS and the included service components, depending on the intended functionality of the service.

In existing approaches, the resource demands of service components are defined within *descriptors* [6] and are given, for example, in form of different *flavors* [6] for service components, i.e., a pre-defined set of resources like CPU, memory, and storage that define the size and configuration of a virtual server that can be launched on a target platform [15]. The exact number of instances required for each component should also be defined so that the cloud and NFV Management and Orchestration (MANO) frameworks can calculate a placement for the instances and map their inter-connections to the network links.

Typically, the structure of a VCS is modeled as a directed **service graph** consisting of Virtual Network Functions (VNFs), Physical Network Functions (PNFs), cloud-based micro-services, or application components as nodes and the data flows between pairs of service components as edges of the graph. The simplest case for a VCS is a linear chain of at least one service component connected to an endpoint, like a front-end server for a group of users. Inserting service components that can split or merge network flows over different paths makes the structure of a VCS more complex.

The way the resource demands and the structure of VCSs are modeled and specified directly affects the way VCSs can be managed and orchestrated, as decisions about the life cycle of VCS are taken based on the provided specifications. In Section 1.1, I describe some of the problems of existing approaches to modeling VCSs and the life-cycle management decisions based on these descriptors. I also elaborate on the opportunities offered by network softwarization and heterogeneous infrastructures and propose solutions that can bring more precision and flexibility to the life-cycle management of VCSs. In Section 1.2, I describe my methodology in approaching the mentioned problems. Section 1.3 includes the list of publications I have (co-)authored as well as the implementation of the work that has led to this dissertation. In Section 1.4, I describe the structure of the remainder of the dissertation.

1.1 Problems and Opportunities

I have identified the following problems and opportunities.

1.1.1 Inflexible Ordering of Service Components

In conventional descriptors, the structure of the VCS is always fixed, following the initial description of its *service graph*, i.e., the set of required service components with a fixed number of instances and their inter-connections.

Based on the dependencies among a set of service components [16], typically, a fixed structure is assumed for services. For instance, if the packets have to go through a WAN optimizer and an Intrusion Detection System (IDS), the packet inspection by the IDS should typically be carried out before the WAN optimizer compresses the contents. In such a case, a special attention to the *order* of traversing the service components is required.

However, in case the components of a VCS do not have such a dependency, there can be multiple possibilities for composing the service. Depending on how each component modifies the data rate of the flows, different composition options can have different impacts on the traffic in the network links, on application performance, or on the latency.

As a result of network softwarization, modifying the deployed VCSs is not as complex as in non-virtualized setups. For example, flexible MANO solutions can be developed that easily modify the *order* of traversing service components, e.g., to switch to a better composition of service components for better resource utilization or service performance.

One shortcoming of the conventional service descriptors and the existing MANO solutions is that they assume a total order among service components and, therefore, are not able to exploit this additional degree of freedom in composing VCSs.

My first proposal to solve this shortcoming is to *under-specify* the structure of VCSs, where possible, rather than defining total orders among all service components. I define **Pliable[1] VCSs with Arbitrarily Ordered Components** in which the order of chaining a subset of components in a VCS can be determined and modified dynamically, provided that the overall functionality of the service is not impaired.

Figure 1.2 shows an example pliable VCS. S and E represent the start and end points of the service, respectively. In this example, network flows need to

[1]Defined [17] as "Easily bent or shaped", "Capable of being changed or adjusted to meet particular or varied needs".

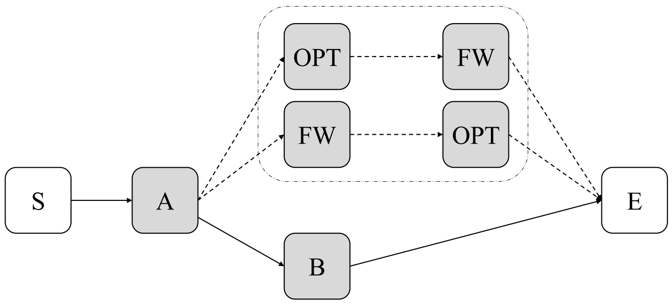


Figure 1.2: Example VCS with arbitrarily ordered components

be processed by two service components, a video optimizer (OPT) and a firewall (FW) as part of a video streaming service. The functionality of these service components and their effect on the flows is independent from each other. Therefore, traversing these functions with the order of OPT→FW or FW→OPT gives the same *processing result*.

In this example, FW is a function that can block certain access attempts and, with that, can reduce the data rate of the traversing flows. Given the historical usage data, the service provider may know that, e.g., 10 % of the incoming flows to this firewall will be dropped. Using this information, different resource allocation plans can be made for composing this VCS; the network path taken by the flows traversing the option FW→OPT can be planned with a lower link capacity than that of the option OPT→FW. In this way, if deploying the service as OPT→FW is not possible because of insufficient link capacity, the MANO system can re-order these functions and try deploying FW→OPT that has lower link capacity requirements.

Similar examples can be imagined with arbitrarily ordered components that can influence the branching structure in the VCS. For example, we assume all the flows of a service need to go through a firewall (FW) and a DPI before being processed by a service component C. A Load Balancer (LB) is required for distributing the load over three instances of C. If the MANO system is allowed to arbitrarily chain the load balancer and the firewall, different deployments of this VCS are possible, e.g.:

- Distributing the flows over three different branches by the load balancer and placing an instance of FW and DPI on each branch, as shown in Figure 1.3a. This option results in a lower data rate on each branch but three instances of the FW and three instances of the DPI are required.
- All flows traversing a single FW instance and a single DPI before reaching the load balancer, as shown in Figure 1.3b. This option requires fewer instances of FW and DPI but the load each instance needs to handle and the load on each outgoing branch of the load balancer is larger than in the previous case.

For pliable VCSs with arbitrarily ordered components, the scaling, placement and routing decisions depends on how the service components are composed. For

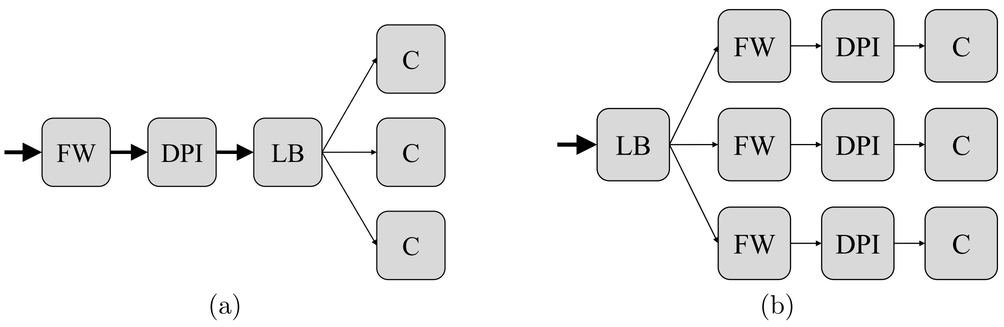


Figure 1.3: Two of the possible chaining options for a set of arbitrarily ordered service components

these VCSs, first of all, I tackle the problem of finding the best possible ordering of the service components. Afterwards, I describe a joint placement and routing problem, which also has a limited scaling functionality integrated in the decision process (described in Section 3.2.3). Therefore, I refer to it as the joint **Scaling, Placement, and RoutING (SPRING) Problem** for pliable VCSs with **arbitrarily** ordered components, shortly the **A-SPRING Problem**.

1.1.2 Limited Precision and Flexibility of Descriptors

One requirement for automated service management and orchestration, as required and targeted in softwarized networks, is the precise specification of the *structure* of VCSs by describing the required service components, their resource demands, their inter-connections, and how they modify the traversing flows.

Usually, service providers rely on the *developers* of individual service components to create dedicated *descriptors* defining the exact amount of compute, storage, and networking resources required for serving the desired amount of traffic for each component. The problem with this approach is that developers of individual components cannot have a precise view of the exact resource demands of every service component or the number of instances required for a service component to handle the load. The performance and the resource usage of different service components are influenced by those of other components in the same VCS, requiring global knowledge of the service for building individual component descriptors. Therefore, the conventional approach to specifying resource demands can easily result in over-/under-allocation of network resources or violation of Service-Level Agreements (SLAs) (i.e., agreements between network operator and service providers or between service providers and users, for example, regarding the performance, availability, and reliability of the provided service or operation).

I propose describing VCSs using *service templates*[2] instead of fixed descriptors. The templates describe the *structure* of the VCSs and express their resource demands *relative to the amount of load* they should handle, e.g.,

[2]The concept is inspired by a comparable approach in distributed cloud computing [18]. I describe the differences of the approaches in Chapter 2.

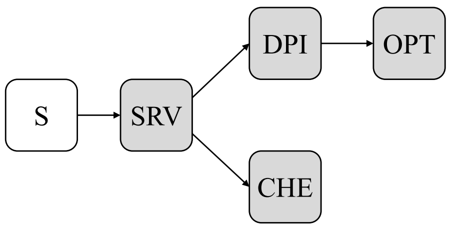


Figure 1.4: Example uni-directional VCS

- The required compute capacity (e.g., CPU and memory) is described for each service component as a function of the input data rate. This can be used to calculate the network node capacity required to host the service component.
- The amount of traffic leaving each service component towards other components is specified as a function of the data rate that enters the component. This can be used to calculate the link capacity required to host the traffic flowing between any two inter-connected instance.

Given additional information about the expected traffic originating from the *sources* of the service flows (e.g., the location of users, content servers, etc.), resource demands can be calculated dynamically. The number and the location of required instances for each service component can also be determined based on traffic. This results in the second category of pliable VCSs that I consider in this dissertation: **Pliable VCSs with Load-Proportional Structures**[3].

The load can be characterized, e.g., as the request rate or the data rate of the corresponding service flows. As the load may constantly be changing, the current traffic needs to be monitored to keep the network service in an optimal state. This reduces the risk of over- or under-estimating the resource demands. Given reasonable predictions about possible changes in the load and based on the functions describing the dependency of resource requirements and outgoing data rates on incoming data rates, it is also possible to plan and predict the required changes to the deployment and their impact, which is a pre-requisite for effective optimization of the services and the underlying network.

1.1.3 Fixed Structure of Uni-Directional VCSs

In most of the existing work in the fields of NFV and cloud computing for resource allocation and service embedding (described in Section 2.2), VCSs are considered to be uni-directional. They are modeled as pre-defined, connected, acyclic, directed graphs and described using the imprecise and inflexible conventional descriptors, as mentioned in Section 1.1.2.

My third proposal in this dissertation, in-line with existing solutions and to introduce more flexibility and precision beyond that, is to define **Uni-Directional**

[3]I use the term load-proportional in this context to express the dependence of the structure of a VCS to the amount of load it has to handle. In this context, proportionality is not intended as a mathematical relation that can be defined between two scalars.

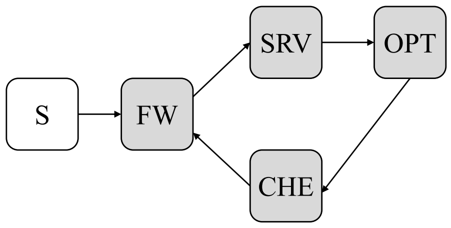


Figure 1.5: Example bi-directional VCS

Pliable VCSs with Load-Proportional Structures using service templates. These VCSs are flexibly defined compositions of service components that are traversed in one forwarding direction (i.e., upstream or downstream) only [16]. Figure 1.4 shows an example service template for a simple uni-directional video streaming VCS. If the requested content is not available in the cache (CHE), the content is streamed through a video streaming server (SRV) towards a DPI. The DPI inspects the request and, accordingly, instructs a video optimizer (OPT) to prepare the suitable content for the requesting device.

1.1.4 Bi-Directional VCSs

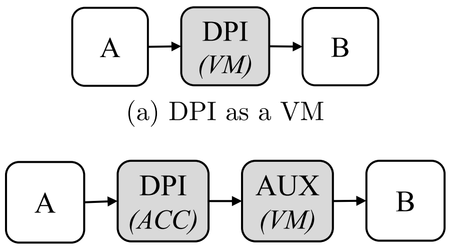
In spite of the practical relevance [16] of *bi-directional* VCSs, they have not been the focus of resource allocation and MANO solutions so far, and hence, there are not many adequate modeling approaches for these services so far. In contrast to the uni-directional VCSs, bi-directional VCSs consist of service components that must be traversed in both directions.

The service components required in each forwarding direction can be defined and specified as two uni-directional VCSs with common components. As certain service components, e.g., stateful firewalls, may store flow-related state information, considering the upstream and downstream traffic independently may result in state inconsistencies. In such cases, defining bi-directional VCSs can help creating a fine-grained control on the service flows and the required configurations for stateful service components.

My fourth proposal is, hence, to model these VCSs as **Bi-Directional Pliable VCSs with Load-Proportional Structures**, defined flexibly using service templates. Figure 1.5 shows an example bi-directional service template that describes the structure of a video streaming service. This example VCS consists of a stateful firewall (FW) for the upstream and downstream flows. The content is streamed using a streaming server (SRV), passes a video optimizer (OPT), and is stored in a cache (CHE) for future requests.

1.1.5 Multi-Version Service Components

In the context of softwarized, heterogeneous networks, there are several attempts towards unifying the tools and mechanisms required for the control and orchestration of distributed infrastructures providing heterogeneous resources [19, 20].



(a) DPI as a VM

(b) Accelerated DPI with an auxiliary function

Figure 1.6: Example service including a DPI in different deployment versions

This would allow deploying and modifying VCSs on a variety of multi-technology network, compute, and storage resources, fast and cost-efficiently.

My fifth proposal in this dissertation is to define **Heterogeneous Pliable VCSs with Load-Proportional Structures**, to fully exploit the advantages of a heterogeneous infrastructure. Using this model, service providers can develop and offer flexibly defined VCSs with components that are produced in different *versions*, i.e., using different software implementations, each made for running on a different resource type, offering different advantages. For example, a DPI can be deployed as a Virtual Machine (VM) at a low cost only using general-purpose hardware. DPI is a network- and compute-intensive network function, so it can achieve a higher performance using special-purpose hardware support, which of course is a more expensive option than a VM.

To conform to different needs of the service users (e.g., low cost vs. high performance), the service provider can submit both a VM and a hardware-accelerated version of the DPI to a network operator that offers general-purpose as well as special-purpose hardware (like Graphics Processing Unit (GPU)). Using appropriate service management and orchestration tools and algorithms, the right version of the DPI can be deployed in the required locations to serve the users. I refer to service components that have different deployment options as *multi-version* service components.

In a more complex scenario, different versions of such a function may consist of different number of components. E.g., the hardware-accelerated version of a DPI might require an additional VM for post-processing its results while the VM version can perform all of the required operations within the same component. In this way, as shown in Figure 1.6, the VCSs that include these DPI versions, have a different *structure*. I refer to such a VCS as a *multi-structure* VCS consisting of multi-version components or a **heterogeneous service**.

1.1.6 Separated Scaling, Placement, Routing Decisions

Figure 1.7 shows a conventional life-cycle management process using typical cloud and NFV MANO frameworks. Typically, the VCSs are deployed by allocating the required resources corresponding to one of the pre-defined flavors in their fixed descriptors.

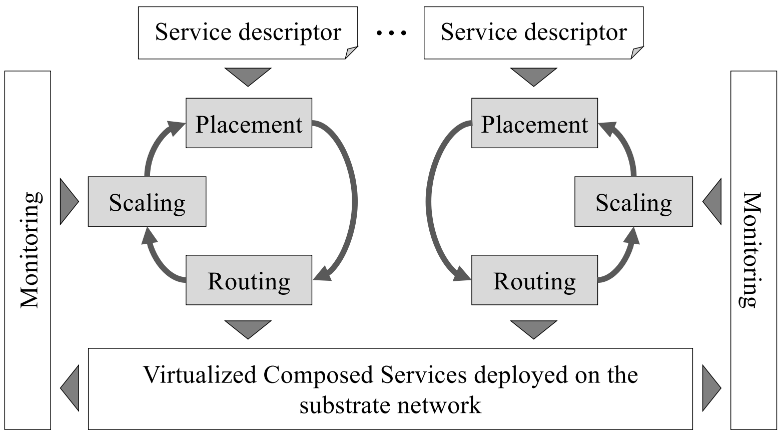


Figure 1.7: Conventional service life cycle, from descriptors to running services

To react to addition and removal of VCSs, fluctuations in the request load of a VCSs, or to serve new user in a new location, some of the following adaptations might be needed:

- Vertical scaling of deployed services by adding or removing instances of service components
- Horizontal scaling of deployed instances of service components by increasing or decreasing the resources allocated to them
- Modifying the placement of the service components
- Re-routing the service flows between the service components over different, more suitable paths

These actions are highly inter-dependent. Given the large number of degrees of freedom for finding the best adaptation, deciding scaling, placement, and routing independently can result in sub-optimal decisions for the network and the running services.

As an example, we can consider a network operator hosting a dynamically changing set of VCSs from different service providers, where each VCS serves dynamically changing user groups that produce dynamically changing data rates. The trade-offs among the conflicting goals of service providers (e.g., high performance, low resource costs, etc.) and network operators (e.g., optimal resource utilization, high service request acceptance ratio, etc.) can be highly non-trivial, for example:

- Placing a compute-intensive service component on a node with limited resources near the source of requests minimizes the delay but placing it on a more powerful node further away in the network minimizes the throughput.
- Allowing a single instance of a data-processing component serve multiple user groups minimizes the number of required instances and the number of

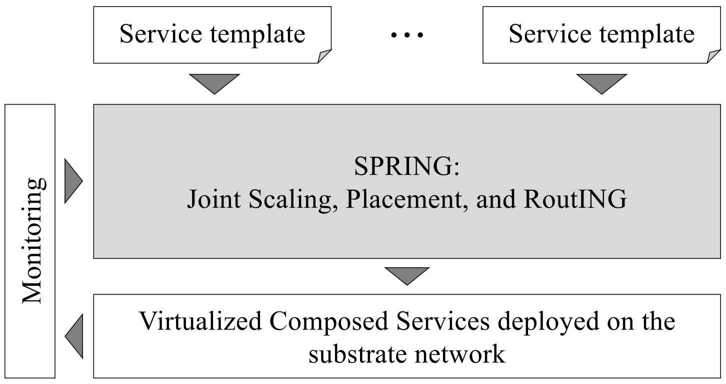

Service template
...
Service template
Monitoring
SPRING:
Joint Scaling, Placement, and RoutING
Virtualized Composed Services deployed on the substrate network

Figure 1.8: Joint scaling, placement, and routing for pliable VCSs

idle resources (like memory) allocated to these instances at low-load situations but using dedicated instances near the sources reduces the network load.

Existing algorithms and MANO solutions decide and perform these actions either completely independently from one another or consider only a subset of scaling, placement, and routing decisions together (as described in Section 2.2).

My sixth proposal, as shown in Figure 1.8, changes the way the life cycle of pliable VCSs is handled, by combining scaling, placement, and routing steps into a joint decision process.

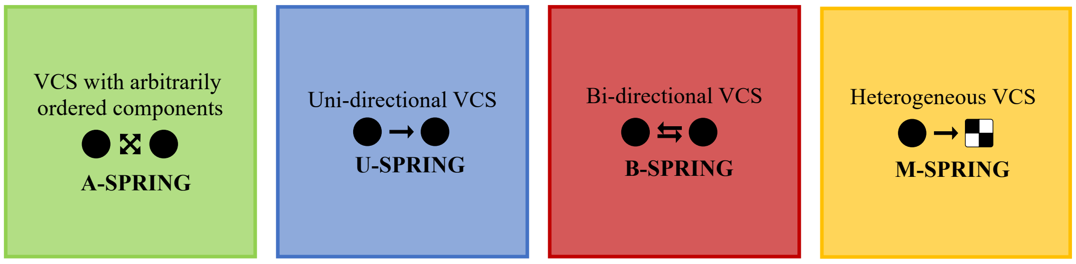
In this approach, depending on the location and data rate of the sources, in a single-step *template embedding* process,

- each service template is scaled out into an overlay with the necessary number of instances for each service component,
- each instance of each required component is mapped to a network node and is allocated the required amount of resources on that node,
- the inter-connections among the instances are mapped to paths along network links, with the required data rate.

This approach can be used upon initial deployment of a VCS and for adapting and optimizing the VCSs that are already embedded into the network, e.g., when the amount of traffic changes or when new user locations need to be supported. I consider all VCSs that should be mapped into the same substrate network together. In this way, newly requested and already deployed services are optimized jointly, allowing a global optimum to be achieved.

In this dissertation, I address the problem of joint *Scaling, Placement, and RoutING (SPRING)* for uni-directional VCSs (described in Chapter 6) without loops in their structure, where only upstream *or* downstream flows traversing the service components are considered. I refer to this problem as the **U-SPRING Problem**.

I also describe an extended version of this problem that supports bi-directional VCSs (described in Chapter 7), where both upstream *and* downstream service



(a) A-SPRING (b) U-SPRING (c) B-SPRING (d) M-SPRING

Figure 1.9: Symbols used to refer to the SPRING approaches in this dissertation

flows are considered, allowing service flows to return to their sources after traversing the required service components. I call this problem the **B-SPRING Problem**.

Finally, I describe another variant of the SPRING problem for heterogeneous pliable VCSs (described in Chapter 8). These VCSs consist of multi-version components that have been developed and prepared with different (software) versions. Each version can be deployed on different hosting platforms, e.g., a VM version of a certain service component might use CPUs and an accelerated version of it may require additional special-purpose hardware like GPUs or Field-Programmable Gate Arrays (FPGAs). I refer to this problem as the **M-SPRING Problem**.

Figure 1.9 shows an overview of the SPRING approaches and the symbols that I have used throughout this dissertation as indicators of the type of the pliable VCS being discussed in each approach, e.g., at the bottom of the pages in the corresponding chapters.

1.2 Methodology

For each category of pliable VCSs, I have taken the following steps to produce my contributions:

- I have formulated a SPRING problem as a multi-objective optimization problem that can be solved using an appropriate optimizer to get optimal solutions. I have used the Gurobi Optimizer [21] to analyze the optimal solution to the problem.

- I have presented a heuristic that can solve the SPRING problem in a sub-optimal but quick manner.

- I have evaluated the results from the optimization and heuristic approaches using simulation-based analysis, considering different objectives of service users, service providers, and network operators.

I have designed the evaluation scenarios and the optimization objectives independently for each category of pliable VCSs to highlight the exclusive features and capabilities of the models and approaches in each case. More details regarding the solutions are described in the corresponding chapters.

1.3 Contributions

I have (co-)authored the following publications and manuscripts:

- S. Mehraghdam, M. Keller, and H. Karl. “Specifying and Placing Chains of Virtual Network Functions”. In: *3rd International Conference on Cloud Networking (CloudNet)*. IEEE. Oct. 2014, pp. 7–13. DOI: `10.1109/CloudNet.2014.6968961`.
- S. Mehraghdam and H. Karl. “Specification of Complex Structures in Distributed Service Function Chaining Using a YANG Data Model”. In: *CoRR* abs/1503.02442 (2015). arXiv: `1503.02442`. URL: `http://arxiv.org/abs/1503.02442`.
- S. Mehraghdam and H. Karl. “Placement of Services with Flexible Structures Specified by a YANG Data Model”. In: *2nd IEEE International Conference on Network Softwarization (NetSoft)*. IEEE. June 2016. DOI: `10.1109/NETSOFT.2016.7502412`.
- H. Karl, S. Dräxler, M. Peuster, A. Galis, M. Bredel, A. Ramos, J. Martrat, M. S. Siddiqui, S. V. Rossem, W. Tavernier, et al. “DevOps for Network Function Virtualisation: An Architectural Approach”. In: *Transactions on Emerging Telecommunications Technologies* 27.9 (2016), pp. 1206–1215. DOI: `10.1002/ett.3084`.
- S. Dräxler, H. Karl, and Z. Á. Mann. “Joint Optimization of Scaling and Placement of Virtual Network Services”. In: *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGrid ’17. Madrid, Spain: IEEE Press, May 2017, pp. 365–370. ISBN: 978-1-5090-6610-0. DOI: `10.1109/CCGRID.2017.25`. URL: `https://doi.org/10.1109/CCGRID.2017.25`.
- S. Dräxler, H. Karl, M. Peuster, H. R. Kouchaksaraei, M. Bredel, J. Lessmann, T. Soenen, W. Tavernier, S. Mendel-Brin, and G. Xilouris. “SONATA: Service Programming and Orchestration for Virtualized Software Networks”. In: *IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE. May 2017, pp. 973–978. DOI: `10.1109/ICCW.2017.7962785`.
- M. Peuster, S. Dräxler, H. R. Kouchaksaraei, S. V. Rossem, W. Tavernier, and H. Karl. “A Flexible Multi-PoP Infrastructure Emulator for Carrier-Grade MANO Systems”. In: *3rd IEEE International Conference on Network Softwarization (NetSoft) Demo Track*. IEEE. July 2017. DOI: `10.1109/NETSOFT.2017.8004250`.
- S. Dräxler and H. Karl. “Specification, Composition, and Placement of Network Services with Flexible Structures”. In: *International Journal of Network Management* 27.2 (2017). DOI: `10.1002/nem.1963`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.1963`.

- S. Dräxler, M. Peuster, M. Illian, and H. Karl. “Towards Predicting Resource Demands and Performance of Distributed Cloud Services”. In: *KuVS-Fachgespräch Fog Computing 2018*. Technical Report. Mar. 2018. URL: `http://www.infosys.tuwien.ac.at/docs/proceedings.pdf#page=14`.

- S. Dräxler, M. Peuster, M. Illian, and H. Karl. “Generating Resource and Performance Models for Service Function Chains: The Video Streaming Case”. In: *4th IEEE International Conference on Network Softwarization (NetSoft)*. IEEE. June 2018. DOI: `10.1109/NETSOFT.2018.8460029`.

- S. Dräxler, S. Schneider, and H. Karl. “Scaling and Placing Bidirectional Services with Stateful Virtual and Physical Network Functions”. In: *4th IEEE International Conference on Network Softwarization (NetSoft)*. IEEE. June 2018. DOI: `10.1109/NETSOFT.2018.8459915`.

- S. Dräxler, H. Karl, H. R. Kouchaksaraei, A. Machwe, C. Dent-Young, K. Katsalis, and K. Samdanis. “5G OS: Control and Orchestration of Services on Multi-Domain Heterogeneous 5G Infrastructures”. In: *2018 European Conference on Networks and Communications (EuCNC)*. June 2018. DOI: `10.1109/EuCNC.2018.8443210`.

- H. R. Kouchaksaraei, S. Dräxler, M. Peuster, and H. Karl. “Programmable and Flexible Management and Orchestration of Virtualized Network Functions”. In: *2018 European Conference on Networks and Communications (EuCNC)*. June 2018. DOI: `10.1109/EuCNC.2018.8442528`.

- S. Dräxler, H. Karl, and Z. Ádám. “JASPER: Joint Optimization of Scaling, Placement, and Routing of Virtual Network Services”. In: *IEEE Transactions on Network and Service Management* 15.3 (Sept. 2018), pp. 946–960. ISSN: 1932-4537. DOI: `10.1109/TNSM.2018.2846572`.

- S. Schneider, S. Dräxler, and H. Karl. “Trade-offs in Dynamic Resource Allocation in Network Function Virtualization”. In: *1st Workshop on Advanced Control Planes for Software Networks (ACPSN) at IEEE Global Communications Conference (GLOBECOM)*. IEEE. Dec. 2018. DOI: `10.1109/GLOCOMW.2018.8644352`.

- S. Dräxler and H. Karl. “SPRING: Scaling, Placement, and Routing of Heterogeneous Services with Flexible Structures”. In: *5th IEEE International Conference on Network Softwarization (NetSoft)*. IEEE. June 2019.

The source code used for the evaluation of the optimization and heuristic approaches presented in this dissertation is available online [37].

1.4 Structure of the Dissertation

The remainder of this dissertation is structured as follows.

Chapter 2: State of the Art and Related Work In this chapter, I describe the state of the art in modeling VCSs in softwarized networks. I also give an overview of the theoretical background and related work regarding the placement, scaling, and routing problems for VCSs. For this, I focus on the (distributed) cloud computing and NFV areas.

Chapter 3: Services with Arbitrarily Ordered Components In this chapter, I describe the challenges of modeling and orchestrating pliable VCSs with arbitrarily ordered components. As a background for the contributions in the next chapter, I describe the model and optimization approach from my previous research on the A-SPRING problem.

Chapter 4: Embedding Services with Arbitrarily Ordered Components This chapter includes the formal problem description for service graph selection and service graph embedding for pliable VCSs with arbitrarily ordered components.

Chapter 5: Services with Load-Proportional Structures In this chapter, I describe the challenges of modeling and handling pliable VCSs with load-proportional structures, including uni-directional, bi-directional, and heterogeneous VCSs. I also present the results of some experiments showing the feasibility of modeling resource demands of service components as a function of the load they need to handle. Such functions are used in the SPRING approaches presented in the rest of the dissertation.

Chapter 6: Embedding Uni-Directional Services with Load-Proportional Structures This chapter includes the uni-directional pliable VCS model as well as the problem description, optimization and heuristic approaches to the U-SPRING problem. It also includes the evaluation results of the proposed approaches.

Chapter 7: Embedding Bi-Directional Services with Load-Proportional Structures In this chapter, I present the bi-directional pliable VCS model and the problem description, optimization and heuristic approaches to the B-SPRING problem, as well as the evaluation results of the proposed approaches.

Chapter 8: Embedding Heterogeneous Services with Load-Proportional Structures In this chapter, I describe the model for heterogeneous pliable VCS with multi-version service components and the problem description, optimization and heuristic approaches to the M-SPRING problem. I also present the evaluation results of the proposed approaches.

Chapter 9: Results and Future Research Directions In the final chapter, I describe how the models and approaches presented in this dissertation contribute to tackling the shortcomings of existing approaches and making use of the opportunities offered by network softwarization, which are described in Section 1.1. I also discuss the practical applicability of the approaches and the requirements to enable adoption of the solutions beyond experimental scenarios. Finally, I give an overview of possible future research directions.

Chapter 3 and 4 are based on work I have partly done together with M. Keller and H. Karl [38, 22, 23, 24, 29]. Chapters 5–7 are based on work I have partly done together with M. Peuster, M. Illian, Z. Á. Mann, S. Schneider, and H. Karl [26, 31, 32, 34, 30, 39, 40]. Chapter 8 is based on the work I have partly done together with H. Karl [36]. These chapters partially include figures and verbatim copies of the text from the corresponding publications. To ease the flow of reading, such copies from my own publications are not explicitly marked as such, yet all sources are mentioned in each chapter. Parts of these publications, in which I did not have a significant contribution are not included in this dissertation.

2 State of the Art and Related Work

In this chapter, I first describe the state of the art in modeling approaches for Virtualized Composed Services (VCSs) and their components. Afterwards, I position the scaling, placement, and routing approaches described in this dissertation against related work and existing studies.

2.1 Modeling Virtualized Composed Services

In this section, I give an overview of how VCSs and service components are modeled and described in the context of cloud computing and Network Function Virtualization (NFV), focusing on different aspect.

2.1.1 Services with a Pliable Structure

The majority of related studies assumes a fixed and pre-defined structure for VCSs. For example, in the cloud computing context, Sun et al.[41] have published a survey of service description languages, all assuming a fixed structure. There are, however, a limited number of studies that foresee a pliable structure for VCSs.

Keller et al.[18] have proposed a template-based description of service structures, defining the structure of distributed cloud applications using generic templates that can be modified and adapted during and after deployment. The service templates described in this dissertation are more powerful than that model; e.g., the service templates in Chapter 6 express the resource demands and outgoing data rates of service components as functions of their input data rates and in Chapter 7, they allow describing bi-directional VCSs, in which (possibly different) service components are traversed by upstream and downstream service flows.

2.1.2 Arbitrarily Ordered Components

Related to the A-SPRING model, Beck and Botero [42] use a model of pliable VCSs with arbitrarily ordered components. While their model of these VCSs is similar to those of the A-SPRING problem, they tackle the problem of deciding the order of service components and placing them in the substrate network in one step (unlike the two-step approach to the A-SPRING problem in this dissertation).

2.1.3 Bi-Directional Services

Schneider et al. [43] present a model for describing and analyzing VCSs using Queuing Petri Nets. The model is designed for unambiguous specification of the structure of VCSs and the behavior of service components (Virtual Network Functions (VNFs) in their model). The model supports bi-directional structures with loops, can express processing delays of service components, as well as the relationship of outgoing data rate to incoming data rate at each service component. This relationship is considered in several other placement approaches for VCSs and service components, e.g., by Ma et al. [44, 45], Gao et al. [46], and Addis et al. [47]. The pliable VCSs with arbitrarily ordered components cannot be specified using the model of Schneider et al. but for the remaining SPRING approaches described in this dissertation, this model can be used for building accurate service templates.

2.1.4 Heterogeneous Services

Moens and De Turck [48] use a model of VCSs that consist of different VNFs, some of them requiring dedicated physical hardware and some of them deployable using virtual resources. They differentiate between Virtual Machine (VM) requests, which can only be served using physical resources, on the one hand and service requests, which can be fulfilled using dedicated hardware as well as shared virtual resources, on the other hand. The M-SPRING model is more flexible than that model, in the sense that the type of resources that can be allocated to the service components is only limited by the template that describes it. Razzaghi Kouchaksaraei et al. [49] consider VCSs that consist of a combination of VNFs and cloud-based micro-services. None of these models consider different deployment options (versions) for service components, as it is done in the M-SPRING approaches presented in this dissertation.

2.1.5 Standards and Implementations

The model used for specifying the connections and relationships among service components in most of the existing NFV Management and Orchestration (MANO) solutions is based on and similar to the Network Service Descriptor and the Virtual Network Function Forwarding Graph (VNFFG) description defined by the European Telecommunications Standards Institute (ETSI) NFV Industrial Specification Group [6], for example, in SONATA [50], UNIFY [51], T-NOVA [52], and OSM [53] projects. The models for the structure of pliable VCSs presented in this dissertation are compatible with the ETSI descriptors and can be implemented as an extension of them.

Moens and Volckaert [54] have published a survey of different modeling strategies for VCSs. They analyze the trade-off between flexibility and management complexity for service modeling approaches. The pliable VCS models described in this dissertation bring along new complexities and require management and orchestration mechanisms that can make use of the new degrees of freedom offered by the ability to change the structure of VCSs.

2.2 Scaling, Placement, and Routing Problems

The SPRING problem is a joint, single-step optimization of scaling, placement, and routing for pliable VCSs. In general, the SPRING approaches presented in this dissertation can be applied in different contexts, e.g., (distributed) cloud computing and NFV. In this section, after an analysis of related approaches from a theoretical point of view, I give an overview of related work in the cloud computing and NFV areas. The major difference among the existing work in these two fields is usually the abstraction level considered for the substrate network and the resulting assumptions for the model. In particular, in the cloud computing context, embedding is typically done on top of physical machines in data centers, while in the NFV context, the embedding is done on top of geographically distributed points of presence.

2.2.1 Theoretical Framework

The VCS placement and routing sub-problems of the SPRING problem have similarities to the Supply Chain Network Design (SCND) [55] (NP-hard) problem. SCND is a variant of the facility location problem and aims to open a chain of facilities of different types such that the demand is satisfied and the facility or transportation costs are minimized. This corresponds to finding the optimal placement of service components of a VCS in a substrate network. Location-Routing Problem (LRP) [56, 57] is another related problem that aims at the placement of components while reducing the costs in nodes, edges or paths. In this problem, each path has one start point and one end point. The SPRING problem needs to create several paths between different pairs of service components and connect these paths to compose the VCS. In this case, the routing problem turns into a multi-commodity flow problem [58] with inter-commodity dependencies.

Without considering the scaling aspect of the SPRING problem, it is also related to the Virtual Network Embedding (VNE) problem. The VNE problem is a variant of the multi-commodity flow problem. These problems allocate virtual network nodes and connect them together through links with constraints while trying to minimize costs. This corresponds to finding the optimal location for the components of a VCS and connecting them through optimal paths in the SPRING problem. The VNE problems treat the nodes of the virtual networks (corresponding to the service components in this dissertation) independently. In the B-SPRING problem, flows from different tenants can share and reuse service components.

In contrast to static VNE solutions that consider the initial mapping process only, in the U-SPRING, B-SPRING, and M-SPRING problem, I also deal with optimizing and modifying already embedded templates. Some VNE solutions, for example, Houidi et al. [59], can modify the mapping in reaction to node or link failures. The modifications in their work, however, are limited to recalculating the location for the embedded virtual network, i.e., migrating some of the nodes and changing the corresponding paths among them. In addition to these modifications, the SPRING approaches presented in this dissertation can also modify the *structure* of the graph to be embedded (i.e., the VCS) by adding or removing instances of components and their inter-connections, changing the order of the components (A-SPRING), or replacing a component with other components (M-SPRING), if necessary.

In VNE, virtual networks have a fixed size and structure. Therefore, the number of required virtual nodes and their inter-connections have to be determined in separate steps. The SPRING approaches perform scaling, placement, and routing in a single step. In this way, they take the characteristics of the substrate network and the service flows into account leading to better solutions. The SPRING approaches also consider the changing data rates of the service flows, resource requirements depending on the load, and latencies between instances. These aspects are usually not considered in VNE approaches.

Among recent VNE studies, some consider a heterogeneous substrate network, as in the M-SPRING problem. For example, Li et al. [60] propose a joint resource allocation and VNE solution in 5G core networks. They enable efficient physical resource sharing by optimizing the resource demands before embedding. The nodes of the virtual networks in their approach (the service components in my models), however, have a pre-defined number of instances and a fixed deployment version. Baumgartner et al. [61] consider the VNE problem in the mobile core network, optimizing the structure of the the virtual core network service chain. The flexibility of the service structure in their solution is limited to the way a fixed number of VNFs are grouped and distributed.

2.2.2 Cloud Computing Context

Resource allocation is an important problem in the field of (distributed) cloud computing [62]. The problem is typically formulated as resource allocation for individual components. Scaling and placing instances of VMs on top of physical machines while adhering to capacity constraints are the usual problems tackled in this context [63, 64]. The communication among different virtual machines, however, is usually left out or considered only in a limited sense [65]. Even the approaches that consider the communication among virtual machines [66, 67, 68, 69] do not include routing decisions, whereas the SPRING problem also includes routing.

Relevant to the placement sub-problem of the SPRING problem, Bellavista et al. [70] focus on the technical issues of deploying flexible cloud infrastructure, including network-aware placement of multiple virtual machines in virtual data centers. Wang et al. [71] study the dynamic scaling and placement problem for network services in cloud data centers, aiming at reducing costs. These papers also do not address routing.

Keller et al. [18] consider an approach similar to the SPRING problem in the context of distributed cloud computing. The terminology used in this dissertation is partly based on their work but there are important differences in the assumptions and the models. In contrast to their model, where the number of users determines the number of required instances, the deciding factor in SPRING approaches is the *data rate* originating from different sources. Data rate can be represented, for example, as requests or bits per second and is a more observable parameter in practical applications and gives a more fine-grained control over the embedding process. Moreover, the SPRING approaches do not enforce strict scaling restrictions for components as done in their work. For example, their method needs as input the exact number of instances of a back-end server that is required behind a front-end server. Also, the optimization objective in their model is limited to minimizing the total number of instances for embedded templates. The SPRING approaches are multi-objective optimization problems where different metrics like processing capacity of network nodes, data rate on network links, and latency of VCSs are considered.

Cappanera et al. [72] consider the placement and routing sub-problems of the SPRING problem in a distributed cloud environment with a focus on business-related objectives. They formulate the problem with the objective of maximizing the request acceptance rate (desired by service providers), while considering the requirements of the service users, e.g., service priorities and quality-of-service objectives, as well the information disclosure limitations of network operators. The requirements of different stakeholders in a service management and orchestration scenario are (directly and indirectly) reflected in the objectives described for the optimization problems in this dissertation.

2.2.3 Network Function Virtualization Context

The SPRING problem is also relevant in the field of NFV. In the NFV context, the *forwarding graphs* of network services composed of multiple VNFs are mapped into the network. Herrera and Botero [73] have published an analysis of existing solutions for placing network services as part of a survey on resource allocation in NFV. The SPRING approaches cover the service chain composition and embedding stages that are identified for NFV resource allocation in this survey. The dynamic scaling, placement, and routing approaches presented in this dissertation can address the "Chain Re-Composition and Resilience to Failure" challenge described in the survey. For example, switching to another deployment version of a running heterogeneous pliable VCS can prevent insufficient service performance and failures.

Kuo et al. [74] consider the joint placement and routing problem, focusing on maximizing the number of admitted network service embedding requests. Luizelli et al. [75] propose efficient and cost-effective placement and chaining approaches. Ahvar et al. [76] propose a solution to the placement and routing problem, with the assumption that the VNFs can be reused among different flows. Their objective is to find the optimal number of VNFs for all requests and to minimize the costs. Other approaches that consider re-using service components are proposed by Bari et al. [77] and Savi et al.[78]. The B-SPRING approaches also allow re-using service components among different VCSs, if allowed and requested by the provider of the VCS. In the B-SPRING approaches, the components can also be jointly used by the upstream and downstream flows of bi-directional VCSs. None of the mentioned solutions consider bi-directional VCSs.

Kebbache et al. [79] aim at solving the placement and routing sub-problems of the SPRING problem in an efficient way that can scale with the size of the underlying infrastructure and the embedded services. They measure the efficiency of their algorithms with respect to run time, request acceptance rate, and costs. Another attempt to solve this problem in an efficient and scalable way has been made by Luizelli et al. [80], focusing on minimizing resource allocation. Nguyen et al. [81] formulate the joint placement and routing problem as a Mixed-Integer Linear Program (MILP) and propose heuristics for solving it. Gao et al. [46] also solve the VNF placement and routing optimization problem. As in the M-SPRING model, they consider the forwarding latency of the VNFs (Time in System (TiS) in my model). None of these proposed solutions consider the scaling problem. Compared to all these approaches, I consider more comprehensive optimization objectives in the SPRING problem formulations, e.g., trying to minimize the delay for embedded VCSs, the number of added or removed instances of the service components, node and link resource consumption, as well as the over-subscription of resources.

Beck and Botero [42] provide a heuristic for embedding services with arbitrarily ordered components, based on the optimization model assumptions of the A-SPRING problem. They solve the problems of finding the best composition for VNFs in VCS and finding the best placement for the service in one step. In contrast to the two-step approach I describe in this dissertation, this combined approach is limited to a specific optimization objective. Similar to the two-step approach, their solution cannot guarantee the optimality of the selected composition, either.

Sahhaf et al.[82] discuss the placement and routing sub-problems, taking internal decomposition possibilities for VNFs into account. In their model, some VNFs can be replaced with a set of multiple inter-connected VNFs that have the same external interfaces as the original VNF. This is related to the M-SPRING approaches, where different service components can be deployed in different *ver-*

sions, possibly consisting of different number of elements. Their approach does not include the scaling step of the M-SPRING problem. Moreover, unlike the M-SPRING approach, they solve the problem in two phases, a service decomposition selection step and a mapping step for the selected decomposition. Similar to the M-SPRING approaches, Mehta and Elmroth [83] study the trade-off between cost and performance in mobile edge clouds within heterogeneous 5G networks. None of the mentioned solutions consider the ability of instantiating different versions of the same service component.

Many of the existing approaches allow fixing the location of start and end points of services. In addition to this feature, in the B-SPRING model, instances of any *intermediate* service component can be fixed as well, e.g., to model legacy Physical Network Functions (PNFs). The placement model from Moens and DeTurck [48] supports hybrid networks, which partly consist of dedicated physical hardware. This model is related to the B-SPRING model that supports combinations of virtualized and legacy components in VCSs but it does not offer the flexibility of the SPRING approaches, as it is a placement-only solution.

Most of the existing models consider scaling [84, 85] and placement [74, 86, 42] separately and only produce initial embeddings without taking previous ones into account when load changes. Only few models consider scaling or modification of existing embeddings. Ghaznavi et al. [87] focus on optimizing existing embeddings while minimizing the overhead of modifications, which is also considered in the SPRING approaches. Mijumbi et al. [88] also consider online modifications of existing embeddings. However, they assume that VNF instances are already placed in the network and the requests for these VNFs are then mapped to the instances.

While there are similar aspects among the mentioned studies and the SPRING approaches, no existing approach combines the following steps: (i) scaling, i.e., deciding the right number of instances and allocating the right amount of resources to each of them, (ii) placement, i.e., deciding the location of each instance, (iii) routing, i.e., deciding the optimal paths among instances, which is the core of the SPRING problem. Moreover, in contrast to many of the existing solutions, the SPRING approaches can be used for finding the initial embedding of a template, as well as for adjusting existing embeddings.

3 Services with Arbitrarily Ordered Components

In Section 3.1 I describe in detail the notion of pliable Virtualized Composed Services (VCSs) that include arbitrarily ordered components and the challenges of modeling these VCSs as well as mapping them to the underlying network. In Section 3.2, I describe the groundwork that builds the basis of my contributions in this part of the dissertation. This chapter partially includes figures and verbatim copies of the text from my papers [22, 23, 24, 29].

3.1 Challenges

For delivering a service, different virtual or physical Network Functions (NFs) and application-specific components need to be deployed in the network and the corresponding flows need to be routed through them. As described in Section 1.1.1, there might be different possibilities for the flows to traverse a set of service components, which result in different service graphs for a VCS, all of them delivering the same functionality.

I model the influence of each service component on the structure of the VCS by categorizing the components based on their expected impact on the network flows into two different types as follows:

1. *Non-splitting components* that forward incoming flows without splitting them, with a data rate that can be equal to, more than, or less than their data

rate when entering the component (e.g., as a result of changing the data encoding by the component),

2. *Splitting components* that distribute incoming flows over different branches with equal or different data rates (e.g., for load balancing or as a result of traffic classification).

In both categories, the expected Ratio of Outgoing to Incoming Data Rate (ROID) is assigned to each branch leaving a service component. Based on this categorization, changing the order of traversing two arbitrarily ordered components in a service can result in the following cases:

- If both components are non-splitting and have a similar ROID, then chaining them together in either of the ordering options results in the same data rate on the path between them and in the same number of instances for the components and, therefore, in the same resource demands in total.

- If both components are non-splitting but have different expected ROIDs, then chaining them in different orders results in different data rates on the path between them and, therefore, different total resource demands. The number of instances required for each component is the same in both ordering options.

- If at least one of the components is a splitting component, then based on the ROID of each component, chaining them in different orders can result in different data rates on the paths between them, different number of required instances, different number of paths that should be created to connect the instances, and different total resource demands. For example, if the splitting component is a load balancer and the non-splitting component is a firewall, placing the firewall before the load balancer means one instance of each component is needed. But placing the load balancer first could mean having one firewall instance on each of the outgoing branches from the load balancer.

I consider these options as a new degree of freedom in service composition and propose a way to profit from this; namely, defining pliable VCSs with arbitrarily ordered components, such that the order of traversing a subset of service components in a VCS can be determined and modified dynamically, in case the intended functionality of the service is not impaired. I have addressed two challenges in this regard.

The *first challenge* is to formalize a request for composing a pliable VCS out of different components while considering the possible dependencies among them. Specifying the flexibility in the service structure cannot be done using traditional graph representations in an efficient way. As a part of my master's thesis [38], also published as a conference paper [22], I have proposed a context-free grammar for specifying the structure of pliable VCSs. In subsequent publications [23, 24, 29], after my master's thesis, I have enhanced and extended the grammar and designed a YANG [89] model for defining, modifying, and reusing complex and flexible chaining structures for VCSs. I give a brief overview of the model and its use cases in Section 3.2.1.

Upon receiving such a request, the network operator has the freedom to compose the VCS in the best possible way to fit the requirements of the VCS and the network. It is also used by the Management and Orchestration (MANO) system of the network operator to calculate the best embedding for the VCS, according to the current state of network and the requirements of existing VCSs.

The *second challenge* is, hence, to find the best location for the service components (i.e., *placement*) and creating the paths among them (i.e., *routing*), considering the requirements of individual requests as well as the overall requirements of all network applications and the running VCSs.

If the structure of the VCS is fully specified as a set of totally ordered service components, the only decision that needs to be taken in the placement step is mapping the service components to network nodes and creating required paths among them. If necessary, previous mappings can also be modified to accommodate new services.

However, when VCSs have a pliable structure, the placement and routing decisions depends on which service graph is used for each VCS. n different service components that are specified with an arbitrary order can be chained together in $n!$ ways. Therefore, the graph generation, placement, and routing calculations for a set of pliable service deployment requests can quickly result in combinatorial explosion.

To improve the practical applicability of defining, managing, and orchestrating several pliable VCSs in an environment, the solution space needs to be limited. This, however, is not an straightforward task, because we have to deal with different metrics for two different decisions, namely, composing the service components together in a specific order and calculating the embedding into the network for a set of VCSs.

Measurable metrics for comparing different candidate service graphs for different VCSs *before* calculating the embedding are limited to the information available in the service descriptors, e.g., the resource demands for individual service components. These metrics are not the same as the **metrics of interest** that can only be measured *after* the services are mapped to the network, e.g., the end-to-end latency of an entire VCS or resource utilization in a network that hosts several VCSs.

In Section 4.2, I describe a *selection heuristic* that uses the limited information available before the embedding for selecting a representative subset of possible options that can potentially result in a close-to-optimal state for the network after the actual embedding. The output of the selection heuristic is a combination of different service graphs, given as input to the placement, scaling, and routing step, where the service graphs need to be mapped to the network.

For evaluating the selection heuristic in Section 4.3, I have used a joint placement, scaling, and routing optimization problem that I had developed as a part of my master's thesis [22]. For completeness and to provide the required background, I include an overview of a Mixed-Integer Quadratically Constrained Program (MIQCP) formulation of this problem in Section 3.2.

In Section 4.4, I describe a heuristic that finds close-to-optimal solutions for the joint **Scaling, Placement, and RoutING (SPRING)** problem with respect to the relevant metrics in large-scale distributed networks, e.g., the remaining link capacity in the network after the mapping. I consider variants of the SPRING problem in other chapters, with different capabilities. To differentiate among these problems, I refer to the SPRING problem in the context of pliable VCSs with arbitrarily ordered components as the **A-SPRING Problem**. I show the evaluation results of the heuristic in Section 4.5.

3.2 Service Specification, Graph Generation, and Embedding Models

In this section, I briefly describe the model and assumptions regarding the underlying network and the deployment requests for pliable VCSs with arbitrarily ordered components. Afterwards, I explain how the deployment requests are processed and briefly describe the joint placement, scaling, and routing optimization problem for these services. These descriptions are based on my previous research [22] and serve as a background for my main contributions regarding pliable VCSs with arbitrarily ordered components, described in Chapter 4.

3.2.1 Service Deployment Requests

I model the *substrate network*, where VCSs are submitted and deployed, as a connected, directed graph $G = (V, E)$. I assume some of the network nodes are *switch nodes*, with typical routing and switching capabilities. Every switch node has a special-purpose compute capacity $c_s(v) \geq 0$, $\forall v \in V$ that can be used for running service components, e.g., using Field-Programmable Gate Arrays (FPGAs) in the switch. The remaining nodes are distributed sites with general-purpose computational capacity $c_d(v) \geq 0$, $\forall v \in V$. I consider each of these sites as a large computational unit, called a *data center node*, without looking into their internal topology. In this model, I assume that for switch nodes, c_d is zero and for data center nodes, c_s is zero. I consider the more general case, where every network node may offer special- and general-purpose resources in Chapter 8.

The network links are directed edges in the graph, with data rate $d(v, v')$ and latency $l(v, v')$ for every edge $(v, v') \in E$.

I assume the following information about the offered network functions and application components (i.e., the service components) in the network is available and maintained by the network operator as a catalog:

- Set F of available service components. The service components might be offered by the network operator or submitted to the operator's network for deployment by a specific service provider.
- Computational resource demands $p(f)$, $\forall f \in F$, of an instance of the service component f per each request, specified as $p_s(f)$ for special-purpose compute

resources and as $p_d(f)$ for general-purpose compute resources. $p(f)$ can easily be extended to describe demands for multiple different resource types like memory, storage, etc. For simplicity, I only consider the general- and special-purpose computational resources. Some service components can be placed either on a switch or on a data center node, e.g., a load balancer ($p_d > 0$ and $p_s > 0$), and some can be placed only on a data center node, e.g., a Virtual Machine (VM) hosting a video optimization functionality ($p_d > 0$ and $p_s = 0$).

- The maximum number of instances of each service component that can be deployed $n_{\text{inst}}(f)$, e.g., determined by the number of licenses that the network operator or the service provider owns for the corresponding software.
- The maximum number of VCSs that can use an instance of a component simultaneously $n_{\text{req}}(f)$. For example, an anti-virus function can be configured once and used for every VCS that needs this functionality (i.e., the number of requests it can handle is only limited by hardware specifications) but a firewall might need specific configurations for each VCS and one instance of it cannot be shared and reused among different VCSs (i.e., $n_{\text{req}} = 1$).

The network operator receives *deployment requests* for different partially ordered sets of service components. These requests specify which of the service components in the network (set F) should be applied in which order to the service flows between fixed start and end points. A deployment request contains the following information:

- Set U of individual requests for using instances of available service components.
- *Composition request*, denoted as c, for specifying the desired order of functions. Later in this section, I describe in detail how the composition requests can be expressed.
- For each branch leaving a requested service component, the percentage of incoming data rate it produces given as an ordered set $r(u)$, $\forall u \in U$ for each service component. For example, for a Deep Packet Inspector (DPI) that is expected to send 20 % of the incoming packets towards a video optimizer and 80 % towards a firewall, this set is given as $\{0.2, 0.8\}$.
- Set A of fixed start or end points for the service flows, e.g., an application component deployed in a data center node or a router that connects the operator's network to external networks. These points are paired together to represent directed flows, as I describe in the next points.
- Locations of start or end points of flows in the network $\text{loc}(a) \in V$, $\forall a \in A$.
- Set $A_{\text{pairs}} \subseteq A \times A$ of *pairs* of start and end points belonging to different VCSs. Each tuple $(a_1, a_2) \in A_{\text{pairs}}$ represents a directed flow from the starting point a_1 of a service to its end point a_2.

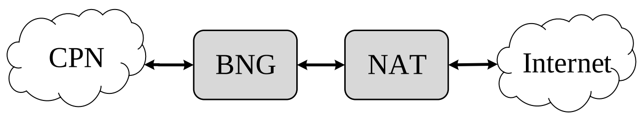


Figure 3.1: A simple sequence of service components (adapted from Ref. [90])

- Initial data rate d_{in} entering each VCS.
- Maximum tolerable latency between the start and end points $l_{\text{req}}(a, a')$, $\forall (a, a') \in A_{\text{pairs}}$.

Within each instance of a service component, I assume an M/D/1 queuing model (in Kendall notation). The maximum tolerable latency between every pair of start and end points of a VCS excludes the actual service time (request processing time) within the service components, as those are deterministic values (according to the M/D/1 model) that can easily be included in the final end-to-end delay. To simplify the model, I do not consider the waiting times for service flows within service components in the A-SPRING approaches.

For formalizing the *composition requests*, I have defined a context-free language [23]. The corresponding context-free grammar in Extended Backus-Naur Form (EBNF) is as follows.

⟨start⟩ ::= **service** **{**⟨composition⟩(**,**⟨composition⟩)***}**
⟨composition⟩ ::= ⟨components⟩ | ⟨bestbind⟩ | ⟨allbinds⟩ | ⟨splt⟩ | ⟨comp⟩
⟨bestbind⟩ ::= **best-binding** **{**⟨components⟩**}**
⟨allbinds⟩ ::= **all-bindings** **{**⟨components⟩**}**
⟨splt⟩ ::= **split** **{**⟨comp⟩(**,**⟨bestbind⟩)?(**;**⟨branch⟩)$^+$**}**
⟨branch⟩ ::= ⟨composition⟩(**,**⟨composition⟩)*(**.**⟨num⟩)? | **pass**
⟨components⟩ ::= ⟨comp⟩(**,**⟨components⟩)*
⟨comp⟩ ::= ⟨nf⟩ | ⟨endpoint⟩
⟨num⟩ ::= ⟨nonzero⟩ ⟨digit⟩*
⟨nonzero⟩ ::= **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
⟨digit⟩ ::= **0** | ⟨nonzero⟩
⟨nf⟩ ::= $\boldsymbol{f_1} \,|\, \boldsymbol{f_2} \,|\, \cdots \,|\, \boldsymbol{f_{|F|}}$
⟨endpoint⟩ ::= $\boldsymbol{p_1} \,|\, \boldsymbol{p_2} \,|\, \cdots \,|\, \boldsymbol{p_{|P|}}$

The terminals of the grammar are given in bold font. ⟨nf⟩ and ⟨endpoint⟩ correspond to the set of available NFs (representing service components like virtual network functions, cloud-based micro-services, etc.) and the set of locations in the network where the service flows start or end.

This grammar can be used to specify a set of *totally* ordered service components to be chained together in the given order (simple sequence) and a set of *partially* ordered service components to be chained together in the most efficient order according to the optimization objectives in the network.

A simple example of a totally ordered VCS in fixed broadband networks consists of a Broadband Network Gateway (BNG) and a Network Address Translator (NAT) as endpoints of service flows between the Customer Premises Network

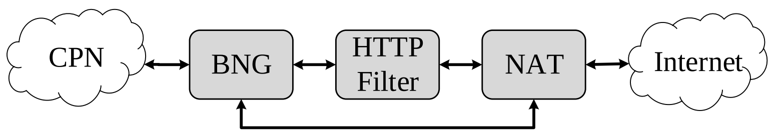


Figure 3.2: A VCS with a branched structure (adapted from Ref. [90])

(CPN) and the public Internet. Assuming that all network flows need to traverse these two functions, Figure 3.1 shows the structure of this VCS. Using a simple sequence from my model, the abstract description of the structure of this VCS is the following:

service{BNG, NAT**}**

In case the order of traversing the service components does not affect the functionality of the service, the structure can be described using a **best-binding** composition instead of a simple sequence of service components. For example:

service{best-binding{BNG, NAT**}}**

A complex branched structure for splitting the flows over different branches can also be expressed. For this, the **split** composition can be used, which consists of:

- a service component that can classify and split the flows over different branches, specified as the first ⟨comp⟩ in the **split** composition,
- an optional best-binding composition to be traversed before the flows reach the branches,
- branches that can consist of a single service component or endpoint, a composition of multiple service components, or an empty branch (**pass**) that can be used for skipping a part of the service structure. In case the branches are identical, they need to be specified only once together with the number of required replications (e.g., for load-balancing).

In the previous example, we can further assume that HTTP traffic is detected and sent through an HTTP filter function and non-HTTP traffic is routed directly between the BNG and the NAT. The corresponding graph for this VCS is shown in Figure 3.2. The structure of this VCS can be expressed as follows using the **split** composition type comprising a branch type **pass** to skip the HTTP filter function for some flows:

service{split{BNG; HTTP-Filter; **pass},** NAT**}**

The first component in the **split** composition (BNG) is the splitter function. The HTTP Filter and the **pass** keyword each correspond to one outgoing branch.

As a more complex example, we can consider a scenario in a mobile broadband network, shown in Figure 3.3. This is a VCS between the Internet and a Packet Data Network Gateway (PGW) where the user equipments are connected via the

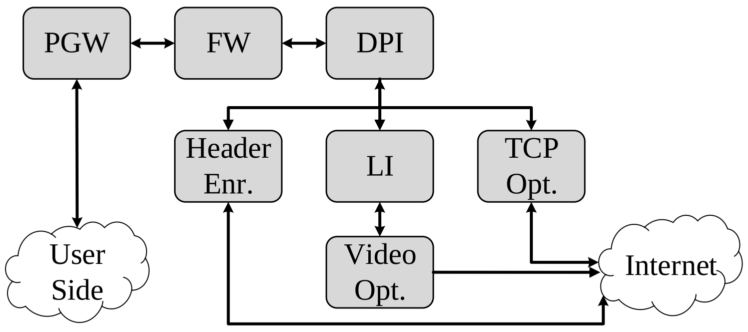


Figure 3.3: A complex VCS with a branched structure (adapted from Ref. [91])

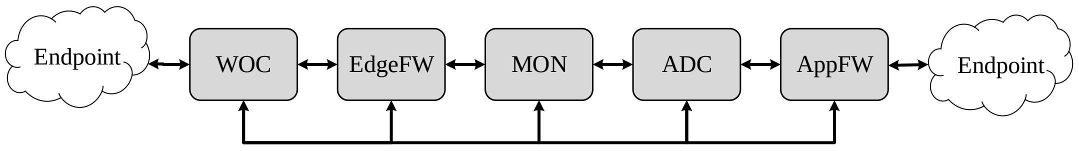


Figure 3.4: A VCS consisting of a full mesh of service components (adapted from Ref. [92])

access network. Firewall (FW) and DPI are applied to all flows and later on the flows are divided over three branches. TCP flows need to traverse a TCP optimizer function, flows belonging to a certain video streaming VCS go through a Lawful Interception (LI) and a video optimizer function, and other flows need to go through a header enrichment function. This structure can be expressed as follows, using a **split** composition:

service{
  PGW, FW,
  split{DPI; Header-Enr; LI, Video-Opt; TCP-Opt**}**
}

A set of service components can also be specified to be chained together in a way that all possible permutations of them are traversable (**all-bindings**), i.e., a full mesh of paths has to be built among the service components.

For this, we can consider an example scenario from a data-center network, shown in Figure 3.4. Different flows need to traverse different subsets and different permutations of the set of service components, including a Web Optimization Controller (WOC), a firewall responsible for external threats (EdgeFW), a network and application monitoring function (MON), an Application Delivery Controller (ADC), and an application-specific firewall (AppFW). This complex structure can be compactly described using an **all-bindings** composition:

service{all-bindings{WOC, EdgeFW, MON, ADC, AppFW**}}**

Existence of partially ordered sets of service components in a VCS structure turns the deployment request for the VCS into a flexible request that can be translated into the best possible service graph depending on requirements of the VCS and available resources in the network. Every such deployment request should be processed into a connected, directed graph, i.e., a service graph with a fixed structure, before it can be deployed.

The required service components as well as the start and end points of the service flows are the nodes of this graph ($U \cup A$). The start and end points are mapped to fixed locations in the substrate network. The location of the service components has be to determined. Each one of the directed links in the set of edges in the service graph (U_{pairs}) represents the order of traversing the functions. Every link in the graph has to be mapped to a path (consisting of at least one edge or an internal connection in a network node) in the substrate network graph.

In the rest of this section, I describe a two-step process for deploying a set of VCSs based on the corresponding composition requests: (i) processing the set of requests and building service graphs (Section 3.2.2), (ii) finding the optimal embedding for the service graphs into the network based on optimization goals (Section 3.2.3).

3.2.2 Generating Candidate Service Graphs

In this section, I assume a network operator needs to find the best embedding for several VCSs based on multiple deployment requests. The first step is to generate the possible service graphs for every deployment request.

In this step, for each deployment request, the composition request (c) needs to be parsed and processed. The parts of the composition request that consist of a single service component and the start and end point of flows can be stored as a node of the service graph. For the parts where a number of service components can be ordered arbitrarily, every possible permutation of the set of service components should be computed and considered as a candidate for being included in the final service graph. Moreover, for every match of a split composition (described in Section 3.2.1) with identical branches, the components on the branches should be replicated for the requested number of times and stored as a part of the graph.

Using the specified orders and depending on the component types that build the composition request, at the end of the parsing process, different components are stored as parts of the service graph with *total orders* among them.

Considering the different permutations for different arbitrarily ordered components, at least one service graph is built out of each composition request. Using the rest of the information in the deployment request and the information available about the service components in the network (as described in Section 3.2.1), the service graphs are annotated with the compute resource demands and the data rate and latency requirements.

Each of the graphs that can be created from a request can have different characteristics when embedded, e.g., in terms of the average data rate among service components and the number of instances required for the service components.

Revisiting a previous example, we can see that the Load Balancer (LB) shown in Figure 1.3 (page 8) splits the incoming flows into three different branches to balance the traffic over three instances of component C. I assume, for this example, the ROID in all service components is 1, except the LB that forwards 1/3 of its incoming data rate over each outgoing branch. Placing this load balancer earlier in the directed graph, as in Figure 1.3a, reduces the data rate of the links on each branch after it. But it also means that up to three instances of all subsequent service components will be required on the paths towards C. Each of these instances has can be deployed with less compute resources than the instances in Figure 1.3b, which needs to handle higher data rates.

For every such VCS that includes $n \in \mathbb{N}_{\geq 0}$ service components with an arbitrary order among them, $n!$ permutations need to be computed as a candidate. For a VCS that contains subsets of arbitrarily ordered service components, the number of different ways for composing the VCS is the product of the number of permutations for each of the included subsets. For example, if a composition request contains one **best-binding** composition with 3 different components and one **split** composition in which 4 components can be placed with an arbitrary order, a total of $3! \cdot 4! = 144$ combinations of these components is possible. That means, for finding the best service graph and deployment of this pliable VCS, the embedding would have to be calculated 144 times, if we needed to compare all possible service graphs so that the option that fits the requirements could be chosen.

As the number of combinations increases very quickly in the number of deployment requests and the number of service components in each request, trying every possible combination becomes impractical. I address this problem in Section 4.2.

3.2.3 Service Embedding Optimization Problem

Among the candidate service graphs that have been generated for a pliable VCS, the one that is selected for deployment needs to be embedded into the substrate network. The input to the service embedding problem is the capacities of the network nodes and links, the resource demands of the required service components, and the service graphs of the VCSs, as defined in Section 3.2.1. In this section, I describe an optimization approach for service embedding, in which it is decided:

- which service components should be instantiated on which network nodes (*placement*),
- which request for using a service component should be mapped to which instance of the corresponding component
- and how the traffic should be routed among the service components to deliver each requested service (*routing*).

As some components can be shared among different VCSs, there might be fewer instances of the shared components needed than initially described in each service deployment request. For example, if three services request to use one instance

Table 3.1: A-SPRING Parameters

Domain	Parameter	Description
$\forall v \in V$	$c_d(v)$	General-purpose compute capacity in v
	$c_s(v)$	Special-purpose compute capacity in v
$\forall (v, v') \in E$	$d(v, v')$	Link capacity on (v, v')
	$l(v, v')$	Latency of (v, v')
$\forall f \in F$	$n_{\text{inst}}(f)$	Number of allowed instances for f
	$n_{\text{req}}(f)$	Number of requests f can handle
	$p_d(f)$	General-purpose resource demand of f
	$p_s(f)$	Special-purpose resource demand of f
$\forall u \in U$	$t(u)$	Requested service component
$\forall (u, u') \in U_{\text{pairs}}$	$d_{\text{req}}(u, u')$	Expected data rate of (u, u')
$\forall a \in A$	$\text{loc}(a)$	Network node where a is placed
$\forall (a, a') \in A_{\text{pairs}}$	$\text{paths}(a, a')$	All possible paths between a and a'
	$l_{\text{req}}(a, a')$	Maximum latency between a and a'

of a DPI Virtual Network Function (VNF), the total number of requested DPI instances is three; however, I have formulated the optimization problem in a way that it can map the requests to less than three instances of DPI, e.g., if the node capacities allow forwarding more traffic to an instance. In this way, the mapping process also offers *scaling* capabilities, in addition to placement and routing.

This service *embedding* process solves the **A-SPRING problem** (defined in Section 3.1). I have formulated this optimization problem in a previous work [22] as a MIQCP. Table 3.1 shows an overview of the input parameters to the MIQCP.

The decision variables are described in Table 3.2. "remdr" and "lat" are continuous variables and all other ones are binary indicator variables. I show an overview of the constraints of the optimization problem in Section 3.2.3.1 and the objective functions in Section 3.2.3.2. Further details about the optimization problem can be found in the corresponding publication [22].

3.2.3.1 Constraints

I show the constraints of the optimization problem in three parts:

1. Placing the service components in the network nodes and mapping the requests for using instances of the required components to these nodes;

2. creating paths between the service components;

Table 3.2: A-SPRING Decision variables

Domain	Variable	Description
$\forall u \in U$, $\forall v \in V$	$m_{u,v}$	1 iff u mapped to v, otherwise 0
	$\text{ms}_{u,v}$	1 iff u mapped to a switch function on v, otherwise 0
	$\text{md}_{u,v}$	1 iff u mapped to a data center function on v, otherwise 0
$\forall f \in F$, $\forall v \in V$	$i_{f,v}$	1 iff an instance of f mapped to v, otherwise 0
$\forall (v, v') \in E$, $\forall x, y \in V$, $\forall (u, u') \in U_{\text{pairs}}$	$e_{v,v',x,y,u,u'}$	1 iff (v, v') belongs to the path between x and y, where u and u' are mapped to, otherwise 0
$\forall v \in V$	used_v	1 iff at least one service component usage request is mapped to v, otherwise 0
$\forall (v, v') \in E$	$\text{remdr}_{v,v'}$	Remaining data rate on (v, v')
$\forall (u, u') \in U_{\text{pairs}}$	$\text{lat}_{u,u'}$	Latency of the path between u and u'

3. collecting metric values.

Placement and path creation constraints are clearly separated, facilitating the extension of the model in either part without influencing the other part. Necessary ties between these parts are also carefully defined, for example, using the decision variable e, to build a consistent and uniform model for placing service components and chaining them together optimally.

Service Components Placement Constraints

$$\forall u \in U: \quad \sum_{v \in V} m_{u,v} = 1 \tag{3.1}$$

$$\forall a \in A: \quad m_{a,\text{loc}(a)} = 1 \tag{3.2}$$

$$\forall f \in F, \forall v \in V: \quad \sum_{u \in U t(u)=f} m_{u,v} \leq \mathcal{M} \cdot i_{f,v} \tag{3.3}$$

$$\forall f \in F, \forall v \in V: \quad i_{f,v} \leq \sum_{u \in U, t(u)=f} m_{u,v} \tag{3.4}$$

$$\forall u \in U, \forall v \in V: \quad \text{ms}_{u,v} + \text{md}_{u,v} = 1 \tag{3.5}$$

$$\forall u \in U, \forall v \in V, p_s(t(u)) = 0, p_d(t(u)) \neq 0 : \quad \mathrm{ms}_{u,v} = 0 \tag{3.6}$$
$$\forall u \in U, \forall v \in V, p_s(t(u)) = 0, p_d(t(u)) \neq 0 : \quad \mathrm{md}_{u,v} = 1 \tag{3.7}$$
$$\tag{3.8}$$

$$\forall v \in V : \quad \sum_{u \in U} m_{u,v} \cdot \mathrm{md}_{u,v} \cdot p_d(t(u)) \leq c_d(v) \tag{3.9}$$
$$\forall v \in V : \quad \sum_{u \in U} m_{u,v} \cdot \mathrm{ms}_{u,v} \cdot p_s(t(u)) \leq c_s(v) \tag{3.10}$$
$$\forall f \in F : \quad \sum_{v \in V} i_{f,v} \leq n_{\mathrm{inst}}(f) \tag{3.11}$$
$$\forall v \in V, \forall f \in F : \quad \sum_{u \in U, t(u)=f} m_{u,v} \leq n_{\mathrm{req}}(f) \tag{3.12}$$

Start and end points of the flows are fixed in the network, so $\forall a \in A$, $m_{a,\mathrm{loc}(a)}$ are not decision variables but pre-set constants.

$\mathcal{M} \in \mathbb{N}$ is a number larger than the sum on the left side of the inequality in Constraint 3.3 (a so-called “big M” constraint).

Path Creation Constraints

$$\forall (v,v') \in E, \forall x,y \in V, \forall (u,u') \in U_{\mathrm{pairs}} : \quad e_{v,v',x,y,u,u'} \leq m_{u,x} \cdot m_{u',y} \tag{3.13}$$

$$\forall (u,u') \in U_{\mathrm{pairs}} : \quad \sum_{(x,v) \in E, y \in V} e_{x,v,x,y,u,u'} \cdot m_{u,x} \cdot m_{u',y} = 1 \tag{3.14}$$
$$\forall (u,u') \in U_{\mathrm{pairs}} : \quad \sum_{(x,v) \in E, y \in V} e_{x,v,x,y,u,u'} \cdot (1 - m_{u,x} \cdot m_{u',y}) = 0 \tag{3.15}$$
$$\forall (u,u') \in U_{\mathrm{pairs}} : \quad \sum_{(v,y) \in E, x \in V} e_{v,y,x,y,u,u'} \cdot m_{u,x} \cdot m_{u',y} = 1 \tag{3.16}$$
$$\forall (u,u') \in U_{\mathrm{pairs}} : \quad \sum_{(v,y) \in E, x \in V} e_{v,y,x,y,u,u'} \cdot (1 - m_{u,x} \cdot m_{u',y}) = 0 \tag{3.17}$$

$$\forall (u,u') \in U_{\mathrm{pairs}}, \forall w,x,y \in V : \sum_{\substack{v \in V, v \neq y,\\ (v,w) \in E}} e_{v,w,x,y,u,u'} = \sum_{\substack{v' \in V, w \neq x,\\ (w,v') \in E}} e_{w,v',x,y,u,u'} \tag{3.18}$$

$$\forall (u,u') \in U_{\mathrm{pairs}}, \forall v,x,y \in V, x \neq y : \ e_{v,v,x,y,u,u'} = 0 \tag{3.19}$$

$$\forall (u,u') \in U_{\mathrm{pairs}}, \forall x,y \in V, \forall (v,v'),(v',v) \in E, v \neq v' : \quad e_{v,v',x,y,u,u'} + e_{v',v,x,y,u,u'} \leq 1 \tag{3.20}$$

$$\forall(v,v') \in E: \quad \sum_{(u,u') \in U_{\text{pairs}}, \forall x,y \in V} e_{v,v',x,y,u,u'} \cdot d_{\text{req}}(u,u') \leq d(v,v') \tag{3.21}$$

$$\forall(a,a') \in A_{\text{pairs}}: \quad \sum_{\substack{(v,v') \in E, x,y \in V, \\ (u,u') \in \text{paths}(a,a')}} e_{v,v',x,y,u,u'} \cdot l(v,v') \leq l_{\text{req}}(a,a') \tag{3.22}$$

$$\forall(v,v') \in E, \forall x,y \in V, \forall(u,u') \in U_{\text{pairs}}: \quad e_{v,v',x,y,u,u'} \leq m_{u,x} \cdot m_{u',y} \tag{3.23}$$

Metrics Calculation Constraints

$$\forall v \in V: \sum_{f \in F} i_{f,v} \leq \mathcal{M}' \cdot \text{used}_v \tag{3.24}$$

$$\forall v \in V: \text{used}_v \leq \sum_{f \in F} i_{f,v} \tag{3.25}$$

$$\forall(v,v') \in E: \quad \text{remdr}_{v,v'} = d(v,v') - \sum_{\substack{(u,u') \in U_{\text{pairs}}, \\ \forall x,y \in V}} e_{v,v',x,y,u,u'} \cdot d_{\text{req}}(u,u') \tag{3.26}$$

$$\forall(u,u') \in U_{\text{pairs}}: \text{lat}_{u,u'} = \sum_{x,y \in V, (v,v') \in E} e_{v,v',x,y,u,u'} \cdot l(v,v') \tag{3.27}$$

$\mathcal{M}' \in \mathbb{N}$ is a number larger than the sum on the left side of the inequality in Constr. 3.24.

3.2.3.2 Optimization Objective

Different objectives can be targeted for this optimization problem and each of them can result in a different mapping of the service graphs into the network graph. I define three objective functions and describe the behavior of the embedding process using each objective.

Maximizing the Remaining Data Rate on Network Links

$$\text{maximize} \sum_{(v,v') \in E, v \neq v'} \text{remdr}_{v,v'} \tag{3.28}$$

As highly utilized links can result in congestion in the network, solutions that leave more capacity on the links are desirable. This objective aims at leaving more data rate on the links. By maximizing the sum of remaining capacity over all links, it forces the algorithm to prefer self-loops (i.e., links between two service components that are placed on one network node), over than other links. There self-loops do not consume any inter-node networking capacity and are realized as intra-node connectivity. In this way, network traffic on the links is avoided by preferentially placing service components on the same node.

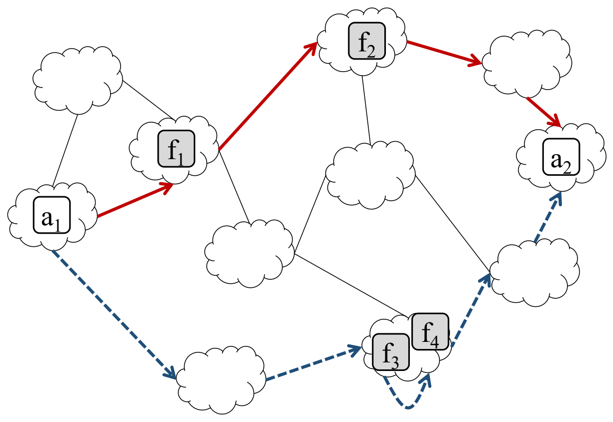


Figure 3.5: Two simple paths between start and end points of an example VCS

Minimizing the Number of Used Nodes in the Network

$$\text{minimize} \sum_{v \in V} \text{used}_v \tag{3.29}$$

This objective function can result in an energy-efficient solution by allowing more unused nodes to be switched off. It might, however, concentrate the service components on a small subset of nodes, causing congestion in the network.

Minimizing the Latency of the Created Paths

$$\text{minimize} \sum_{(a,a') \in l_{\text{req}}} \Bigg(\sum_{P \in \text{paths}(a,a')} \Bigg(\sum_{(u,u') \in P} \text{lat}_{u,u'} \Bigg) \Bigg) \tag{3.30}$$

In complex VCSs with branches in the structure, there are multiple *simple paths* between the start and end points. A simple path connects the starting point of a service to one end point of it via a sequence of network links and does not include any branches in its structure. For example, there are two simple paths between the start point (a_1) and end point (a_2) of the example VCS shown in Figure 3.5. The upper simple path consists of the links shown with red, solid lines and connects the start and end points via the service components f_1 and f_2. The lower simple path in this figure, consists of the links shown with blue, dotted lines and includes a self-loop on a node that represents the intra-node connectivity of the service components f_3 and f_4.

As each path consists of different sequences of network links, they can have different latencies. This objective function minimizes the mean latency of all simple paths created for all deployment requests.

4 Embedding Services with Arbitrarily Ordered Components

In Chapter 3, I have described how deployment requests for pliable Virtualized Composed Services (VCSs) that include arbitrarily ordered components are expressed and processed. I have also defined the A-SPRING problem for scaling, placing, and routing these services.

In Section 4.1, I give an overview of the problem of processing and fulfilling the deployment requests for a set of pliable VCSs that include arbitrarily ordered components. In Section 4.2, I describe a heuristic for generating and combining the service graphs of such VCSs and show the results of evaluating this approach in Section 4.3. In Section 4.4 I present a heuristic for solving the A-SPRING problem and evaluate it in Section 4.5. I conclude this chapter in Section 4.6. This chapter partially includes figures and verbatim copies of the text from my papers [22, 23, 24, 29].

4.1 Problem Formulation

In this chapter, I assume multiple VCS deployment requests are submitted by different service providers to a network operator and need to be processed together to find the optimal embedding, considering the requirements of the new and the existing VCSs. The information included in a deployment request, used for placement, scaling, and routing of the VCSs, follows the service descriptor definition from European Telecommunications Standards Institute (ETSI) for Network Function Virtualization (NFV) Management and Orchestration (MANO) [6], e.g, the way the components included in the VCS and their resource demands should be described. The structure of the VCSs and dependencies among the service components, however, go beyond the ETSI descriptors and are specified in a flexible way, as described in Section 3.2.1.

I assume the input data rate to a VCS is variable and can change during the lifetime of the VCS. I also assume the resource demands of each service component are defined as a function of the data rate it needs to handle, increasing linearly with the incoming data rate to that component.

In the rest of this chapter, I refer to the process of creating service graph(s) out of a service deployment request (defined in Section 3.2.2) as the *graph generation process*. The *combination process*, in turn, refers to combining different service graphs from different VCSs into a single input to the embedding process, which I describe in this section.

Figure 4.1 shows a possible set of actions required for finding the best embedding for three example VCSs. In this figure, VCSs X and Z each include two components that can be traversed in an arbitrary order. Therefore, in step ①, the graph generation process builds two different service graph candidates for each of them (denoted as X_1, X_2 and Z_1, Z_2, respectively). The deployment request for VCS Y specifies a total order, resulting in one single service graph.

To consider *all* possible service graphs in the service embedding process, the created service graphs go through step ②, where the combination process creates all possible combinations of different service graphs and passes them to the service embedding process. For the small example in Figure 4.1, the embedding process in step ③ needs to run four times, once for each possible combination of the service graphs. The embedding that provides the best values for the metrics of interest can be chosen in step ④ for actual deployment[1].

Performing these steps for a large set of pliable VCSs, can quickly result in a very large number of possible combinations of the service graphs. In this case, for solving the composition and embedding problem, one end of the spectrum is to explore the whole set of possible combinations of service graphs. In spite of the benefits resulting from defining pliable VCSs, running a (computationally expensive) service embedding process for every possible combination is not a practical solution.

Alternatively, the set of all steps shown in Figure 4.1 can be modeled as one large optimization problem that needs to be solved only once. Considering the complexity of the VCS embedding optimization problem for one single VCS, solving such a complicated optimization problem will not be possible in acceptable

[1]As the state of the underlying network and the deployed services change, the selected combinations might not be desirable anymore. In this case, another combination can be selected as a means of scaling and adapting the deployed VCSs

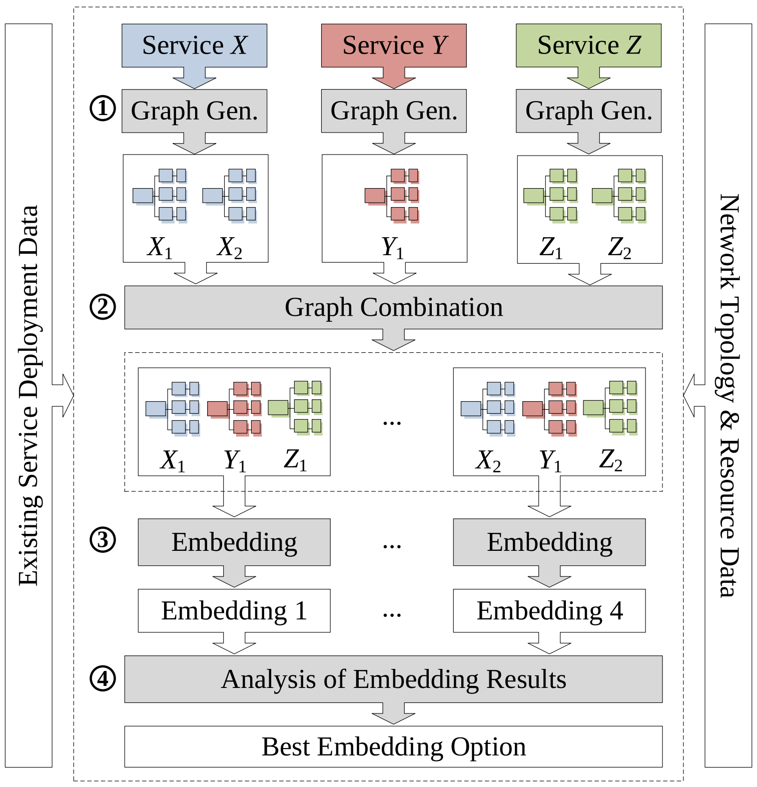


Figure 4.1: Possible steps for finding the best embedding option for a set of example VCSs

time-scale, either.

The other end of the spectrum is to define a "default" way of generating the service graphs, irrespective of different functionalities and requirements of VCSs. For example, a rule can be defined in step ① to sort the service components in ascending order of the Ratio of Outgoing to Incoming Data Rate (ROID) [22], if they are specified with an arbitrary order. In this way, we have exactly one service graph for each deployment request; in Figure 4.1, this corresponds to filtering out X_1 or X_2 (and Z_1 or Z_2). Using the example rule, the resulting graph has the smallest value for the average required link capacity. The embedding process needs to run only once for the single combination of all of these pre-filtered service graphs.

The drawback of choosing one single combination in this way is that it eliminates the variety of optimization options offered by different ways of generating service graphs for the pliable VCSs. For example, there might be candidate service graphs that optimize metrics like the number of required instances for service components, required compute resources for satisfying all deployment requests, number of utilized network nodes, etc. But they are simply discarded using such a *single-solution* heuristic. This solution would only be acceptable if the best possible service graph could be determined already in step ② in Figure 4.1. In practice, the information available about the possible combinations in this step does not directly correspond to the metrics of interest after embedding, which can only be measured and analyzed in step ④.

What we can ideally achieve using the information available before the embedding process is to relate them to the structure and requirements of VCSs by concentrating on multiple metrics. Using a *multi-solution* heuristic, we can select a representative subset of possible combinations for VCSs. Then, we can use the reduced set as the set of input options for the embedding process. This reduces the decision time, by reducing the number of times the embedding process needs to be performed, compared to exploring the whole set of options. Moreover, regardless of the optimization objectives used for calculating the embedding, there is a good chance of finding close-to-optimal solutions, as in the input set, we have representatives for all possible options.

I describe such a *selection heuristic* in Section 4.2; it provides a generic method for selecting a subset of possible combinations of the requests as input for the embedding process. The metrics used for filtering the combinations should be selected based on the objectives in each scenario. In Section 4.3, I have evaluated the heuristic using metrics and objectives that are targeted towards congestion control over nodes and links in the network, as one of possible optimization objectives. The solution can easily be generalized to other metrics and objectives.

Once the combination and selection processes are completed, the actual embedding can be calculated, e.g., using the MIQCP formulation described in Section 3.2.3 for optimally embedding the selected combinations of the service graphs, with the intended optimization objective.

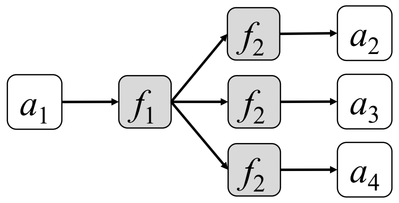
This MIQCP formulation consists of a rather complex model to ensure a realistic decision. Therefore, even if the output of the selection heuristic is one single combination, finding the optimal embedding on large-scale networks can take several hours for large request sets with complex structures. To address this issue, in addition to the selection heuristic and to further enhance the decision time, I present a service embedding heuristic in Section 4.4.

4.2 Service Graph Selection Heuristic

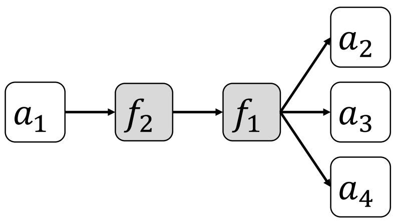
For this heuristic, first of all, every possible combination of different service graphs need to be calculated. Although the number of combinations can be large, computing them *without* performing the actual embedding is not an expensive task.

For choosing a representative subset of the combinations, meaningful metrics need to be extracted out of the information available about the combinations *before* the embedding. I have used the following metrics for evaluating and comparing different combinations of VCSs.

- Sum of data rates over all virtual links in all service graphs in each combination of service graphs
- Sum of resource demands (e.g., an abstract value representing the amount of required compute, memory, and storage resources) over all components in all service graphs in each combination

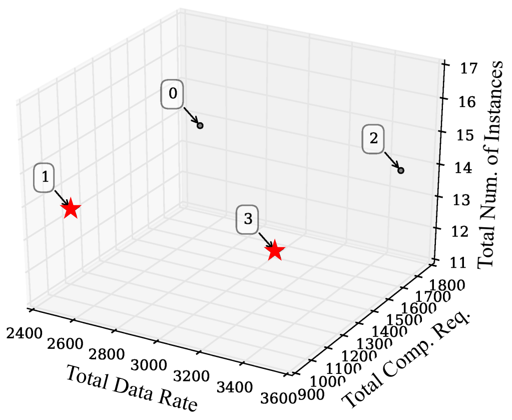


(a) Service graph candidate 1

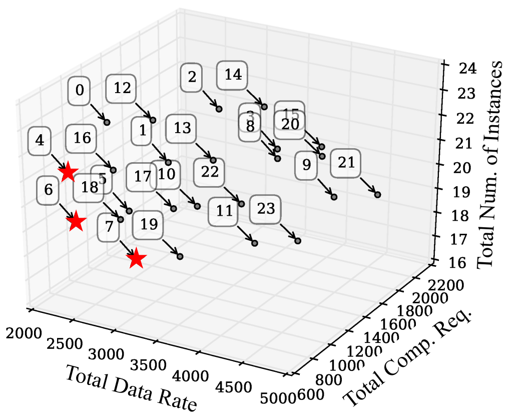


(b) Service graph candidate 2

Figure 4.2: Two different candidates for the service graph of an example pliable VCS, consisting of two components f_1 and f_2 that can be traversed in an arbitrary order between the service endpoints



(a) Combinations of S_0 and S_1



(b) Combinations of S_2, S_3, and S_4

Figure 4.3: Selected combinations for example VCSs

- Total number of required instances of all service components over all service graphs in each combination

This can, of course, be modified to any other metric aggregating the information available in service deployment requests that can represent the different options for the structure and requirements of the VCSs.

The combinations with the best possible trade-offs among the chosen metrics can then be identified by taking the Pareto-optimal combinations regarding these metrics. In a scenario where different metrics need to be optimized, Pareto-optimal solutions are the solutions that cannot improve one metric without worsening at least one other metric. The *selected combinations* by our multi-solution heuristic are the combinations in this Pareto set.

To demonstrate how the Pareto sets could look like, I have used two different sets of example service deployment requests consisting of five VCSs, S_0–S_4. Each VCS consists of two or three service components (Virtual Network Functions (VNFs), in this example) specified with an arbitrary order.

The service components need to be traversed between given start and end points. In each VCS, one component can split incoming flows over three different branches. I have set the input data rates of the VCSs and the ROIDs of service components randomly. Resource demands of each component increase linearly with the data rate that enters the component. I have also set the resource demand for a unit of data rate randomly, for each service component.

S_0 and S_1 have a similar structure, with different input data rates and different ROID for each included component. Their structure is shown in Figure 4.2. Each of them results in two different service graphs and, therefore, they can be combined into four different inputs for the embedding process (in step ② of Figure 4.1).

These combinations (numbered from 0 to 3) result in different resource demands and different values for the metrics of interest, as shown in Figure 4.3a. For simplicity and visibility, I assume the compute resource requirement of components in these plots represent the overall resource demands. The Pareto-optimal combinations (1 and 3) are marked with a star. Combination 1 consists of the service graph candidate 2 for S_0 and service graph candidate 1 for S_1, while combination 3 consists of the service graphs candidate 2 for both S_0 and S_1.

Similarly, as shown in Figure 4.3b, there are 24 possible combinations for service graphs that can be built for S_2–S_4, out of which 3 combinations belong to the Pareto set and are selected by the heuristic. The structure of $S_3$and S_4 also corresponds to the structure shown in Figure 4.2 but S_2 consists of 3 components (one splitting components and two non-splitting components) that can be traversed in an arbitrary order, resulting in 6 different chaining options.

Taking the selected combinations instead of the complete set of combinations creates inputs for the embedding process with the best possible trade-offs among the metrics available before the actual embedding. In Section 4.3, I evaluate the quality of the selected combinations by comparing the embedding results for these combinations to the set of optimal embedding solutions with respect to the objectives I have selected for controlling the congestion in network nodes and links.

4.3 Evaluation of Selection Heuristic

To evaluate how good the selected combinations can represent the set of all possible combinations, I have embedded different sets of example VCSs in a total of 450 runs. The graph generation and embedding process follows the steps shown in Figure 4.1. An implementation of this heuristic is available online [37].

Each embedding run consists of computing the optimal embedding for a set of deployment requests for pliable VCSs. During each run, I have embedded *all* possible combinations of the service graphs generated out of the deployment requests. This, of course, includes the combinations selected by the selection heuristic. In this way, the quality of the solutions achieved using the selection heuristic can be positioned among all possible solutions.

I have chosen the structure of the example VCSs based on an Internet Engineering Task Force (IETF) Service Function Chaining (SFC) draft on general use cases for SFC [93]. I have used 9 different sets, each including 1–3 pliable VCSs deployment requests. Each VCS includes 2–3 arbitrarily ordered components. In order to have interesting distinctions among different candidate service graphs of a VCS, each VCS includes at least one component that splits the flows over different branches (splitting component as defined in Chapter 3).

The sum of the input data rates for the requests in a set is the same for all embedding runs and is distributed uniformly, randomly over the requests in the set. The endpoints of the requested VCSs are pinned to randomly selected nodes in the network in each run. The underlying network has 12 nodes and 42 directed edges (including self-loops) and is based on the *abilene* network from SNDlib [94] problem instances. Among available network instances in this library, I have selected a small one to be able to apply the optimization approach in a reasonable time. Similarly, I have chosen the size and complexity of the deployment requests in a way that running the optimization approach for all possible combinations of the resulting service graphs is feasible in a reasonable time.

I have used a slightly modified version of the optimization problem described in Section 3.2.3 for embedding the request sets in the network.

Assuming the substrate network graph $G = (V, E)$, I have used the following additional constraints in the optimization problem:

$$\forall v \in V : \text{NodeUtil}_v \leq \text{MaxNodeUtil}$$

$$\forall (v, v') \in E, v \neq v' : \text{LinkUtil}_{(v,v')} \leq \text{MaxLinkUtil}$$

where NodeUtil_v and $\text{LinkUtil}_{(v,v')}$ are continuous variables with values between 0 and 1 that show the utilization of node v and link (v, v'), respectively. With similar definitions, MaxNodeUtil and MaxLinkUtil serve as upper bounds for node and link utilization, which I minimize using the objective function, as an attempt to minimize the node and link congestion in the network. I have used the equally-weighted sum of MaxNodeUtil and MaxLinkUtil as one possible option for the objective function, assuming optimizing the utilization of nodes and links are equally important:

$$\text{minimize} \quad (0.5 \cdot \text{MaxNodeUtil} + 0.5 \cdot \text{MaxLinkUtil}).$$

Out of the results of each embedding run, for each combination of a set of service graphs, I have calculated the maximum link utilization and maximum node utilization in the network.

For each set of requests, the *preferred combinations* are the combinations corresponding to the results that belong to the Pareto set regarding these two metrics *after* the actual embedding. This should not be confused with the *selected combinations* that were chosen by the selection heuristic (described in Section 4.2) *before* performing the embedding.

To evaluate the quality of selected combinations, we need to quantify the differences and the relations between the set of selected combinations and the set of preferred combinations. For this, in the following sections, I present two views:

- In Section 4.3.1, I evaluate the preferred combinations that were not selected by the heuristic in each embedding run (Figure 4.4a)

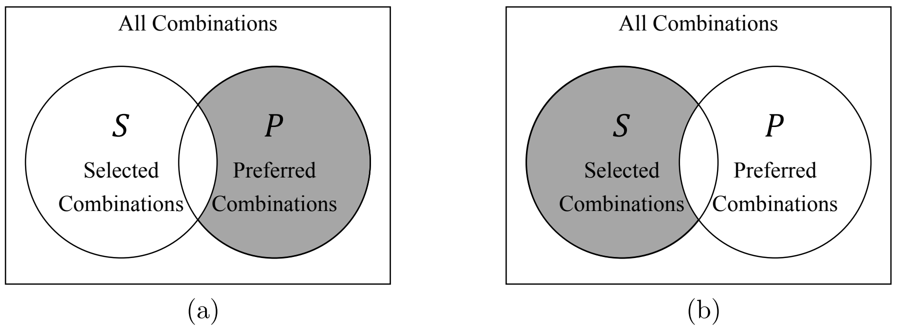


Figure 4.4: Sets of combinations chosen by the selection heuristic and combinations that give the best embedding results (preferred combinations) among all possible combinations of service graphs resulting from pliable VCSs

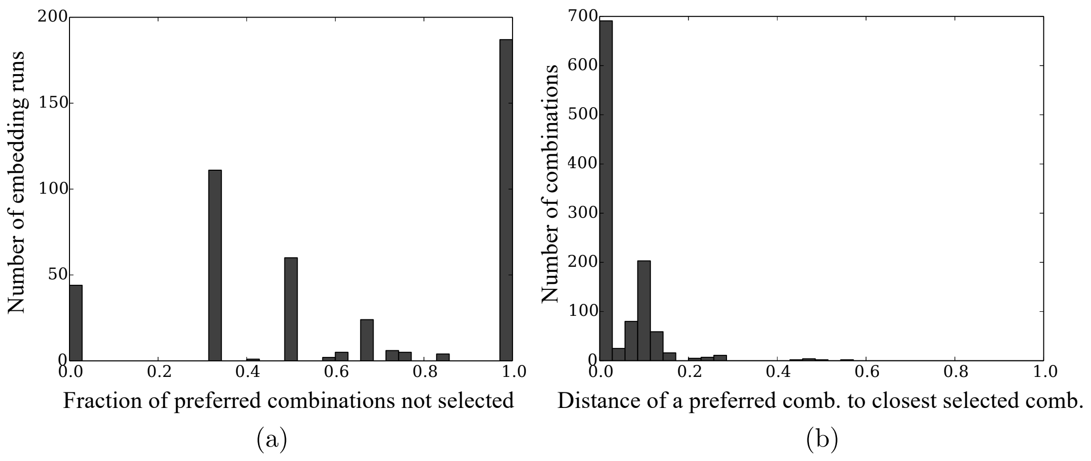


Figure 4.5: Evaluation of preferred combinations that are not selected by the heuristic

- In Section 4.3.2, I evaluate the selected combinations by the heuristic that do not belong to the set of preferred combinations in an embedding run (Figure 4.4b)

In Section 4.3.3, I describe the gain in decision time that using the selection heuristic can provide.

4.3.1 Preferred Combinations not Selected by Heuristic

Figure 4.5a shows the histogram of the fraction of the preferred combinations in embedding results that were *not* selected by the heuristic.

For each embedding run, I have calculated the *false negative rate* as

$$\mathrm{FN} = \frac{|P \setminus S|}{|P|},$$

where P is the set of preferred combinations in that run and S is the set of selected combinations by the heuristic. $|X|$ denotes the cardinality of set X and $\setminus$ denotes the set difference. $P \setminus S$, the set of preferred combinations that are not selected, is shown in Figure 4.4a. In terms of classification theory, FN is the false negative rate, if we consider the selection heuristic as a classifier to detect preferred combinations.

The values show that in many embedding runs, none of the preferred combinations are included in the combination set chosen by the heuristic. This result is not necessarily a negative outcome, depending on how well the selected combinations can represent the preferred but not selected combinations. I have done further experiments to further investigate this.

To evaluate the importance of the combinations that were missed by the heuristic, I have calculated the max-norm distance of each preferred combination to the closest selected combination. In a vector space, the max-norm distance of two vectors, also known as the chessboard distance, is their greatest difference along any dimension. This can give a meaningful comparison of the quality of two combinations in this case; the max-norm distance of two combinations reflects the difference in the resulting maximum link utilizations or the difference in maximum node utilizations (depending on which one is more significant) when these combinations are embedded into the network. The largest possible distance between two combinations according to this metric is 1.

Figure 4.5b shows the resulting histogram. For the majority of preferred combinations, there is at least one selected combination with a distance close to 0. For the remaining combinations, the distance is negligible. The largest recorded distance has a value of around 0.54 and has been recorded for 2 combinations out of around 1100 preferred combinations over all embedding runs.

From this, I conclude that in spite of the large number of preferred combinations that are not included in the set of combinations selected by the heuristic, the variety of possible service graphs and resulting resource demands in the set of all possible combinations is indeed captured by the heuristic results.

4.3.2 Selected Combinations not in Preferred Combinations

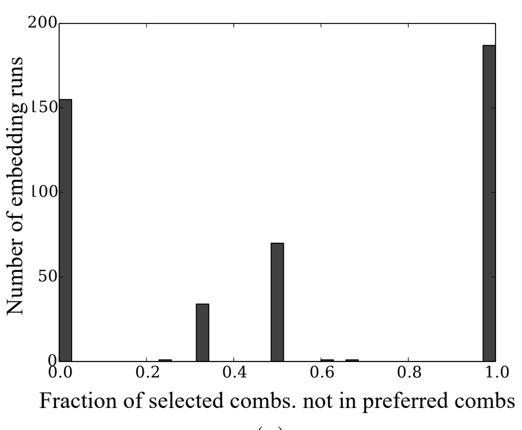
For the second part of the evaluation, Figure 4.6a shows the histogram of the fraction of selected combinations that do not belong to the preferred combinations over different embedding runs.

For each service embedding run, I have calculated the *false discovery rate* as

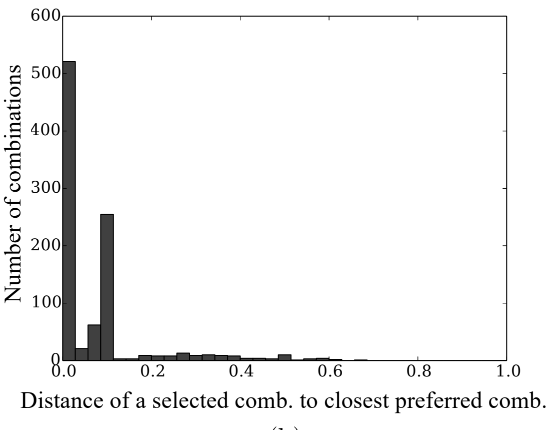
$$\mathrm{FD} = \frac{|S \setminus P|}{|S|}.$$

Figure 4.4b illustrates $S \setminus P$, as the set of selected combinations that were not among the preferred ones.

In more than 150 service embedding runs, all of the combinations selected by the heuristic belong to the preferred combination set. In close to 200 runs none of the selected combinations are in the preferred combination set. Figure 4.6b shows that for the majority of the selected combinations, the max-norm distance to the



(a)



(b)

Figure 4.6: Evaluation of selected combinations that are not among preferred combinations after embedding

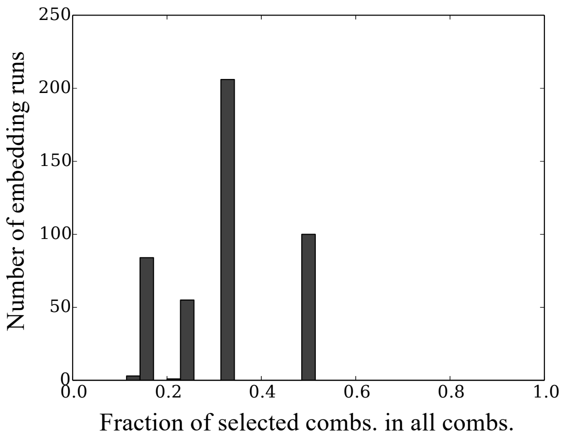


Figure 4.7: Fraction of selected combinations among all combinations over all embedding runs

closest preferred combination is close to 0 and negligible. The largest distance has a value of around 0.66, which is recorded for one combination out of around 970 selected combinations over all embedding runs.

From these observations, I conclude that the combinations selected by the heuristic can closely represent the preferred results in these embedding runs.

4.3.3 Gain in Decision Time

Figure 4.7 shows the ratio of the combinations selected by the heuristic to the total number of combinations over all embedding runs. As illustrated in Figure 4.1, the required time to reach a final embedding decision depends on the number of times the embedding process needs to be repeated. This, in turn, is determined by the number of selected combinations.

Using the selection heuristic described in Section 4.2, the embedding needs to be calculated less than half as often as the case where no heuristic is applied.

However, most of the service deployment requests were small requests, resulting in cases similar to the Pareto set shown in Figure 4.3a. Comparing this case to Figure 4.3b shows that the larger the number of VCSs is, the larger the number of possible combinations is, and the ratio of selected combinations to all combinations tends to decrease.

Therefore, I conclude that for large sets of pliable VCSs including arbitrarily ordered components, the multi-solution selection heuristic can select combinations of service graphs that can result in optimal or close-to-optimal solutions after embedding, in significantly less time compared to the option of exploring all possible combinations.

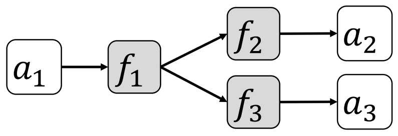
4.4 Service Embedding Heuristic Approach

In this section, I describe a heuristic that can very quickly find a close-to-optimal solution for the service embedding problem as described in Section 3.2.3.

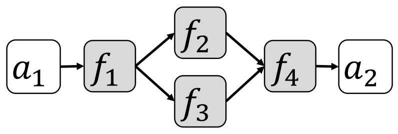
While the objective function in an optimization approach can be replaced to optimize the values for different metrics as necessary, for a heuristic, the objective should be a part of the algorithm design. My model is based on the assumption that the substrate network is a geographically distributed network, e.g., a large-scale telecommunications operator's network, with multiple data and compute centers connected to each other. In these networks, applications like video streaming and file sharing are increasingly taking up link capacities and need low-latency paths. Moreover, in such scenarios, it is important to place the VCSs in a way that enough capacity is left on the network links to satisfy larger number of subsequent requests as well as requests with large data rates.

Considering these requirements, I have designed a heuristic that embeds a set of VCSs in a way that the traffic between the endpoints of a VCS is routed through the shortest path whose bottleneck link has just enough capacity for the requirements of the VCS (smallest-fit first). The bottleneck link on a path is the link with the smallest capacity along this path. Among different paths with equal lengths that can carry the required data rate, the algorithm selects the path with the smallest bottleneck so as to leave the paths with a higher capacity for serving other, possibly larger requests. Similarly, among different paths with equal bottleneck capacities, it prefers the path with the least number of hops. A path is accepted for embedding only if the nodes along this path have enough capacity to host all of the components of the VCSs and the latency of the path matches the end-to-end latency requirements of the VCSs.

For calculating the paths, I have used one of the variations of the bottleneck shortest paths problems, an algorithm to solve the *single-source shortest paths for all flows problem* (SSSP-AF) by Shinn and Takaoka [95]. In this problem, it is assumed that a set of flows with different data rates are given as a set W and the network links have limited capacities. For every flow with a data rate $w \in W$, the goal is to find the shortest paths (with respect to the number of hops) from a source node to all other nodes in the network such that the paths can carry flows with data rates of up to w. The output of this algorithm is a set of tuples (d, w)



(a) Two different end points: a_2 and a_3



(b) Two simple chains between its start and end points: $a_1 \rightarrow f_1 \rightarrow f_2 \rightarrow f_4 \rightarrow a_2$ and $a_1 \rightarrow f_1 \rightarrow f_3 \rightarrow f_4 \rightarrow a_2$

Figure 4.8: Example service graphs

for each destination node, where d is the number of hops in the shortest path that can carry flows with data rates up to w. This algorithm has a complexity of $\mathcal{O}(mn)$, where m is the number of links and n is the number of nodes in the substrate network graph [95].

The input and the basic assumptions of the heuristic for service embedding are identical to those of the optimization approach (Section 3.2). Some of the important assumptions are as follows.

For the heuristic design, I assume that the service deployment requests include an (exact or estimated) upper bound for the input data rates at the starting points of the VCSs and the input data rates to all subsequent components in the VCSs can be calculated based on given expected ROID for each component. Moreover, I assume the start and end point locations of service flows are given. They can be, for example, one of the network nodes where the requests for a group of end users in a specific geographical location enter the network or the back-end server of the application that is already placed and used by other instances of the VCS, or a physical network function with a fixed location that needs to be used along with other service components.

To parse the deployment requests and to build service graphs out of service composition requests, the requests go through a pre-processing step (step ② in Figure 4.1). In this step, the exact resource demands of the components (which are given per unit data rate in my model) are also calculated using the input data rate to the VCS. The service embedding heuristic can replace the service embedding optimization approach in step ③ of this figure.

An overview of the steps of this heuristic solution is shown in Algorithm 4.1. The input to this algorithm is a combination of service graphs that need to be mapped to the network, e.g., one of the selected combinations given by the selection heuristic in Section 4.2.

The combination of service graphs is a (disconnected) directed graph annotated with data rates of the logical links between pairs of service components, resource demands of the service components (e.g., given as tuples of compute, memory, and storage requirements), the end-to-end latency that can be tolerated by each VCS, and the location of the start and end points of VCSs in the network. The substrate network graph is also given, annotated with the capacity and the latency of network links, and the capacity of nodes (e.g., in terms of available compute, memory, and storage resources).

Service graphs might have more than one end point, as shown in Figure 4.8a,

Algorithm 4.1 Heuristic for embedding a combination of service graphs

```
1: $C \leftarrow$ a combination of annotated service graphs
2: $G \leftarrow$ annotated substrate network graph
3: function EmbedCombination($C$, $G$)
4:     $A \leftarrow$ sorted list of pairs of start and end points of services
5:     for $(a_s, a_e) \in A$ do
6:         chains$_{a_s,a_e} \leftarrow$ sorted list of simple chains between $a_s$ and $a_e$
7:     for $(a_s, a_e) \in A$ do
8:         for $c \in$ chains$_{a_s,a_e}$ do // Every chain $c$ is a subgraph of $C$
9:             EmbedChain($c$, $G$)
10:            if embedding successful then
11:                if latency of created path $\leq$ latency bound from $a_s$ to $a_e$ then
12:                    embedding for $c$ is valid
```

Algorithm 4.2 Embedding algorithm for a simple chain

```
1: function EmbedChain($c$, $G$)
2:     $R \leftarrow c$ // Remaining pairs of components from the chain to be placed
3:     $D \leftarrow$ data rate between each component pair
4:     for $(f_1, f_2) \in c$ do
5:         $p_s \leftarrow$ location of $f_1$ // Current location
6:         if $f_2$ is not already embedded then
7:             if demands of $f_2$ can be satisfied by available resources on $p_s$ then
8:                 embed $f_2$ on $p_s$ and update $G$
9:             else
10:                $d \leftarrow \max_{\forall(x,y)\in R}(D(x,y))$ // Largest data rate over all pairs
11:                $p_e \leftarrow$ location of first embedded component in $c$ after $f_2$
12:                $P \leftarrow$ GetPath($p_s, p_e, d, G$) // Ordered set of nodes in the path
13:                while there are unexplored nodes on $P$ do
14:                    $v \leftarrow$ next node on $P$
15:                    if requirements of $f_2 \leq$ available resources on $v$ then
16:                        embed $f_2$ on $v$ and update $G$
17:                        break // Embedding of $f_2$ done, stop iterating over $P$
18:                if there are no more nodes to explore on $P$ then
19:                    return $c$ cannot be embedded
20:            $R \leftarrow R \setminus (f_1, f_2)$ // Remove $(f_1, f_2)$ from pairs to be placed
21:    return embedding results and updated $G$
```

and different branches between one pair of start and end points, as shown in Figure 4.8b. For such VCSs, the algorithm finds and stores every *simple chain* of components between every pair of start and end points in the service graph. A simple chain is a subgraph of the service graph with a linear structure, which starts at the start point of the VCS and ends at one of the end points of the VCS, without loops and branches. Each of the simple chains in a service graph might have different requirements, e.g., different data rate over their edges, so the embedding of them needs to be prioritized and regulated.

In line 4 of Algorithm 4.1, pairs of start and end points (a_s, a_e) of different VCSs are sorted in decreasing order according to the sum of data rates over all virtual links between them and stored in A. Similarly, if there are multiple simple chains between one start and end point, in lines 5 and 6, the simple chains are sorted in decreasing order according to the sum of data rates over all virtual links among them and stored as a list called chains_{a_s,a_e} for every (a_s, a_e) in A. This ordering ensures that the chains with higher data rates are placed first and have a higher chance of being mapped to shorter paths.

Every simple chain is an ordered list of pairs of service components (f_1, f_2) (including the start and end points), such that f_1 and f_2 are nodes of the combined graph C, (f_1, f_2) is a virtual link in graph C, and by following the pairs in a chain in the given order we can get from the starting point of that chain to the end point of the chain after traversing all of the service components in that chain.

The embedding starts with the *heaviest* pair of start and end points, i.e., the pair that has the largest sum of data rates over its virtual links. If there are multiple simple chains between them, the algorithm starts with the *heaviest* one. In line 9, the EmbedChain function is called, which calculates the embedding for the input chain c on the substrate network graph G. This function is shown in Algorithm 4.2.

If the embedding is successful, the results are returned together with the updated network graph G (e.g., with less capacity on its nodes and links after accommodating the embedded chain). If the latency requirements of the embedded chain are met, the embedding is accepted as valid. For simplicity, this algorithm does not include any backtracking steps in case the embedding is unsuccessful or invalid. Implementing the backtracking using different path options provided by the SSSP-AF algorithm [95] is straightforward.

The EmbedChain function in Algorithm 4.2 iterates over the pairs of service components (f_1, f_2) (i.e., virtual links in the chain) and embeds them one by one. As the start point of the chain is also a part of the chain, while iterating over these pairs, in the simplest cases, f_1 is already mapped to a location in the network and the algorithm needs to find the location for f_2. For this, in line 7 and 8, similar to the behavior of the optimization approach, the algorithm first checks the feasibility of embedding f_2 onto the same node where f_1 was embedded. If that is not possible, in line 12, it calculates the shortest path towards the end point that has enough capacity for the largest data rate over the remaining parts of the chain to be embedded, based on the SSSP-AF algorithm.

As an input for path calculation using the SSSP-AF algorithm, in line 10, the algorithm finds the largest data rate over the pairs of components that still need to be embedded.

SSSP-AF calculates the paths from a given node towards all other nodes in the network, out of which the algorithm only needs the paths towards one specific end point. Function GetPath processes the output from SSSP-AF to extract the path P towards this end point as an ordered list of the network links belonging to this path.

The end point p_e used by GetPath for calculating P is equal to the location of a_e (end point of the chain) if the current chain has no overlaps with another chain that has already been placed in the network. Figure 4.8b shows an example of such an overlap. I assume the simple chain $a_1 \to f_1 \to f_2 \to f_4 \to a_2$ is the heavier chain and needs to be embedded before the other chain between a_1 and a_2. For embedding the first chain, for all component pairs in it, p_e is set to the location of a_2 and GetPath is called to find the shortest path between the current node and the location of the end point of the chain. However, while embedding the second chain $a_1 \to f_1 \to f_3 \to f_4 \to a_2$, the component f_4 has already been embedded, so in line 11, p_e is set to the location of f_4 and GetPath will be called to find the shortest path towards that node. Moreover, to avoid calculating the embedding more than once for components like f_1 and f_4 that appear on multiple chains of a VCS, in line 6 the algorithm checks if the component has already been mapped to a node.

Once a component is embedded into the network, for embedding the next component in the chain, the algorithm does not rely on the previously calculated shortest path towards the end; instead, it re-calculates the path for every new (f_1, f_2) to be embedded. In this way, it can make use of the opportunity that once the virtual link with the largest data rate on the chain has been mapped to the network, the subsequent components can be placed along a path that might be even shorter than the one initially calculated.

In the Section 4.5, I evaluate the performance and run time of this heuristic.

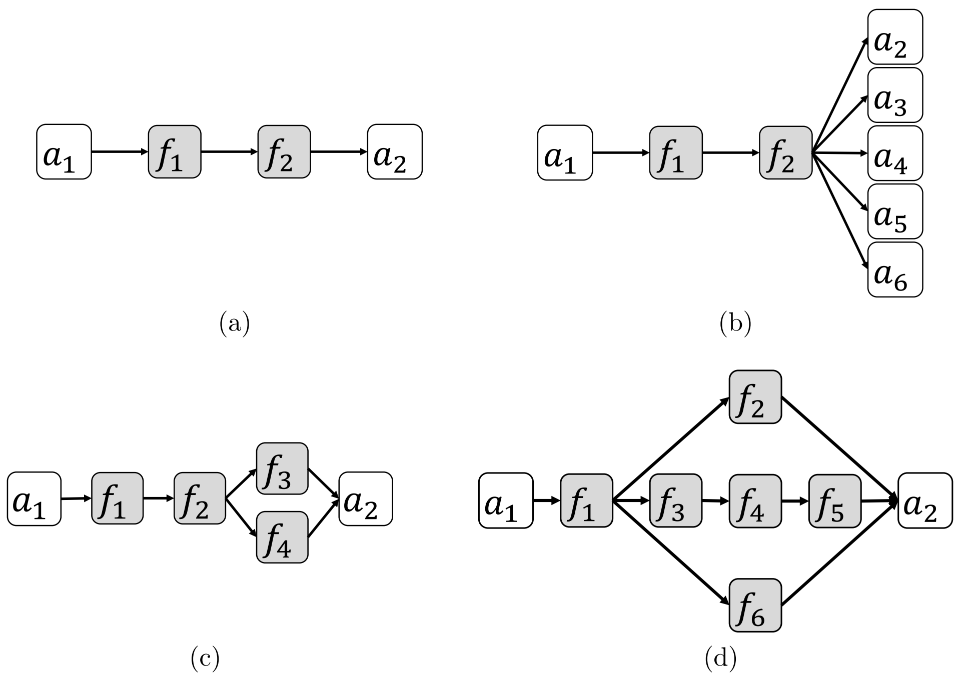
4.5 Evaluation of Service Embedding Heuristic

As described in Section 4.4, I have designed the service embedding heuristic in a way that the VCSs are embedded into network nodes along short paths with acceptable latencies and just enough link capacity. I evaluate the solutions produced by this heuristic against the optimal embedding results obtained using the optimization approach described in Section 3.2.3. My implementation of this heuristic is available online [37].

For pliable VCSs, I have used the service embedding heuristic for embedding the combinations selected by the selection heuristic (Section 4.2): among different options for generating service graphs for VCSs that include arbitrarily ordered components, some are selected and passed one by one to the embedding step as a fixed and explicitly defined graph. Therefore, the VCSs I have used for evaluating the service embedding heuristic are service graphs with totally ordered components.

I have used 5 different sets of example service deployment requests that include simple chains as well as more complex branched structures. As in the evaluation of the selection heuristic, the structure of the example VCSs is based on the IETF SFC draft on general use cases for SFC [93]. I have selected the sets of VCSs as follows, in increasing complexity, based on the time required by the optimization approach to find a solution for them:

- *Set 1* includes 4 simple chains each having a structure as shown in Figure 4.9a.


(a)
a_1 f_1 f_2 a_2
(b)
a_1 f_1 f_2 a_2 a_3 a_4 a_5 a_6
(c)
a_1 f_1 f_2 f_3 f_4 a_2
(d)
a_1 f_1 f_2 f_3 f_4 f_5 f_6 a_2
Figure 4.9: Service graphs used for evaluating the service embedding heuristic

- *Set 2* includes 3 VCSs, one having a structure as shown in Figure 4.9b and two having a structure as shown in Figure 4.9c.
- *Set 3* includes 8 simple chains each with a structure as shown in Figure 4.9a, making this set similar to *Set 1* in structure.
- *Set 4* includes 7 VCSs in total, four simple chains like *Set 1* and three VCSs like in *Set 2*.
- *Set 5* includes 3 VCSs with a structure as shown in Figure 4.9d.

These sets include VCSs with different structures, i.e., simple chains of components, VCSs with converging branches, and VCSs with independent, diverging branches. The components used in the VCSs can split incoming flows over different outgoing branches, increase or decrease data rates of incoming flows, or forward them without modifying the data rate. The combination of these cases represents what a typical embedding algorithm would need to deal with, highlighting various aspects of the heuristic approach, e.g., making sure the flows converge towards the required components after being distributed over different branches, with the right flows reaching the right endpoints, etc.

For each of these sets, I have performed 300 embedding runs. Each run uses a new random seed for setting up the input. In each run, I have calculated the embedding once using the heuristic and once using the optimization approach.

Similar to the evaluation setup in Section 4.3, I have used the *abilene* network from SNDlib [94] as the substrate network.

I have mapped the start and end points of the service graphs to random nodes in the network in each run. For one set, the sum of input data rates for the VCSs is always the same for all runs, with each VCS getting a new, randomly assigned share of the total data rate as input in each run. In different runs, different amounts of data rate need to be routed among different network nodes. I have used this approach to create enough variation in the amount and sources of the traffic in the network, while keeping the total input load in a fixed level to reduce cases where the mapping is infeasible and no insight is provided for comparing the optimization approach to the heuristic approach.

I have chosen the following objective function for the optimization approach:

$$\text{maximize} \quad \sum_{(v,v')\in E, v\neq v'} \text{remcap}_{v,v'}$$

where E is the set of links in the substrate network and $\text{remcap}_{v,v'}(\forall(v,v') \in E)$ shows the remaining capacity on every link (v,v') after the embedding. The value of $\text{remcap}_{v,v'}$ is calculated by subtracting the data rate of every flow that passes (v,v') from the capacity of the link. I exclude the internal links of network nodes (self-loops in the network graph) from the objective function and maximize $\text{remcap}_{v,v'}$ only for those links $(v,v') \in E$ where $v \neq v'$. In this way, I force the embedding to prefer these internal links over other links. This results in consecutive components in a VCS to be mapped to the same network node as long as there is enough capacity on the current node to host the next component (also taking other constraints of the optimization approach into account).

The embedding that is computed using this objective function has the maximum possible value for mean remaining capacity over all network links, excluding the self-loops. Therefore, as a first step to compare the results of the heuristic to the optimal results, in Figure 4.10a I show a comparison between the mean remaining capacity over all network links for each of the VCS sets based on the results of all embedding runs.

The results of the heuristic, with respect to the metric that is optimized by the optimization approach, are very similar and in some cases almost identical to the optimal results. The largest difference between the results of the heuristic and the optimization approach is around 5 % and belongs to the results of *Set 1*. The plots (Figure 4.10a–4.10e) include confidence intervals at 95 % of confidence level.

To highlight the differences better, I ignore the unused network links and in Figure 4.10b, I show a comparison between the mean remaining capacity only over those links that were used for mapping the VCSs in each run.

The largest difference between the results is around 22 % and can be seen in the results of *Set 1*. The optimization approach can handle the embedding of this set much better than the heuristic, partly due to the type of the components used in the VCSs in this set. These VCSs consist of simple chains of components and one of the components in each chain has a ROID larger than 1. That means, the input data rate to the components increases through an intermediate component in the chains. The optimization approach obviously tries to map the two components with such a large data rate between them into one node. By mapping the link

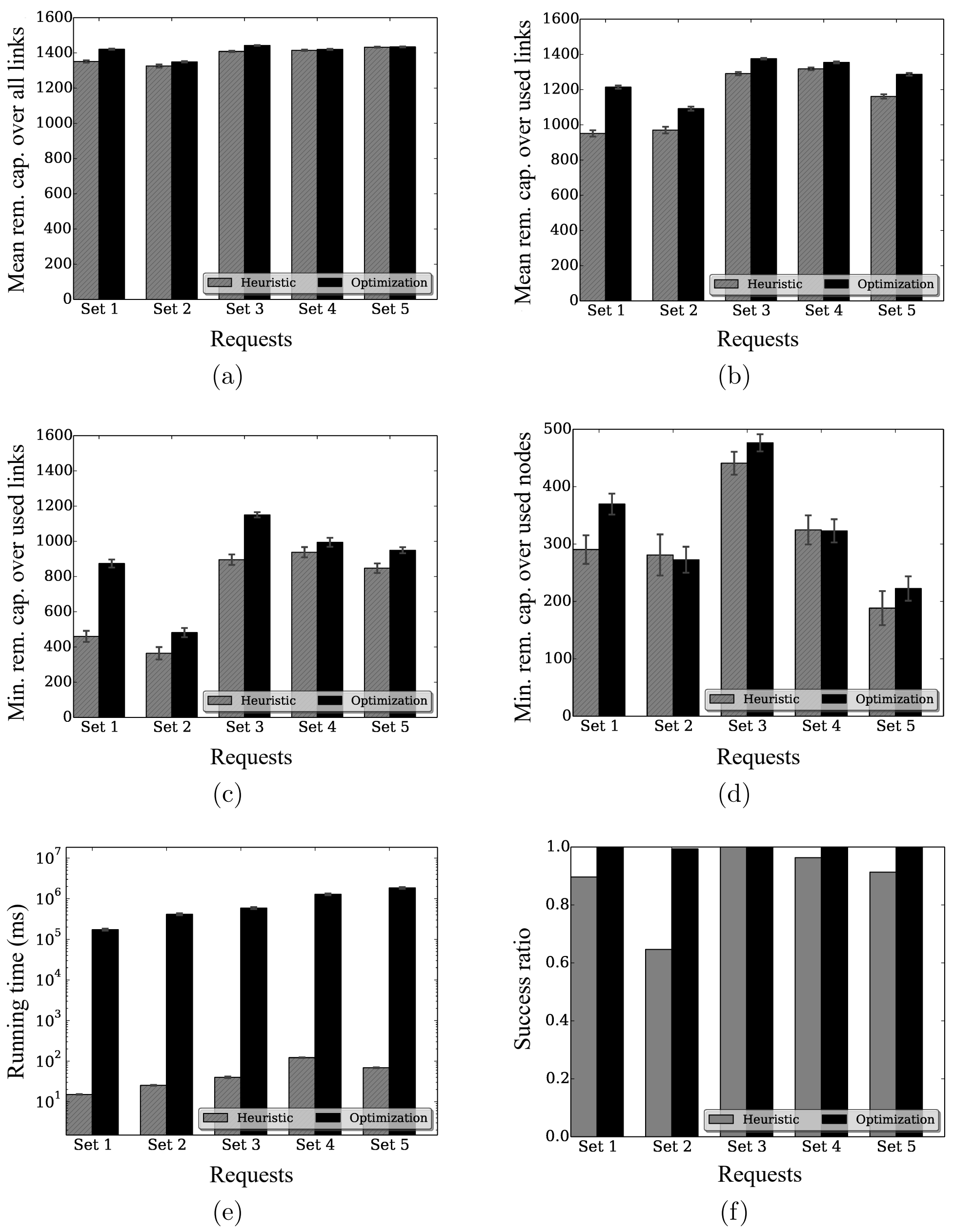


Figure 4.10: Evaluation results for the service embedding heuristic approach

between them to an internal link in the node (a self-loop), this part of the flow does not consume capacity on inter-node links. The heuristic, in contrast, cannot foresee this increase in the data rate and simply embeds the components into nodes with enough capacity along the shortest path that can carry the maximum data rate over the chain.

The same effect can be observed in other sets as well. However, the difference between the results is influenced by other parameters, as well. For example, the VCSs in *Set 3* have the same structure as those in *Set 1* (simple chain) and include some components that increase the data rate. The difference of results in this case is smaller than the difference for *Set 1*, because the number of VCSs in *Set 3* is twice as much as the number of VCSs in *Set 1*; that is, the input data rate to each VCS is smaller in *Set 3* and hence, the data rate of the flow after leaving the component that increases the data rate is smaller and has a smaller effect than it has in the case of *Set 1*.

The difference of the heuristic results to optimal results is also affected by the fact that the VCSs with complex structures are broken into simple chains and the chains are mapped to the network one by one using the heuristic. This is reflected, for instance, in the embedding results for *Set 2* and *Set 5*, which consist of VCSs with branches in their structure. In such VCSs, the location of the service components that appear in more than one simple chain in the service structure (e.g., f_1 in Figure 4.9d, which is a part of all 3 simple chains between a_1 and a_2) is determined only based on the first chain that is embedded. The optimization approach, in contrast, considers the whole VCS at the same time and can find better a better mapping.

I also evaluate the *minimum* link capacity that remains in the network after the embedding, over the links that are used for the VCSs. The heuristic always selects the path with the smallest bottleneck value among all the paths that have enough capacity for the data rate of the VCS that is being embedded. As shown in Figure 4.10c, compared to the results of the optimization approach, this behavior does not cause much higher chances of congesting the network links. The largest difference is around 52 % and again belongs to *Set 1* as described before. However, for VCSs with a more complex structure, like in *Set 2* and *Set 5*, the heuristic approach can get as close as 90 % to the behavior of the optimization approach regarding this metric.

Although none of the embedding approaches explicitly attempts to optimize the usage of resources on network *nodes*, in Figure 4.10d I show the remaining capacity on the most congested network node after the embedding. There is no significant difference between the behavior of the heuristic and the optimization approach in this regard.

The difference in the run times of the two approaches is shown in Figure 4.10e, on a logarithmic scale. *Set 5* is the most complex input for the optimization approach among my test sets. Finding a solution for this set requires around 31 minutes on average in my test environment, using the Gurobi Optimizer [21] on a machine with Intel X6560 CPUs running at 2.67 GHz. On the same machine, the heuristic can find a solution in around 69 milliseconds.

As described in Section 4.4, no backtracking step is included in the heuristic that would, for example, try another path in case a chain cannot be placed along the first suitable path found by the SSSP-AF algorithm. Therefore, I show a comparison of the success ratio between the algorithms in Figure 4.10f. I consider a run as successful if an embedding can be calculated for the complete set of requested VCSs in that run.

A failure ratio of around 35 % can be observed for *Set 2*. Resource demands of the components increase with the incoming data rate. As the input data rates and start and end locations are assigned randomly in each run, the resource demands of the components can differ greatly over different runs. Therefore, in some cases, the embedding is simply not feasible because the network does not have enough resources to host the VCS. Because of the branched structure and ROID of individual service components in *Set 2*, this effect is stronger for this input set. Even the optimization approach cannot find a solution for all of the runs using this set. The results can be improved, for example, by trying the second-best path when the embedding of a chain fails. For other sets, the success ratio is acceptable. As described in Section 3.2.1, I assume every data center node in this model likely has enough capacity for hosting a reasonable number of service components. Therefore, as long as there is a path in the network that has enough capacity to route the required traffic between the start and end point of a chain, the heuristic will find this path and place the components on the nodes along this path.

4.6 Conclusion

In this chapter, I have described a heuristic for selecting a representative subset of candidate service graphs and combining the service graphs from different pliable VCSs that include arbitrarily ordered service components. The output is a set of Pareto-optimal points with respect to different metrics that represent possible variations in the structure and resource demands of the pliable VCSs.

I have shown the evaluation results for the selected service graph combinations using a joint placement, scaling, and routing optimization (A-SPRING) approach, configured for congestion control in network nodes and links. The results show that the selected combinations well represent the variety of options in the Pareto set of all possible service graph generation and combination options. For large sets of pliable VCSs that include arbitrarily ordered components, the selection heuristic can reduce the decision time by eliminating at least half of the possible embedding options.

The evaluation results for this approach show the feasibility of defining pliable VCSs with arbitrarily ordered components, circumventing the extensive computational overhead for calculating the optimal scaling, placement, and routing for all possible candidate service graph that may result from these pliable VCSs.

Depending on the optimization objectives for a group of pliable VCSs and the network where they will be deployed, the selection heuristic may produce more than one suitable combination for the service graphs of all pliable VCSs. These combinations can be used in different ways. For example, one can try embedding all selected combinations and pick the best option for actual deployment. Another interesting possibility would be to change the structure of services for adapting the deployments to the network state. For example, by categorizing the selected combinations according to specific metrics of interest and using different combinations over time. One can do the initial deployment using a combination with the lowest compute resource requirements and switch to the one with the lowest link capacity requirement if a high link utilization is detected. If the current deployment of a service is facing performance issues due to link capacity problems, a new composition of the service components with a more appropriate link utilization may be selected and deployed for serving the resilience objectives of VCSs and the underlying network.

I have also presented and evaluated a heuristic for finding quick and close-to-optimal solutions to the service embedding optimization problem. This algorithm can be used for embedding pliable or fully defined VCSs. The evaluation results shows a maximum of 5 % deviation of the heuristic results from optimal results, designed with the objective of maximizing the mean remaining data rate on network links.

This fast embedding approach provides more opportunities as a result of defining pliable VCSs with arbitrarily ordered components. It reduces the computation time for calculating a close-to-optimal embedding for every combination of different candidate service graphs for a group of pliable VCSs. In this way, if necessary, *all possible combinations* of different service graph candidates can be evaluated before the actual deployment. The results can be stored and used for better scaling and adaptation decisions, e.g., by switching to a more suitable ordering of service components for a group of already deployed pliable VCSs, to accommodate additional services or to react to changes in the load.

Like every other adaptation mechanism, changing the order of traversing service components in a pliable VCS may require additional re-configuration of deployed instances, re-routing service flows, or state migration to ensure an uninterrupted service delivery. I further discuss the practical applicability of the presented approaches in Chapter 9.

5 Services with Load-Proportional Structures

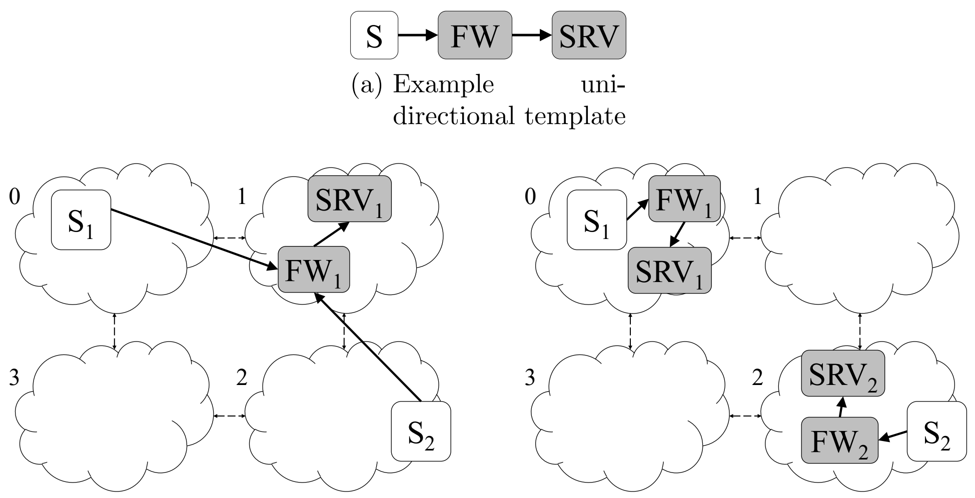
In Chapter 3, I have introduced the pliable Virtualized Composed Services (VCSs) with arbitrarily ordered components and addressed the embedding problem for these VCSs, the A-SPRING problem, in Chapter 4.

In Section 5.1, I describe the challenges that I tackle in the following three chapters, regarding pliable VCSs with load-proportional structures. An important requirement for describing these VCSs using service templates and the following template embedding process is a formal way of describing the resource demands of VCSs based on the load they need to handle. To show the feasibility of such a description, in Section 5.2, I present the results of some experiments and example description and formalization methods that can be used in service templates for pliable VCSs with load-proportional structures. This chapter partially includes figures and verbatim copies of the text from my papers [26, 31, 32, 34, 30].

5.1 Challenges

In Chapter 3, I have described pliable VCSs including a set of components that can be traversed in an arbitrary order. Instead of using the conventional descriptors, these pliable VCSs can be described using a more flexible specification model (Section 3.2.1). Such a flexibility in the service structure is useful and practical only if the resulting service functionality remains intact after changing the order of traversing the components. The specification and embedding approaches described for pliable VCSs in Chapter 3 and 4 are not sufficiently comprehensive and generic to address the limited precision and flexibility of conventional descriptors and service life-cycle management approaches for all kinds of VCSs.

Therefore, I extend the notion of pliable VCSs by focusing on *VCS with load-proportional structures*. I assume the case that is more commonly considered in



(a) Example uni-directional template

(b) Embedding option 1

(c) Embedding option 2

Figure 5.1: Example embedding options for a simple uni-directional template

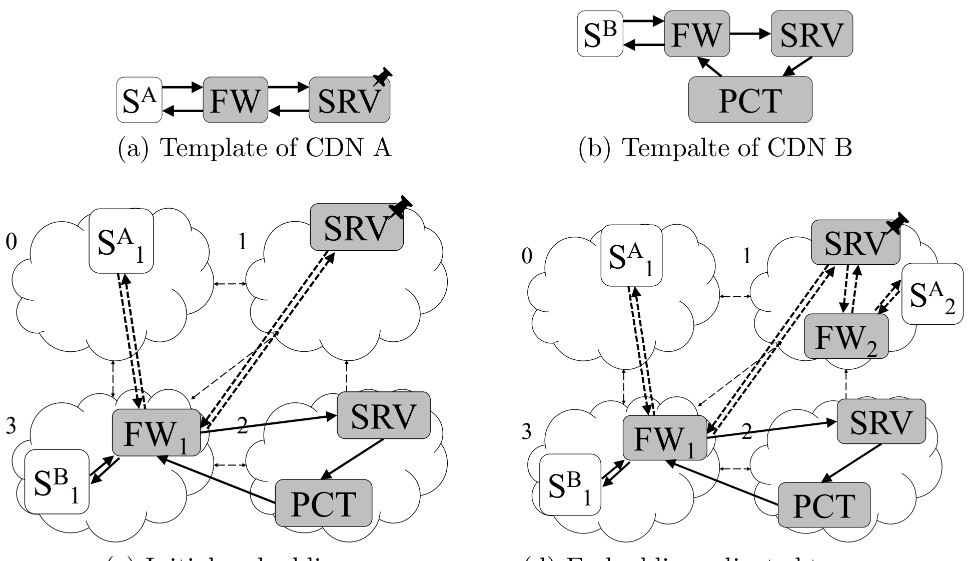
related studies, where the order of traversing the service components is pre-defined and fixed. Instead, the number of instances required for each service component and the resource demands of each instance depend on the data rate that each component should process. This, in turn, is influenced by the data rate and the distribution of the sources of service flows in the substrate network.

As described in Section 1.1, these VCSs can be uni-directional, e.g., considering only the forwarding direction from service components towards the users, or bi-directional, e.g., considering the incoming requests from users and the flows sending back the requested content to the users.

Figure 5.1a shows a simple uni-directional VCS, which consists of a firewall (FW) that should be applied to the requests initiating from sources (S) towards a virtual server (SRV).

This simple template can be embedded into the example substrate network shown in Figure 5.1 in multiple ways. Two possible embedding options (among many) are illustrated in this figure. For finding the best embedding, different trade-offs should be considered. For example:

- Option 1 shown in Figure 5.1b results in a lower number of instances compared to option 2 in Figure 5.1c, possibly resulting in lower resource consumption in idle times.
- Option 2 results in lower latency for the users represented by sources S_1 and S_2 than option 1.
- Option 1 requires high resource capacity on node 1 of the substrate network to handle the data rate of the flows initiating from both sources. Each instance of the FW and SRV components in option 2 have a lower resource demand than the corresponding instances in option 1.
- Option 1 consumes link capacity for the node 0–node 1 connection as well



(a) Template of CDN A (b) Tempalte of CDN B

(c) Initial embedding (d) Embedding adjusted to new source

Figure 5.2: Example embeddings of two bi-directional VCSs

as the node 2–node 1 connection, while the virtual links required for the embedding option 2 can be realized within node 0 and node 2.

I consider these challenges for embedding uni-directional pliable VCSs in Chapter 6.

Figure 5.2a and Figure 5.2b show the structure of two example bi-directional VCSs that model Content Delivery Network (CDN) services A and B, respectively. Figure 5.2c shows an example embedding of these two VCSs. Each VCS has its own user group, represented by *sources* S_1^A and S_1^B. In VCS A, user requests go through a stateful firewall (FW) towards a content distribution server (SRV), deployed as a Physical Network Function (PNF) in node 1; the requested content returns to the users through the same firewall. VCS B is a virtualized version of this CDN and, additionally, requires the service flows to go through a parental control function (PCT) before returning to the users through the firewall. In this example, all of the flows require the same functionality and configuration from the firewall. Therefore, the requests can be mapped to the same instance of the stateful firewall to reduce resource costs. This requires sufficient resources at the node where the firewall is deployed to handle all of the requests.

If the provider of VCS A needs to expand its coverage to additional users in a different geographical location (Figure 5.2d), a new instance of the firewall might need to be instantiated in a suitable location. The embedding of both VCSs needs to be re-calculated (taking the existing deployments into account) to find the optimal number of instances required for the service components and their optimal location (where *optimal* can be defined per-scenario).

I tackle the problem of joint scaling, placement, and routing for bi-directional pliable VCSs in Chapter 7. In addition to the challenges described earlier in this section for embedding uni-directional VCSs, I also deal with the following challenges for bi-directional VCSs:

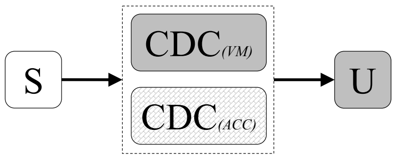
- Correctly routing different flows initiating from different locations in the network through instances of the required service components (as defined in the service template) back to their source location.
- Using the same instance of a stateful service component in upstream and downstream forwarding directions for each flow, to ensure state consistency and correct processing results. E.g., in Figure 5.2d, the content distribution server must forward the flows originating from sources $\mathrm{S}_1^{\mathrm{A}}$ and $\mathrm{S}_2^{\mathrm{A}}$ to the instance of the stateful firewall that has already seen the corresponding upstream flow.
- Incorporating service components with fixed locations and pre-defined resource demands in the template embedding approach (this challenge is not specific to bi-directional VCSs and can be incorporated in uni-directional VCS models, as well).

In Section 1.1, I have described another category of pliable VCSs with load-proportional structures, namely, the heterogeneous VCSs that consist of multi-version components. In Chapter 8, I model these VCSs as a variant of uni-directional pliable VCSs. In Chapter 8, I present solutions for the joint scaling, placement, and routing problem for heterogeneous pliable VCSs.

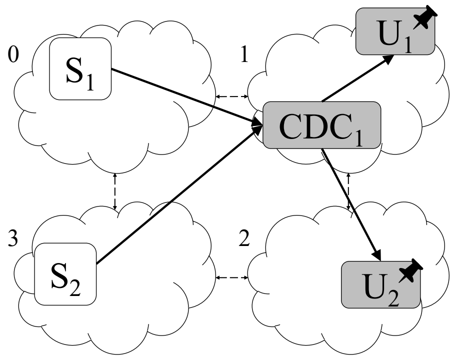
In addition to the trade-offs that should be considered for embedding ordinary uni-directional VCSs (as described earlier in this section), different deployment versions bring along additional challenges and trade-offs that I consider in my model and solution approaches for this problem.

Figure 5.3a shows the structure of an example heterogeneous VCS. In this VCS, videos are streamed from different servers (S) located in different nodes of the network towards pre-defined locations of different user groups (U). A Codec (CDC) function decodes and encodes the video streams, e.g., to adjust them to different end devices. This service component can be deployed as a Virtual Machine (VM) version (shown with a solid background) that uses CPUs for its processing or as a GPU-accelerated version (ACC) (shown with a patterned background) that consumes CPUs and (more expensive) GPUs for processing requests faster. In Figure 5.3b, I assume the VM version is the more efficient option for handling low data rates from the server, based on the cost model that is in place for using the resources. If the source data rate increases, the VM version would, however, require a larger amount of CPU allocation to be able to function with the expected performance. With even higher data rates, the VM version cannot operate efficiently anymore. The amount of load that each instance can handle is also limited by the capacity of the hosting node and the corresponding links in the network.

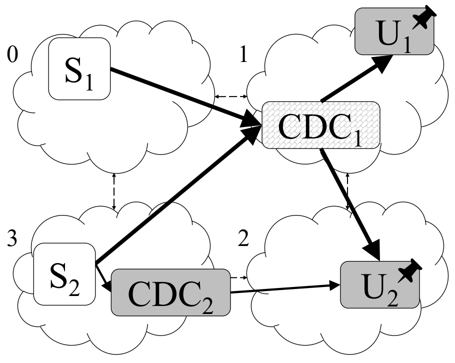
An example adaptation of this embedding is shown in Figure 5.3c. The challenge is to ensure the optimal number of instances of the CDC are deployed in the optimal locations (e.g., with respect to the latency and data rate of the created paths) using the best deployment version, considering the trade-offs between performance and cost. In this example, the initial VM version of the CDC is replaced



(a) Example heterogeneous template



(b) Initial embedding



(c) Embedding adjusted to increased data rate of the sources

Figure 5.3: Example embedding options for a simple heterogeneous template

by a GPU-accelerated version that processes most of the video streams from both sources. To avoid using too many of the costly resources for handling all of the video streams, it might even make sense to create an additional instance of the CDC. In this example, a smaller portion of the video streams are handled by a local VM-based version of the CDC in node 3 and the rest are forwarded to the more powerful remote instance.

In all of the mentioned approaches, the resource demands of each service component in the pliable VCSs are defined as a function of their incoming data rate in the templates. This deterministic relationship between the load and the resource demands can be determined using automatic service profiling methods or based on historical usage data. In Section 5.2, I show the feasibility of formalizing these relationships and show example methods for describing them.

Similar to the A-SPRING model, for pliable VCSs with load-proportional components, I assume an M/D/1 queuing model within each instance of a service component. In the U-SPRING and B-SPRING approaches, for simplicity, I do not consider the waiting times within service components. In these approaches, the deterministic service times can be added to the path delays to get the end-to-end latency for service flows. In the M-SPRING approaches, I assume the total time in system depends on the input data rate, as the waiting time increases when the input data rate increases. The actual service time for each instance, however, does not depend on the load.

5.2 Modeling Resource Demands and Performance

Together with M. Peuster and M. Illian, we have designed and conducted a set of experiments [30, 31] to understand the behavior of service components in different load situations and how the amount of allocated resources can influence the performance of a VCS. The data from these experiment is available online [96]. Some parts of this section are based on an initial version of these experiments done in the course of the bachelor's thesis of M. Illian [39], which are included here for completeness, marked with the corresponding references.

Existing descriptors for service components and how they are used by cloud and Network Function Virtualization (NFV) Management and Orchestration (MANO) systems have two serious shortcomings that we have highlighted using our experiments:

1. Resource demands of a service component depend on the load and the targeted performance. Therefore, defining a fixed and constant set of resources to be allocated to each service component can result in over-/under-estimating the required resources and lead to sub-optimal states for both the service and the underlying network. Moreover, predicting the relationships between the resource demands of service components (e.g., CPU, memory) and the targeted values of performance metrics of interest for each VCS (e.g., frame rate, video resolution) is cumbersome to do for a developer.

2. The resource demands and the performance of a component in a VCS depend on the allocated resources and the performance of other components in the VCS. Therefore, any attempt to model the resource demands of a component that is chained together with other components needs to consider the dependencies to other components as well as the dependencies to the load at each point in time.

We have set up a testbed to characterize such relationships in a video streaming scenario, which is a common application deployed on geographically distributed networks. In this section, I present some results from analyzing the data from these experiments. More details about the experiments and the results can be found in the corresponding publications [30, 31, 39].

We have performed a large set of performance measurements using a real-world VCS to collect the initial data needed to build and train realistic performance models. We have used a video streaming VCS consisting of a video encoder Virtual Network Function (VNF) (*FFserver* [97] and *FFmpeg* [98]) and a cache VNF (*Squid 3.5.12* [99]) configured and built with default settings and installed on Ubuntu 16.04 VMs. To simulate the users that access the video streams, we have used the HTTP client *GNU Wget* [100] deployed in an additional VM. We have measured both individually deployed service components as well as a fully deployed VCS.

Figure 5.4 shows our measurement setup, which is an OpenStack Ocata [101] testbed running on four physical machines with Intel(R) Core(TM) i5-4690 CPU running at 3.50 GHz with 16 GB memory. We have configured three of these

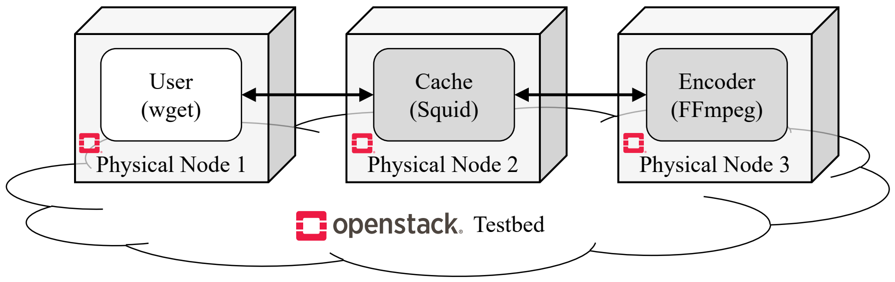


Figure 5.4: Measurement testbed with the used video streaming VCS (*Cache* ↔ *Encoder*) and the simulated users running in three VMs deployed on three physical compute nodes.

Table 5.1: Measurement Parameters

Parameter	**Values**
#vCPUs encoder	1 to 4
#vCPUs cache	1
Codecs	H.264, H.265
Videos	bunny [103], doc1 [104], doc2 [105], game [106], noise [107]
Resolutions	426x240, 640x360, 854x480, 1280x720, 1920x1080
Frame rates	24, 30, 40, 50, 60
Target bit rates	1000 Kb/s to 22000 Kb/s

machines as compute nodes so that each service component and the simulated user VM can be executed on its own physical machine, to eliminate noisy neighbor effects [102] from our measurements. All machines are interconnected with two 1 GigE links, one for control and one for the data plane.

In our experiments [30, 31, 39], the simulated users access a video stream delivered by the cache VNF and encoded by the encoder VNF on-the-fly. For each new experiment, the cache was restarted so that all streaming content had to be fetched from the encoder instead of using the cached streaming data. We collected the CPU and memory utilization of each of the used VNFs, transmission statistics (like data rates) between encoder and cache as well as between cache and users. We also recorded application-level metrics, like encoded frames per second. For each run, we configured the encoder VNF to compress a 60-second video, given in a resolution of 1920x1080, to a target resolution between 426x240 and 1920x1080, using target bit rates between 1000 Kb/s and 22000 Kb/s, frame rates between 24 and 60, and using either the H.264 or H.265 encoding standard. The used encoder requires a pre-set value for the trade-off between video quality and encoding time, which we set to *medium* for videos with low bit rates and good visual quality. Table 5.1 summarizes the full list of used parameters for the executed measurements. We have conducted a total of 18500 experiments with these configurations.

I show examples of the prediction models we have developed based on these experiments in the rest of this section.

We used models based on Support Vector Regression (SVR) and Polynomial

Regression (PR) for predicting the minimum number of Virtual CPUs (vCPUs) for the encoder to reach the desired performance. The training data we have used for creating the SVR-based model consists of the test runs where the actual frame rate was never below the targeted frame rate, using the minimum number of vCPUs among all such observations. For this purpose, we have discarded the experiments where the frame rate was below the target as a result of the selected configuration. We have set the parameters of the model by testing different values and evaluating them based on the resulting Mean Squared Error (MSE) and the visual representation of the models.

Figure 5.5a shows a plot for predicting the number of vCPUs based on bit rate, resolution, and frame rate. To be able to visualize this 4-dimensional relationship, we have fixed the bit rate to 5500 Kb/s in this plot. All plots shown in the rest of this section are generated based on the data from experiments with the H.264 encoding standard.

For the PR-based approach, we have tested polynomials of degrees 0 to 10. While degree 7 gave the lowest MSE, the plots suggest a highly over-fitted model using this degree. Figure 5.5b shows the PR model (for bit rate 5500 Kb/s) using the following 1st-degree polynomial [39], which resulted in the best trade-off between the MSE value and the over-fitting that was visible in the graphical representations. $c(b, r, f)$ represents the number of vCPUs, given bit rate b, resolution r, and frame rate f. All coefficients in this and all following functions have been rounded to 2 decimal points. b is given as Kb/s, r as height of the video in pixels assuming a 16:9 aspect ratio, and f as frames/s. Moreover, we have divided the values of these parameters by powers of 10 in our experiments, such that all values are between 0 and 1, to avoid computational errors we were observing in our SVR models using the actual values. Black dots represent the measured data points.

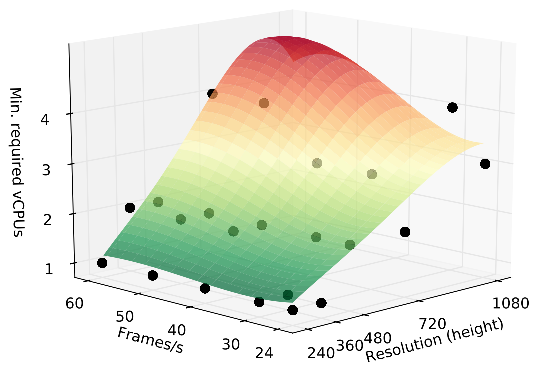
$$c(b, r, f) = 3.29 \cdot r + 1.77 \cdot f + 1.10 \cdot b - 0.96$$

Comparing the MSE and different plots of the SVR and PR models, the SVR-based approach gives better predictions for the minimum number of required vCPUs.

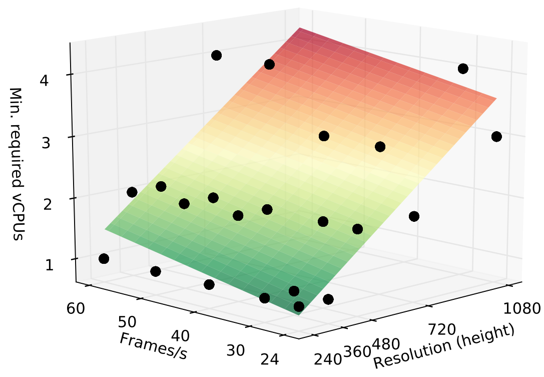
We have also developed additional SVR and PR models to predict the minimum number of required vCPUs c based on the target bit rate b, resolution r, or frame rate f, individually. The following equations [39] show the corresponding polynomials:

$$\begin{aligned} c(b) &= 15.12 \cdot b^5 - 50.31 \cdot b^4 + 62.12 \cdot b^3 - 35.20 \cdot b^2 \\ &\quad + 9.58 \cdot b + 0.10 \\ c(f) &= -0.017 \cdot f^2 + 0.93 \cdot f + 1.70 \\ c(r) &= -111.57 \cdot r^2 + 35.55 \cdot r + 0.73 \end{aligned}$$

Similar to the prediction models for the number of required vCPUs, we have developed SVR and PR models for predicting the maximum amount of memory used for reaching a target performance level in a given configuration among all videos.

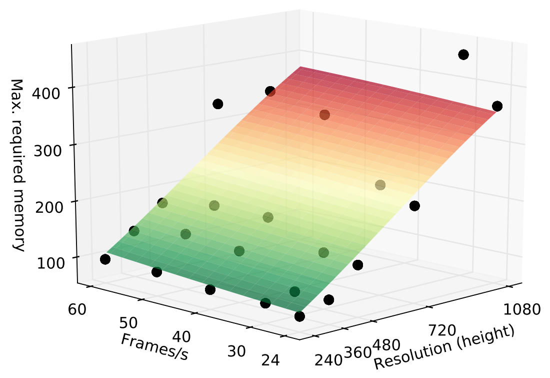


(a) SVR approach [39]

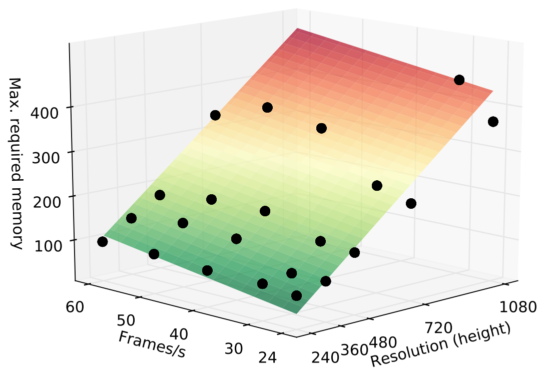


(b) PR approach [39]

Figure 5.5: Prediction of required vCPUs for the encoder based on bit rate, resolution, and frame rate, shown for bit rate of 5500 Kb/s.



(a) SVR approach [39]



(b) PR approach [39]

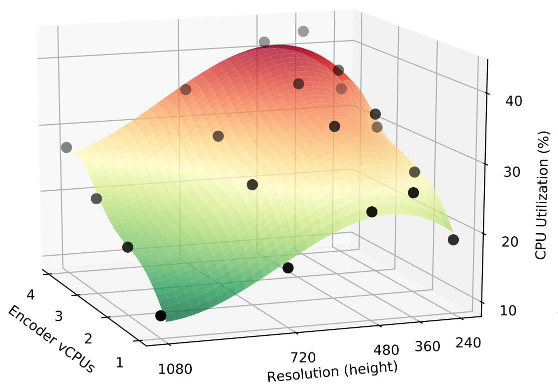
Figure 5.6: Prediction of required memory for the encoder based on bit rate, resolution, and frame rate, shown for bit rate of 5500 Kb/s.

As training data for the SVR-based prediction model, we have taken the memory used in test runs where the minimum number of vCPUs are used and no violation of the target values for bit rate, resolution, and frame rate has occurred.

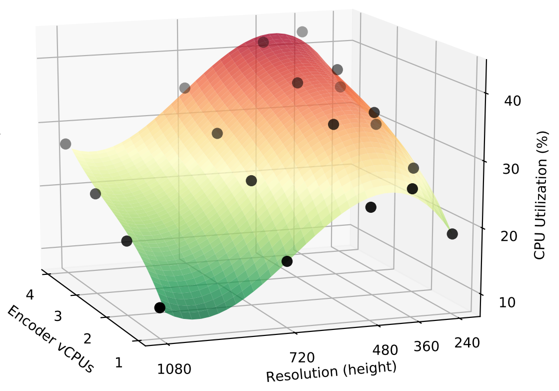
Figure 5.6a shows the SVR-based model, for bit rate of 5500 Kb/s, with model parameters that resulted in the lowest MSE. Figure 5.6b shows the PR-based model using a 1st-degree polynomial [39], which resulted in the best trade-off between the MSE and the over-fitting detectable in different plots. In the corresponding function, $m(b, r, f)$ represents the maximum required memory (in MB) to achieve a given bit rate b, resolution r, and frame rate f.

$$m(b, r, f) = 472.54 \cdot r + 196.77 \cdot f + 18.38 \cdot b - 130.51$$

To capture the inter-dependencies among the VNFs in our VCS, we have also developed models to predict the CPU utilization of the cache VNF based on the resolution and the number of vCPUs that are allocated to the encoder VNF.



(a) SVR approach [31]



(b) PR approach [31]

Figure 5.7: Prediction of CPU utilization of the cache based on resolution and the number of vCPU cores assigned to the encoder.

Figure 5.7a shows the SVR model and Figure 5.7a shows the PR-based model using a 3rd-degree polynomial. It can be observed that the CPU utilization of the cache is clearly influenced by the allocated CPU to the encoder.

These experimental results show the feasibility of characterizing the resource demands and performance metric values of service components, which is a requirement for the placement, scaling, and routing approach I propose in this dissertation for pliable VCSs.

These results clearly show that service components have to be profiled in the target VCS setup, in which they are planned to be executed. Only in this way, the resource utilization and performance dependencies among the service components can be captured and used for accurate prediction and resource planning models. In Chapter 9, I briefly describe the existing approaches to service profiling and their practical applicability.

Our analysis shows that these relationships are non-trivial even for simple functions, reinforcing the need for experimental data for benchmarking and further analysis. Our regression models required manual parameter tuning and checks based on MSE and visual representation of the results. For an automated process, more flexible approaches are required. These approaches are out of the scope of this dissertation. For specifying the resource demands of the pliable VCSs based on the input data rate, I assume there are existing profiling mechanisms that can provide the required performance and resource consumption models.

6 Embedding Uni-Directional Services with Load-Proportional Structures

In this chapter, I tackle the joint scaling, placement, routing problem for uni-directional pliable Virtualized Composed Services (VCSs) (U-SPRING). I describe the model and assumptions in Section 6.1. I present the problem formulation in Section 6.2 and the problem complexity in Section 6.3. In Section 6.4 and Section 6.5, I present the optimization and heuristic approaches to this problem, respectively, which I have developed in a joint work with Z. Á. Mann. I present the evaluation results of the approaches in Section 6.6. I conclude this chapter in Section 6.7. This chapter partially includes figures and verbatim copies of the text from my papers [26, 34].

Table 6.1: U-SPRING Substrate Network Parameters

Symbol	Definition
$G_{\text{sub}} = (V, L)$	Substrate network graph.
$v \in V$, $l \in L$	Substrate network nodes, links.
$\text{cap}_{\text{cpu}}(v)$, $\text{cap}_{\text{mem}}(v)$	CPU, memory capacity of v.
$\text{cap}(l)$, $d(l)$	Capacity, delay of l.

6.1 Model

This model consists of three different graphs for representing (i) the abstract structure of the pliable VCS, (ii) a concrete and deployable instantiation of the VCS, and (iii) the substrate network. I use different terms and notations to distinguish these graphs, described in the rest of this section. While the substrate network model is similar to the A-SPRING model (described in Section 3.2.1), there are differences between the A-SPRING model and the U-SPRING model in how the abstract structure of a pliable VCS and the corresponding deployable service graph are described. For example, in the U-SPRING model, the maximum number of instances for each service component or the maximum number of deployed VCSs that can reuse and share an instance are not specified.

6.1.1 Substrate network

I model the *substrate network* as a connected, directed graph $G_{\text{sub}} = (V, L)$. Each network *node* $v \in V$ has a limited CPU capacity $\text{cap}_{\text{cpu}}(v) \geq 0$ and a limited memory $\text{cap}_{\text{mem}}(v) \geq 0$. This can be easily extended to other types of resources, e.g., disk space, special-purpose compute capacity, etc. Moreover, I assume that every node has routing capabilities and can forward traffic to its neighboring nodes. CPU and memory capacities of the nodes can be 0, e.g., to represent conventional switches with no compute capabilities.

Each network *link* $l \in L$ supports a maximum data rate of $\text{cap}(l)$ and has a given delay of $d(l)$.

For each node v, I assume that the internal communications (e.g., the communication inside a data center) can be realized at unlimited data rate and negligible delay.

On top of such a substrate network, operated by a network operator, different services belonging to different service providers can be deployed and run.

Table 6.1 summarizes the network-related parameters used in the U-SPRING problem formulation.

6.1.2 Service Template

The substrate network has to host a set $\mathcal{T}$ of pliable VCSs. I define the structure of each VCS $T \in \mathcal{T}$ using a *service template*, which is a connected, directed,

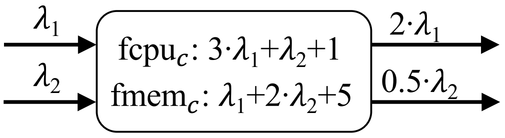

λ_1
λ_2
fcpu$_c$: $3\cdot\lambda_1+\lambda_2+1$
fmem$_c$: $\lambda_1+2\cdot\lambda_2+5$
$2\cdot\lambda_1$
$0.5\cdot\lambda_2$

Figure 6.1: An example component and its resource demands and outgoing data rate defined as functions of the data rates λ_1 and λ_2 on its two inputs

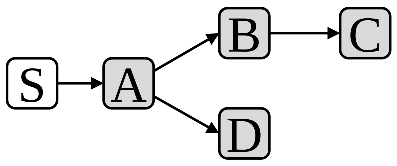


Figure 6.2: An example template consisting of a source and four other components

acyclic graph $G_T = (C_T, A_T)$. I refer to the nodes and edges of the template as *components* and *arcs*, respectively.

Each component $c \in C_T$ in the template represents a Virtual Network Function (VNF), cloud service component, etc. A component c has a given number of inputs n_c^{in} and outputs n_c^{out}, representing the number of ingoing and outgoing connection points, respectively. The outgoing data rate of a component depends on the data rate on all its inputs. This is calculated using a given function $\text{fout}_c(\Lambda) : \mathbb{R}_{\geq 0}^{n_c^{\text{in}}} \to \mathbb{R}_{\geq 0}^{n_c^{\text{out}}}$. Λ is the vector holding the data rates of all inputs. This function can be obtained, e.g., by referring to historical usage data or by testing and profiling the component.

Similarly, the CPU and memory demands of each component c can be calculated using pre-defined functions $\text{fcpu}_c(\Lambda) : \mathbb{R}_{\geq 0}^{n_c^{\text{in}}} \to \mathbb{R}_{\geq 0}$ and $\text{fmem}_c(\Lambda) : \mathbb{R}_{\geq 0}^{n_c^{\text{in}}} \to \mathbb{R}_{\geq 0}$, respectively.

Figure 6.1 shows examples for the functions that define the resource demands and output data rates of an example component.

Each *arc* $a \in A_T$ of the template connects an output of a component to an input of another component, representing the connectivity among them. Arcs may be described using additional details regarding the maximum tolerable delay or the underlying networking technology. They impose additional constraints on the links that can be used for realizing the connection between the two endpoints of it, i.e., the components src_a, dst_a. Adding these details to the model is straightforward but for simplicity, I do not consider these aspects in this model.

Figure 6.2 shows an example template.

Source components are special components in the template. They have no inputs, a single output with unspecified data rate, and zero resource consumption. In the example of Figure 6.2, S is a source component whereas the others are normal processing components.

Table 6.2 shows a summary of the parameters related to the service templates.

Table 6.2: U-SPRING Template Parameters

Symbol	Definition
$G_T{=}(C_T, A_T)$	Template graph.
$c{\in}C_T$, $a{\in}A_T$	Components and arcs of template T.
n_c^{in}, n_c^{out}	Number of inputs, outputs of c.
$\text{fcpu}_c(\Lambda)$, $\text{fmem}_c(\Lambda)$	CPU, memory demands of component c based on Λ, the vector of data rates on inputs of c.
$\text{fout}_c(\Lambda)$	Data rates on outputs of component c, calculated based on Λ, the vector of data rates on inputs of c.
src_a, dst_a	Component where arc a begins, ends.

6.1.3 Template Embedding

A template specifies the types of components and the connections among them as well as their resource demands depending on the load. A specific, deployable instantiation of a VCS can be derived by embedding the template in the substrate network. The *template embedding* process for uni-directional VCSs involves deciding:

- how many instances of each component (*horizontal scaling*),
- with how many resources (*vertical scaling*),
- need to be instantiated in which locations (*placement*),
- and how the traffic should be routed among them (*routing*).

The outcome of the template embedding process[1] is an overlay mapped to the substrate network, described in Section 6.1.4. In each template embedding process, multiple templates can be embedded. This process can be used for the initial embedding of templates as well as for updating existing embeddings.

To be able to create the required number of instances for each component, I assume either that the components are stateless or that a state management system is in place to handle state redistribution upon adding or removing instances. In this way, requests can be freely routed to any instance of a component. Alternatively, additional details can be added to the model, for example, to make sure that the flows belonging to a certain session are routed to the right instance of stateful components that have stored the corresponding state information.

For template embedding, a number of inputs are required. In addition to the templates to be embedded (including the description of their components and arcs), each template T must be accompanied by a set S_T of at least one *source instance.* Source instances are given as tuples $(c, v, \lambda) \in S_T$. $c \in C_T$ refers to the source component of template T and is used to differentiate between source

[1]Typically, a graph/virtual network/service embedding process maps a pre-defined overlay to a given substrate network. The *template* embedding process defined here, however, also includes the scaling decisions that shape the final structure of the overlay

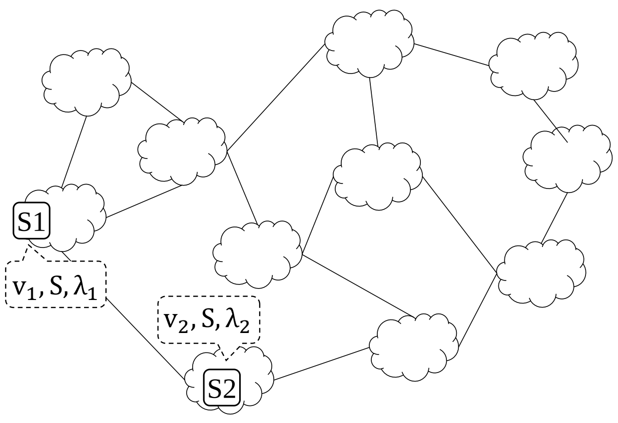


Figure 6.3: Example instances of the source component S (of the template in Figure 6.2), located on nodes v_1 and v_2 of an example substrate network, injecting flows with data rates λ_1 and λ_2 into the service

instances of different templates. $v \in V$ specifies the location in the substrate network where the flow initiates. λ shows the data rate of the flow. Figure 6.3 shows two example sources for the template of Figure 6.2, located on different nodes of the substrate network.

Another optional input is the set of previously existing instances of a template's components. This input is required if the template embedding is used for optimizing and updating an existing embedding. If a template is being embedded for the first time for a tenant or if the new service request is not going to reuse an already deployed shared instance of a service component, no previous embedding is required. Previous embeddings of the components of template T are given as a set P_T of tuples (c, v). Such a tuple specifies that an instance of component $c \in C_T$ exists on node $v \in V$.

Table 6.3 includes an overview of symbols and parameters related to the template embedding process for the U-SPRING problem.

6.1.4 Overlay

The template embedding process (Section 6.1.3) maps the abstract description of the service (service template) to a concrete deployable graph, i.e., the *overlay*, embedded into the substrate network. Each overlay is a connected, directed graph, $G_{\mathrm{OL}}(T) = (I_{\mathrm{OL}}(T), E_{\mathrm{OL}}(T))$. It consists of *instances* and *edges*.

Each overlay has exactly one template. One template can be embedded several times, e.g., each with different identifiers, belonging to different service providers.

For each instance $i \in I_{\mathrm{OL}}(T)$ in the overlay of template T, there exists a component $c \in C_T$ that contains its specification, i.e., inputs, outputs, resource consumption characteristics. Instances are mapped to network nodes and have resources allocated to them. For each component, there can be multiple instances. I make the simplifying assumption that two instances of the same component cannot be mapped to the same node. The rationale behind this assumption is that in this case it would be more efficient to replace the two instances by a single instance

Table 6.3: U-SPRING Template Embedding and Overlay Parameters

Symbol	Definition
$(c, v, \lambda) \in S_T$	Data rate λ of source component c from template T at node v.
$(c, v) \in P_T$	An existing instance of component c with deployment version ver previously embedded at node v.
$\mathcal{T}$	All templates to be embedded.
$\mathcal{C} = \bigcup_{T \in \mathcal{T}} C_T$	All components from templates in $\mathcal{T}$.
$\mathcal{C}_{\text{SRC}} \subset \mathcal{C}$	All source components.
$\mathcal{A} = \bigcup_{T \in \mathcal{T}} A_T$	All arcs of templates in $\mathcal{T}$.
$\mathcal{S} = \bigcup_{T \in \mathcal{T}} S_T$	All sources of templates in $\mathcal{T}$.
$G_{\text{OL}}(T) = (I_{\text{OL}}(T), E_{\text{OL}}(T))$	Overlay graph corresponding to template T.
$i \in I_{\text{OL}}(T)$, $e \in E_{\text{OL}}(T)$	Instances, edges of overlay.
$M_T^C(i)$	The corresponding component of instance i.
$M_T^V(i)$	The node where instance i is mapped to.
$M_T^A(e)$	The corresponding arc of edge e.

and thus save the idle resource consumption of one instance. This is mostly a technicality to simplify the formulation of the optimization problem described in Section 6.4. If one template is embedded multiple times, e.g., each for a different service provider, specified with different identifiers in each template, there are no limitations for embedding multiple instances of the same component into the same node as they are considered different instances in this model.

For each edge $e \in E_{\text{OL}}(T)$ in the overlay of template T, there exists an arc $a \in A_T$ that specifies its endpoints. Each edge e is mapped to a path in the substrate network. This path is a set of network links that starts at the network node to which src_a is mapped and connects it to the node to which dst_a is mapped. Paths must not include loops. I assume the service flows are splittable, i.e., can be routed over multiple paths between the corresponding endpoints in the substrate network.

Figure 6.4 shows an example overlay corresponding to the template in Figure 6.2.

The name of the instances in the figure follows the convention that the first letter identifies the corresponding component in the template, e.g., A1 is an instance of component A. An overlay might include multiple instances of a specific component, e.g., B1, B2, and B3 all are instances of component B.

An output of an instance can be connected to the input of multiple instances of the same component, like the output of A1 is connected to the inputs of B1 and B2. In a case like that, B1 and B2 share the data rate calculated for the connection between components A and B. Similarly, outputs of multiple instances in the overlay can be connected to the input of the same instance, like the input of C1 is connected to the output of B1, B2, and B3. In this case, the input data

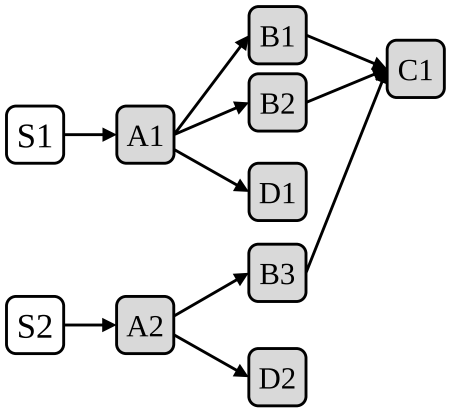


Figure 6.4: Example overlay resulting from scaling the template in Figure 6.2

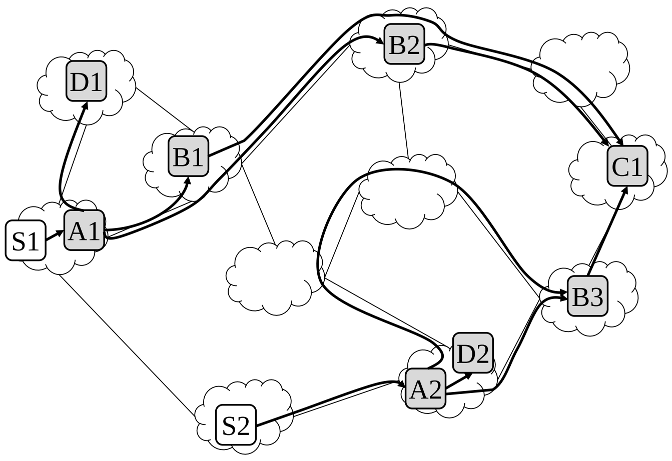


Figure 6.5: Overlay of the template from Figure 6.2 mapped into an example substrate network according to its sources

rate for C1 is the sum of the output data rates of B1, B2, and B3.

Figure 6.5 shows a possible mapping of the overlay of Figure 6.4 to an example substrate network, based on the pre-defined locations of S1 and S2 in the network. It is possible to map two communicating instances to the same node, like A2 and D2 in the example. In this case, the edge between them can be realized inside the node, without using any links. The flow between A2 and B3 is an example of a split flow that is routed over two different paths in the substrate network.

Figure 6.5 shows only a single overlay mapped to the substrate network for the sake of clarity. In general, U-SPRING deals with creating several overlays corresponding to different pliable VCSs into a substrate network.

6.2 Problem Formulation

The U-SPRING problem decides the scaling, placement, and routing for newly requested VCSs as well as already deployed ones. The inputs and outputs of the U-SPRING problem can be summarized as follows:

- Inputs:
  - Substrate network
  - A template for each VCS
  - Location and data rate of the sources for each VCS
  - Location of previously embedded components (optional, can be empty)
- Outputs:
  - For the newly requested VCSs: overlay and its mapping onto the substrate network
  - For the already deployed VCSs: modified overlay and its modified mapping onto the substrate network

A solution to the U-SPRING problem is a *system configuration* for the network of an operator, which consists of the overlays of the VCSs from different service providers and their mapping on the substrate network[2]. *Scaling* is performed while creating the overlay from the template, while *placement* and *routing* are performed when the instances and edges of the overlay are mapped onto the substrate network. These steps are integrated and are performed jointly in the U-SPRING approaches that I present in this chapter.

Ideally, every system configuration must respect all capacity constraints: for each node v, the total resource demands of the instances mapped to v must be within its capacity, concerning both CPU and memory. For each link l, the sum of the flow data rates going through l must be within its maximum data rate. It is, however, possible that some of those constraints are violated in a given system configuration: for example, a system configuration without any violations may become invalid because the data rate of a source has increased, as a result of a temporary peak in resource needs or a failure in the substrate network. Moreover, over-subscribing resources is a common practice, e.g., in the cloud computing context. I allow the violation of the node and link capacity constraints in this problem formulation to give an additional degree of freedom to the solution approaches, which, for example, could reduce the chance of a new service deployment request being rejected because of a temporary peak in the load of an existing service. It is straightforward to put a hard limit or to forbid over-subscribing the resources, if necessary for a specific scenario.

Given a current system configuration σ, the primary objective is to find a new system configuration σ', in which the *number of constraint violations is minimal* (ideally, zero). For this, I assume that violating the node (CPU, memory) and link capacity constraints are equally undesirable.

[2] A similar definition can also be used to formalize the A-SPRING problem and its objectives. However, as I only repeat the initial A-SPRING model definition from my previous research in Section 3.2 for completeness and without major modifications, it does not include such a definition. In the next chapters, I define the corresponding system configurations for the B-SPRING and M-SPRING approaches.

There are a number of further, secondary objectives, which can be used as tie-breaker to choose from system configurations that have the same number of constraint violations. For this, I define the following metrics of interest:

- Total delay of all edges across all overlays
- Number of instance addition/removal operations required to transition from σ to σ'
- Maximum amounts of capacity constraint violations, for each resource type (CPU, memory, link capacity)
- Total resource consumption of all instances across all overlays, for each resource type (CPU, memory, link capacity)

Higher values for these metrics result in higher costs for the system or in lower satisfaction of service providers and their users. Therefore, the objective is to minimize these values by selecting a new system configuration σ' from the set of system configurations with minimal number of constraint violations that is Pareto-optimal with respect to these secondary metrics.

The creation of the overlay from the template and its mapping onto the substrate network are defined for each VCS separately. The overlays, however, share the same substrate network. The objectives defined in this section apply to the whole network including all VCSs, aiming for a global optimum and potentially resulting in trade-offs among the requirements of different VCSs.

6.3 Problem Complexity

In joint work with Z. Á. Mann and H. Karl [34], we have shown that for an instance of the U-SPRING problem as defined in Section 6.2, deciding whether a solution with no violations exists is NP-complete in the strong sense. This means that the problem remains NP-complete even if the numbers appearing in it are constrained between polynomial bounds. Because of the complexity of the problem, we can neither expect a polynomial (or even pseudo-polynomial) algorithm for solving the problem exactly nor a fully polynomial-time approximation scheme, under standard assumptions of complexity theory.

6.4 Optimization Approach

In this section, I describe a Mixed-Integer Program (MIP) formulation of the U-SPRING problem. Table 6.1, 6.2, and 6.3 show an overview the input parameters to the MIP.

All of the problem constraints (described in Section 6.4.1) are linear equations and linear inequalities. The objective function (described in Section 6.4.2) is also linear. Therefore, if the functions fcpu_c, fmem_c, and fout_c are linear for all $c \in \mathcal{C}$, then we obtain a Mixed-Integer Linear Program (MILP), which can be solved by appropriate solvers. For non-linear functions, a piecewise linear approximation may make it possible to use MILP solvers to find good (although not necessarily optimal) solutions.

Table 6.4 shows an overview of the decision variables used in the MIP.

In the problem formulation, $\mathcal{M}$, $\mathcal{M}_1$, and $\mathcal{M}_2$ denote sufficiently large constants, used in the so-called big-M constraints. $(W)_k$ denotes the k-th component of a vector W. $\underline{0}$ denotes a zero vector of appropriate length.

The information about existing instances from previous embeddings of the VCSs should also be taken into account during the decision process. For this, we have defined $x^*_{c,v}$ as a constant given as part of the problem input:

$$\forall (c,v) \in P_T : x^*_{c,v} = 1$$
$$\forall c \in \mathcal{C}, \forall v \in V, \text{if } (c,v) \notin P_T : x^*_{c,v} = 0$$

6.4.1 Constraints

In this section, I describe the constraints of the MIP that enforce the required properties of the template embedding process.

Together, the constraints ensure that for every service template, at least one instance of every component is mapped optimally to a network node that has enough capacity. Additionally, the flows starting from the source components of each template are mapped optimally to paths over network links with enough capacity. The flows traverse instances of all relevant components as defined in the template. I describe the optimization objective that drives the mapping decisions based on these constraints in Section 6.4.2.

Mapping Consistency Rules

$$\forall (c,v,\lambda) \in \mathcal{S} : \quad x_{c,v} = 1 \tag{6.1}$$
$$\forall (c,v,\lambda) \in \mathcal{S} : \quad \text{out}_{c,v} = \lambda \tag{6.2}$$
$$\forall c \in \mathcal{C}, \forall v \in V, k \in [1, n^{\text{in}}_c] : \quad (\text{in}_{c,v})_k \le \mathcal{M} \cdot x_{c,v} \tag{6.3}$$
$$\forall c \in \mathcal{C}, \forall v \in V, k \in [1, n^{\text{out}}_c] : \quad (\text{out}_{c,v})_k \le \mathcal{M} \cdot x_{c,v} \tag{6.4}$$
$$\forall c \in \mathcal{C}, \forall v \in V : \quad x_{c,v} - x^*_{c,v} \le \delta_{c,v} \tag{6.5}$$
$$\forall c \in \mathcal{C}, \forall v \in V : \quad x^*_{c,v} - x_{c,v} \le \delta_{c,v} \tag{6.6}$$

Constraints 6.1 and 6.2 enforce that the placement and the output data rate of source component instances are in line with the tuples specified in $\mathcal{S}$, respectively. Constraint 6.3 guarantees the consistency between the variables $\text{in}_{c,v}$ and $x_{c,v}$: if $\text{in}_{c,v}$ indicated a positive data rate on an input of instance c, then $x_{c,v}$ must be 1, i.e., only an instance mapped to a node can process incoming flows. Constraint 6.4 is analogous for the outgoing flows, represented by the $\text{out}_{c,v}$ variables. Constraints 6.5 and 6.6 together ensure that $\delta_{c,v} = 1$ if and only if $x_{c,v} \neq x^*_{c,v}$.

Flow and Data Rate Rules

$$\forall c \in \mathcal{C}, c \text{ not a source component}, \forall v \in V : \\ \text{out}_{c,v} = \text{fout}_c(\text{in}_{c,v}) - (1 - x_{c,v}) \cdot \text{fout}_c(\underline{0}) \quad (6.7)$$

$$\forall c \in \mathcal{C}, \forall v \in V, k \in [1, n_c^{\text{in}}] : \\ (\text{in}_{c,v})_k = \sum_{a \text{ ends in input } k \text{ of } c, v' \in V} y_{a,v',v} \quad (6.8)$$

$$\forall c \in \mathcal{C}, \forall v \in V, k \in [1, n_c^{\text{out}}] : \\ (\text{out}_{c,v})_k = \sum_{a \text{ starts in output } k \text{ of } c, v' \in V} y_{a,v,v'} \quad (6.9)$$

$$\forall a \in \mathcal{A}, \forall v, v_1, v_2 \in V : \\ \sum_{vv' \in L} z_{a,v_1,v_2,vv'} - \sum_{v'v \in L} z_{a,v_1,v_2,v'v} = \\ = \begin{cases} 0 & \text{if } v \neq v_1 \text{ and } v \neq v_2 \\ y_{a,v_1,v_2} & \text{if } v = v_1 \text{ and } v_1 \neq v_2 \\ 0 & \text{if } v = v_1 = v_2 \end{cases} \quad (6.10)$$

$$\forall a \in \mathcal{A}, \forall v, v' \in V, \forall l \in L : \qquad z_{a,v,v',l} \leq \mathcal{M} \cdot \zeta_{a,v,v',l} \quad (6.11)$$

Constraint 6.7 computes the data rate on the outputs of an instance based on the data rates on its inputs and the fout_c function of the corresponding component. The constraint is formulated in such a way that for $x_{c,v} = 1$, $\text{out}_{c,v} = \text{fout}_c(\text{in}_{c,v})$, whereas for $x_{c,v} = 0$ (in which case also $\text{in}_{c,v} = 0$ because of Constraint 6.3, also $\text{out}_{c,v} = 0$ so that there is no contradiction with Constraint 6.4. Constraint 6.8 computes the data rate on the inputs of an instance as the sum of the data rates on the links ending in that input. Similarly, Constraint 6.9 ensures that the data rate on the outputs of an instance is distributed over the links starting in that output. Constraint 6.10 is the flow conservation rule, also ensuring the right data rate of each flow, thus relating the $z_{a,v,v',l}$ variables (flow data rate on individual links) and the $y_{a,v,v'}$ variables (flow data rates). Constraint 6.11 sets the $\zeta_{a,v,v',l}$ variables (based on the $z_{a,v,v',l}$ variables), so that they can be used in the objective function (Section 6.4.2).

Calculation of Resource Consumption

$$\forall c \in \mathcal{C}, \forall v \in V : \qquad \text{cpu}_{c,v} = \text{fcpu}_c(\text{in}_{c,v}) - (1 - x_{c,v}) \cdot \text{fcpu}_c(\underline{0}) \quad (6.12)$$

$$\forall c \in \mathcal{C}, \forall v \in V : \qquad \text{mem}_{c,v} = \text{fmem}_c(\text{in}_{c,v}) - (1 - x_{c,v}) \cdot \text{fmem}_c(\underline{0}) \quad (6.13)$$

Constraints 6.12 and 6.13 calculate the CPU and memory demands of each instance based on the fcpu_c and fmem_c functions of the corresponding component. The logic here is analogous to that of Constraint 6.7.

Capacity Constraints

$$\forall v \in V: \quad \sum_{c \in \mathcal{C}} \mathrm{cpu}_{c,v} \leq \mathrm{cap}_{\mathrm{cpu}}(v) + \mathcal{M} \cdot \omega_{v,\mathrm{cpu}} \tag{6.14}$$

$$\forall v \in V: \quad \sum_{c \in \mathcal{C}} \mathrm{cpu}_{c,v} - \mathrm{cap}_{\mathrm{cpu}}(v) \leq \psi_{\mathrm{cpu}} \tag{6.15}$$

$$\forall v \in V: \quad \sum_{c \in \mathcal{C}} \mathrm{mem}_{c,v} \leq \mathrm{cap}_{\mathrm{mem}}(v) + \mathcal{M} \cdot \omega_{v,\mathrm{mem}} \tag{6.16}$$

$$\forall v \in V: \quad \sum_{c \in \mathcal{C}} \mathrm{mem}_{c,v} - \mathrm{cap}_{\mathrm{mem}}(v) \leq \psi_{\mathrm{mem}} \tag{6.17}$$

$$\forall l \in L: \quad \sum_{a \in \mathcal{A}; v,v' \in V} z_{a,v,v',l} \leq \mathrm{cap}(l) + \mathcal{M} \cdot \omega_l \tag{6.18}$$

$$\forall l \in L: \quad \sum_{a \in \mathcal{A}; v,v' \in V} z_{a,v,v',l} - \mathrm{cap}(l) \leq \psi_{\mathrm{dr}} \tag{6.19}$$

The aim of these constraints is to set the ω and ψ variables (based on the already defined cpu, mem and z variables), which are used in the objective function (Section 6.4.2). Constraint 6.14 ensures that $\omega_{v,\mathrm{cpu}}$ is 1 if the CPU capacity of node v is over-subscribed, while Constraint 6.15 ensures that ψ_{cpu} is at least as high as the amount of CPU over-subscription of any node (the appearance of ψ_{cpu} in the objective function guarantees that it is exactly the maximum amount of CPU over-subscription and not higher than that). Constraints 6.16 and 6.17 do the same for memory over-subscription and Constraints 6.18 and 6.19 do the same for the over-subscription of link capacity.

To show the interplay of the constraint, assume that the embedding shown in Figure 6.5 needs to be optimized. Constraints 6.1 and 6.2 ensure that instances of the source component, i.e., S1 and S2, are embedded and their output data rates are set correctly. Constraint 6.9 ensures that these data rates are then handed out as flows that can only end up in instances of A. These flows are mapped to network links and instances of A are assigned input data rates using Constraints 6.10 and 6.8, respectively. Constraint 6.3 marks the instances A1 and A2 as embedded, and Constraint 6.7 sets their output data rates using the respective fout_c function. In a similar way, the rest of the components are instantiated and embedded in the network.

Constraints 6.5 and 6.6 ensure that the $\delta_{c,v}$ variables are set correctly. Constraints 6.12 and 6.13 compute the resource consumption of each instance based on the input data rates and the corresponding fcpu_c and fmem_c functions. Constraints 6.14–6.19 make sure that the over-subscription of node and link capacities are captured correctly, and collect the maximum value of over-subscription for each resource type. This maximum value is used in the objective function described in Section 6.4.2, which drives the decisions based on the constraints.

6.4.2 Optimization Objective

We have formalized the optimization objective based on the goals defined in Section 6.2 as follows:

$$\begin{aligned}\text{minimize}\quad & \mathcal{M}_1 \cdot \Big(\sum_{v \in V} (\omega_{v,\text{cpu}} + \omega_{v,\text{mem}}) + \sum_{l \in L} \omega_l \Big) + \\ & + \mathcal{M}_2 \cdot \Big(\sum_{\substack{a \in \mathcal{A} \\ v,v' \in V \\ l \in L}} (d(l) \cdot \zeta_{a,v,v',l}) + \sum_{\substack{c \in \mathcal{C} \\ v \in V}} \delta_{c,v} \Big) + \\ & + \psi_{\text{cpu}} + \psi_{\text{mem}} + \psi_{\text{dr}} + \sum_{\substack{c \in \mathcal{C} \\ v \in V}} (\text{cpu}_{c,v} + \text{mem}_{c,v}) + \sum_{\substack{a \in \mathcal{A} \\ v,v' \in V \\ l \in L}} z_{a,v,v',l} \end{aligned} \tag{6.20}$$

By assigning sufficiently large values to $\mathcal{M}_1$ and $\mathcal{M}_2$, we can achieve the following goals with the given priorities:

1. The number of capacity constraint violations over all nodes and links is minimized.
2. Template arcs are mapped to network paths in a way that their total latency is minimized. Moreover, the number of instances that need to be added or removed is minimized.
3. The maximum value for capacity constraint violations over all nodes and links is minimized. Also, overlay instances and the edges among them are created in a way that their resource consumption is minimized.

This mixed-integer program can be used for initial embedding of service templates as well as for optimizing existing embeddings. For the initial embedding of newly requested network services, the term $\sum_{c \in \mathcal{C}, v \in V} \delta_{c,v}$ could be removed from the objective function. Using this term, embeddings with fewer instances will be preferred, even if having more instances, for example, close to different user locations would improve the value of another metric like total delay.

Table 6.4: U-SPRING Decision Variables

Name	Domain	Definition
$x_{c,v}$	$\{0,1\}$	1 iff an instance of component $c \in \mathcal{C}$ is mapped to node $v \in V$
$y_{a,v,v'}$	$\mathbb{R}_{\geq 0}$	If $a \in A_T$ is an arc from an output of $c \in C_T$ to an input of $c' \in C_T$, an instance of c is mapped to $v \in V$, and an instance of c' is mapped to $v' \in V$, then $y_{a,v,v'}$ is the data rate of the corresponding flow from v to v'; otherwise it is 0
$z_{a,v,v',l}$	$\mathbb{R}_{\geq 0}$	If $a \in A_T$ is an arc from an output of $c \in C_T$ to an input of $c' \in C_T$, an instance of c is mapped to $v \in V$, and an instance of c' is mapped to $v' \in V$, then $z_{a,v,v',l}$ is the data rate of the corresponding flow from v to v' that goes through link $l \in L$; otherwise it is 0
$\text{in}_{c,v}$	$\mathbb{R}_{\geq 0}^{n_c^{\text{in}}}$	Vector of data rates on the inputs of the instance of component $c \in C_T$ on node $v \in V$, or an all-zero vector if no such instance is mapped to v
$\text{out}_{c,v}$	$\mathbb{R}_{\geq 0}^{n_c^{\text{out}}}$	Vector of data rates on the outputs of the instance of component $c \in C_T$ on node $v \in V$, or an all-zero vector if no such instance is mapped to v
$\text{cpu}_{c,v}$	$\mathbb{R}_{\geq 0}$	CPU requirement of the instance of component $c \in C_T$ on node $v \in V$, or zero if no such instance is mapped to v
$\text{mem}_{c,v}$	$\mathbb{R}_{\geq 0}$	Memory requirement of the instance of component $c \in C_T$ on node $v \in V$, or zero if no such instance is mapped to v
$\omega_{v,\text{cpu}}$	$\{0,1\}$	1 iff the CPU demands of the instances mapped to node $v \in V$ exceed the CPU capacity of the node
$\omega_{v,\text{mem}}$	$\{0,1\}$	1 iff the memory demands of the instances mapped to to node $v \in V$ exceeded the memory capacity of the node
ω_l	$\{0,1\}$	1 iff the data rate of the flows mapped to link $l \in L$ exceeded the maximum data rate of the link
ψ_{cpu}	$\mathbb{R}_{\geq 0}$	Maximum CPU over-subscription over all nodes
ψ_{mem}	$\mathbb{R}_{\geq 0}$	Maximum memory over-subscription over all nodes
ψ_{dr}	$\mathbb{R}_{\geq 0}$	Maximum capacity over-subscription over all links
$\zeta_{a,v,v',l}$	$\{0,1\}$	1 iff $z_{a,v,v',l} > 0$
$\delta_{c,v}$	$\{0,1\}$	1 iff $x_{c,v} \neq x^*_{c,v}$

6.5 Heuristic Approach

In this section, I present a heuristic, which is not guaranteed to find an optimal solution but is much faster than the optimization approach. The heuristic can construct new embeddings from existing ones by means of a series of small local changes. The embedding of new templates is done in a similar way, creating component instances one by one. While doing so, we ensure that (i) sources are created in the pre-defined locations, (ii) the data rate produced by each instance is forwarded to the instances according to the template, and (iii) the capacity constraints of the nodes and the links are respected as much as possible. We do this by iterating through the instances of each overlay once in a *topological order*, possibly creating new instances if necessary; for example, when a new data source appears or the output data rate of a source increases. A topological order of vertices in a directed acyclic graph $G = (V, E)$ is a linear order where every vertex $v \in V$ is traversed before vertex $v' \in V$ if there is an edge $(v, v') \in E$. When traversing the instances of the overlay in a topological order it is ensured that all incoming edges of an instance i have an updated, correct data rate as all other instances with an edge towards i have already been processed. In each step, the algorithm aims to keep the resource consumption small, e.g., by only creating new instances if necessary, deleting unneeded instances, or preferring short paths.

Table 6.1, 6.2, and 6.3 summarize the required inputs for the heuristic approach.

The main workflow of the heuristic approach is shown in Algorithm 6.1. It starts by checking that each template has a corresponding overlay and each overlay corresponds to a template (lines 2–6). If a new VCS has been requested or an existing VCS has been stopped since the last embedding, the corresponding overlay is created or removed at this point.

Afterwards, the mapping of the sources and source components is checked and updated as follows (lines 7–12). If a new source has been added for the template, an instance of the corresponding source component is created; if the data rate of a source has changed, the output data rate of the corresponding source instance is updated; if a source has disappeared, the corresponding source instance is removed.

To propagate the changes from the sources to the rest of the instances, the heuristic iterates over all instances and ensures that the new output data rates, which are determined by the new input data rates, are passed on correctly over the corresponding outputs (lines 13–24).

If the outgoing data rate corresponding to an arc a from an output k of instance i needs to be decreased (line 22), the algorithm tries to reduce the data rate of the edges with the lowest data rate first. In this way, some edges might be removed. If no more edges exist that can be removed for reaching the required lower data rate, the data rate of all of the remaining edges from output k of i that belong to arc a is decreased by a factor of its current data rate until the required data rate is reached. If an edge e has a current data rate of λ and the remaining data rate to be decreased (after removing some edges) is $\Delta\lambda$, its data rate is decreased by $(\lambda - \Delta\lambda)/\lambda$. In this way, we limit the number of required changes to be propagated to the rest of the overlay instances, as far as possible.

Algorithm 6.1 Main procedure of the U-SPRING heuristic

```
1:  // Remove old overlays with no templates
2:  if $\exists G_{\text{OL}}(T)$ with $T \notin \mathcal{T}$ then
3:      remove $G_{\text{OL}}(T)$
4:  for all $T \in \mathcal{T}$ do
5:      if $\nexists G_{\text{OL}}(T)$ then
6:          create empty overlay $G_{\text{OL}}(T)$
7:      for all $(c, v, \lambda) \in S_T$ do
8:          if $\nexists i \in I_{\text{OL}}(T)$ with $M_T^C(i) = c$ and $M_T^V(i) = v$ then
9:              create $i \in I_{\text{OL}}(T)$ with $M_T^C(i) = c$ and $M_T^V(i) = v$
10:         set output data rate of $i$ to $\lambda$
11:     if $\exists i \in I_{\text{OL}}(T)$, $M_T^C(i) \in \mathcal{C}_{\text{SRC}}$, $\nexists(M_T^C(i), M_T^V(i), \lambda) \in S_T$ for any $\lambda$ then
12:         remove $i$
13:     for all $i \in I_{\text{OL}}(T)$ in topological order do
14:         if data rates on inputs of $i$ are 0 then
15:             remove $i$ and go to next iteration
16:         compute output data rates of $i$
17:         for all output $k$ of $i$ do
18:             $\lambda$: sum of the data rates leaving output $k$
19:             $\lambda'$: new data rate on output $k$
20:             if $\lambda' < \lambda$ then
21:                 $\mathcal{E}$: set of edges leaving output $k$
22:                 decrease the data rate over $\mathcal{E}$ by $\lambda - \lambda'$
23:             else if $\lambda' > \lambda$ then
24:                 increase the data rate leaving output $k$ by $\lambda' - \lambda$
```

For increasing the data rate that leaves the current instance i (line 24), the algorithm first checks if new instances need to be created to be consistent with the template. This is needed in case the corresponding component of i has an arc to a component c' but there are no existing instances of c' in the template. For this, all nodes of the substrate network are considered for hosting the new instance. The candidate node that can host the instance with the highest possible data rate is selected. The data rate that can be forwarded to an instance is limited by the CPU and memory capacity of the hosting node. If there are multiple such nodes, the one that can be reached using a path with a lower delay from the node where instance i is located is preferred. For finding the paths, we have used a modified best-first-search [108], which runs in linear time. During the search, the nodes to be visited are stored in a priority queue, where priority is defined as described, based on the capacity of the node that determines the possible data rate, as well as the delay of the possible path towards it.

If there are existing instances of c', before creating additional instances to accommodate the increased data rate, the algorithm first tries to increase the data rate along the existing edges up to the required level. If this is not sufficient to achieve the necessary increase, it creates further instances and edges towards

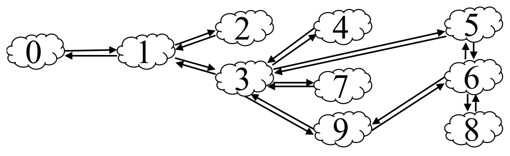


Figure 6.6: Example substrate network

them.

As the instances of each overlay are processed sequentially, the results created by the heuristic are not optimal. Possible optimization steps can be employed after the main procedure of the algorithm, to improve the initial solution. This can be done, for example, by comparing the results of applying some of the following changes and switching to a better embedding if possible:

- Replicating an instance i on another node and distributing the incoming data rates to i over both of them,
- Merging two instances of a component c,
- Re-locating an instance i in the network,
- Changing the path taken by an edge e.

6.6 Evaluation

In this section, I show our evaluation results of the optimization and heuristic approaches to the U-SPRING problem. Both approaches were implemented as C++ programs. The implementation of the approaches is available online [37]. For solving the MILP, we have used the Gurobi Optimizer [21] 7.0.1. The service templates used in these evaluations are inspired by the examples from Internet Engineering Task Force (IETF) Service Function Chaining (SFC) Use Cases [93].

First, I illustrate the optimization approach on a small substrate network with 10 nodes and 20 arcs (shown in Figure 6.6) in which the CPU and memory capacity of each node is set to 100 units. In this network, a VCS consisting of a source (S), a firewall (FW), a deep packet inspection (DPI) component, an anti-virus (AV) component, and a parental control (PC) component is embedded. Initially, there is a single source in node 1 with a moderate data rate. Using the optimization approach, all components of the VCS are mapped to node 1, as shown in Figure 6.7a.

If the data rate of the source increases, the resource demand of the processing components of the VCS increases so that they do not fit onto node 1 anymore. Given the previous embedding as an input to the optimization approach, it scales the VCS by duplicating the DPI, AV, and PC components and places the newly created instances on a nearby node, namely node 3, as shown in Figure 6.7b.

Later on, a second source emerges for the same VCS on node 9. The algorithm creates new instances of the components on node 9 to process the traffic of the new

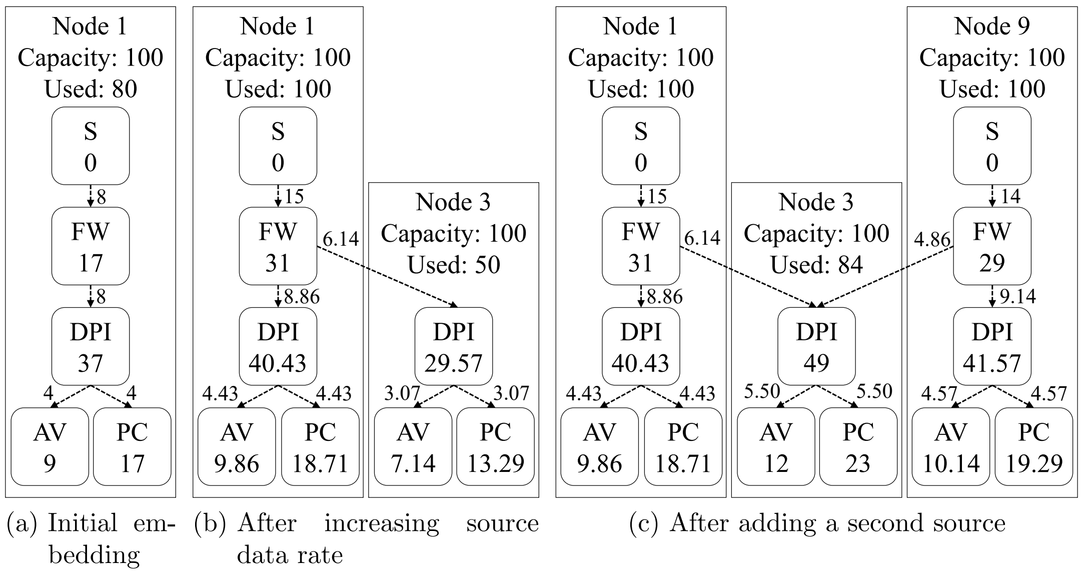


Figure 6.7: Template embedding example (memory values not shown for better readability)

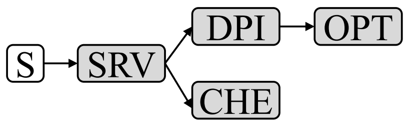


Figure 6.8: Service template with source component (S), streaming server (SRV), deep packet inspector (DPI), video optimizer (OPT), and cache (CHE)

source locally, as far as possible. The excess traffic from the new FW instance that cannot be processed locally due to capacity constraints is routed to the existing DPI, AV, and PC instances on node 3 because node 3 still has sufficient free capacity. The new embedding is shown in Figure 6.7c.

This small example shows the trade-offs that the template embedding process needs to consider. In the rest of this section, I show that the U-SPRING approaches are also capable of handling much more complex scenarios. For larger substrate networks, we have used the benchmarks for the Virtual Network Mapping Problem (VNMP) [109] from Inführ and Raidl [110].

6.6.1 Comparison of Optimization and Heuristic Approaches

In this section, we have used a substrate network with 20 nodes and 44 links (substrate graph eu_20_0_prob [109]), in which multiple VCSs are deployed. Each service is a virtual Content Delivery Network (CDN) for video streaming, consisting of a streaming server, a DPI, a video optimizer, and a cache. The service template is shown in Figure 6.8. The number of concurrently active VCSs varies from 0 to 4 and the number of sources from 0 to 20.

Figure 6.9 shows how the total data rate of the sources (as a metric representing the demand) and the total CPU size of the created instances (as a metric rep-

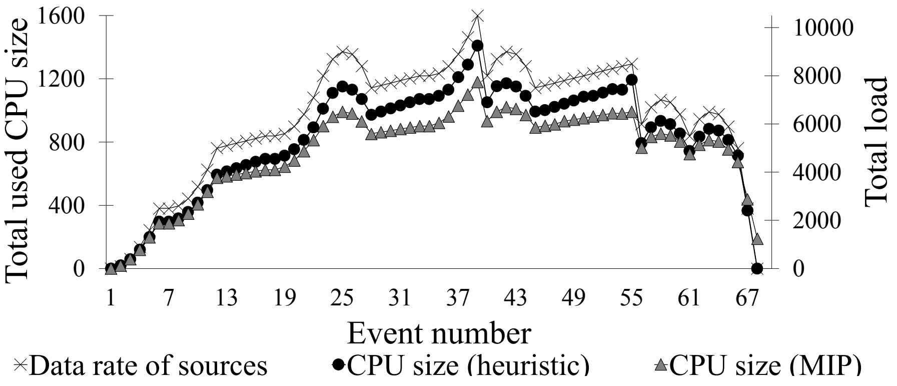


Figure 6.9: Temporal development of the demand and the allocated capacity in a complex scenario

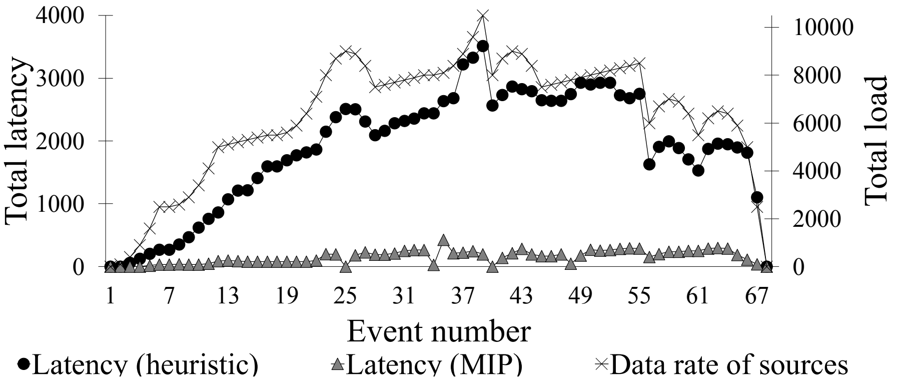
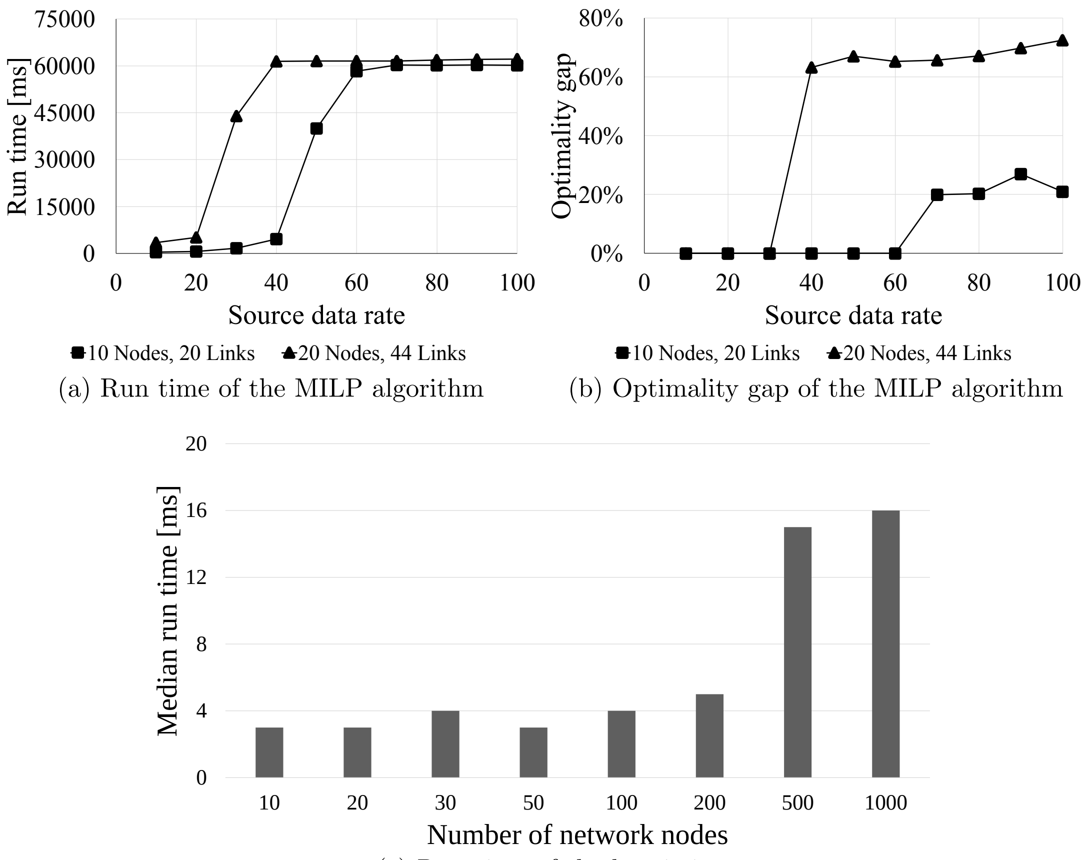


Figure 6.10: Total latency over all created paths for the embedded template

resenting the allocated processing capacity) change through re-optimization after each *event*. An event is the emergence or ceasing of a VCS, the emergence or ceasing of a source, or changes in the data rate of a source. The resource allocation using both the heuristic and the optimization approaches follows the demand very closely, meaning that the algorithms are successful in scaling the templates in both directions to quickly react to the increase and decrease in the total data rate.

Regarding total data rate and total latency of the overlay edges, the optimization approach performs better than the heuristic. For example, Figure 6.10 shows the total latency over all paths created for the template in this scenario. The reason for this difference is that in the optimization approach, the optimal location for all required instances can be determined at the same time. This results in shorter distances between the source and the instances. The heuristic, on the other hand, creates instances one by one, resulting in larger data rates over larger distances in the substrate network. In the high-load area between event 20 and 50, some problem instances are too complex to be solved within the 60 seconds time limit that we had set for the optimizer. This results in solutions with zero latency, as no paths are created.

In this scenario, to handle the peak demand, a total of 127 instances are created



(a) Run time of the MILP algorithm

(b) Optimality gap of the MILP algorithm

(c) Run time of the heuristic

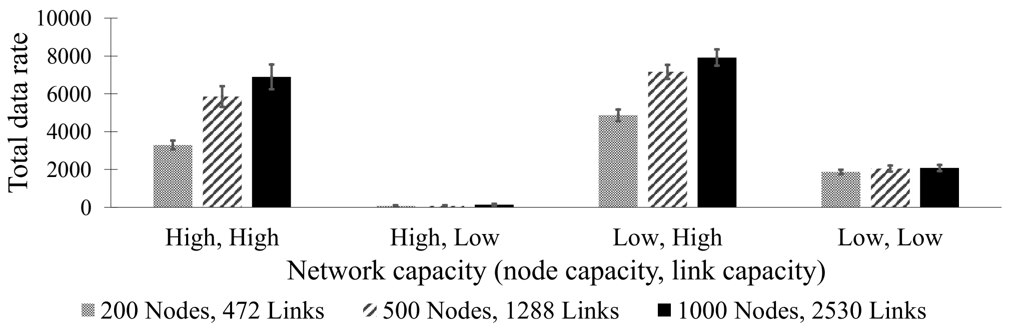
Figure 6.11: Scalability of the U-SPRING approaches

using the optimization approach, while the heuristic creates 261 instances.

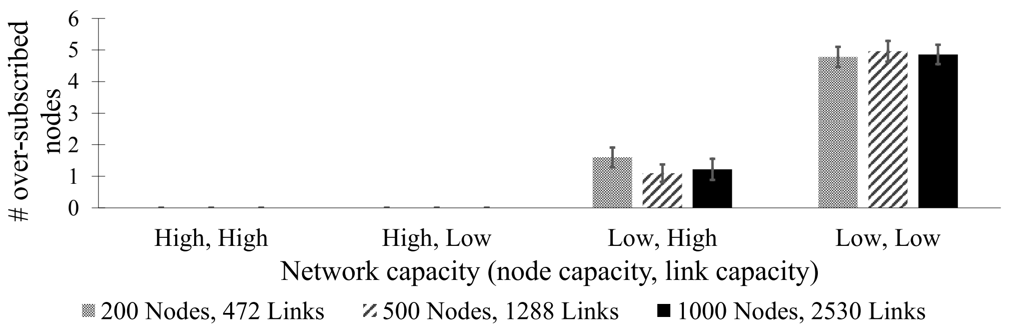
6.6.2 Scalability

Since the U-SPRING problem is NP-hard, the scalability of the optimization approach is limited. In order to illustrate the quality of the solutions found in acceptable time, we have done additional experiments. We have increased the source data rate of the VCS from the previous experiment (Section 6.6.1), leading to an increasing number of required instances. We have also increased the size of the substrate network. In each case, we have run the optimization approach with a time limit of 60 seconds. With this time limit, the MILP formulation is used as a heuristic to solve the problem in a limited time. The solver tries to embed the templates for 60 seconds and after that, it stops with the best solution and the best lower bound that the solver had found until that time. The measurements were performed on a machine with Intel Core i5-4210U CPU running at 1.70 GHz and 8 GB RAM.

Figure 6.11a shows the run time of the optimization approach for different data rates and substrate network sizes and Figure 6.11b shows the corresponding gap between the found solution and the lower bound (optimality gap). For a small



(a) Total used link capacity



(b) Number of over-subscribed nodes

Figure 6.12: Impact of different node and link capacities

network with 10 nodes and 20 arcs (shown in Figure 6.6), the algorithm computes optimal results for the lower half of source data rate values. For larger source data rates, the optimality gap is still acceptable (around 20 %). However, for a bigger substrate network with 20 nodes and 44 arcs (substrate graph eu_20_0_prob [109]), the solver quickly reaches the time limit with much smaller source data rate and also the optimality gap is much bigger. For even bigger substrate networks, the performance of the algorithm further deteriorates, up to the point where it cannot be run anymore because of memory problems. The large sensitivity to the size of the substrate network is because the number of variables of the MILP is cubic in the size of the substrate network.

As shown in Figure 6.11c, the run time of the heuristic approach remains very low even for the largest substrate networks: for 1000 nodes and 2530 arcs (substrate graph eu_1000_0_prob [109]), the run time is still below 20 milliseconds, which makes it practical for real-world problem sizes as well.

6.6.3 Analysis

To gain further insight into the inter-dependencies between the input and output parameters of the U-SPRING problem, we have experimented with different problem sizes and different levels of resource availability. For this, we have used the video streaming template shown in Figure 6.8 with a single source injecting a total data rate of 1000 data units per time unit into the network, from a node selected uniformly at random. We have used three substrate networks from the

VNMP instances, with

1. 200 nodes and 472 links (substrate graph eu_200_0_prob [109])
2. 500 nodes and 1288 links (substrate graph eu_500_0_prob [109])
3. 1000 nodes and 2530 links (substrate graph eu_1000_0_prob [109])

We have created low-capacity and high-capacity configurations as follows.

- CPU and memory capacities of each node selected independently and uniformly at random from the range [1,5] for low capacity and [10,50] for high capacity
- Capacity of each link selected independently and uniformly at random from the range [50,100] for low capacity and [500,1000] for high capacity

These ranges are based on the amount of resources required to embed the template with exactly one instance per component, as an estimation of the required resources for handling the input data rate.

We have run the heuristic 100 times on each setup (low/high node capacity and low/high link capacity) and each substrate network. Figure 6.12 shows some aggregated results, with confidence intervals at 95 % confidence level.

As shown in Figure 6.12a, the algorithm adapts the amount of used link capacity to the amount of available link capacity. On all three networks, in both setups with low link capacity, the links are carrying considerably less data rate than in the setups with high link capacity. This figure also shows that the algorithm uses more link capacity for embedding the same template as the network gets larger, increasing the total available link capacity in the network. With low link capacity, the algorithm concentrates the instances in as few nodes as possible; therefore, most of the traffic remains inside nodes instead of traveling across the network. Obviously, this can result in over-loaded nodes if the node capacities are not enough.

Figure 6.12b shows the number of network nodes with over-subscribed CPU capacities. In the setups with high node capacity, no CPU over-subscription is noticed. With low node capacity and low link capacity, the instances are concentrated in fewer nodes resulting in more node over-subscription. With low node capacity, even with high link capacity, over-subscription cannot be avoided in these experiments but fewer nodes are affected as the data rates could be distributed more freely across the network.

6.7 Conclusion

In this chapter, I have described the U-SPRING problem, the joint, single-step scaling, placement, and routing problem for uni-directional pliable VCSs. Depending on the amount of data rate these VCSs need to handle, different numbers of instances are required for each service component, in different locations with appropriate amount of CPU and memory allocation.

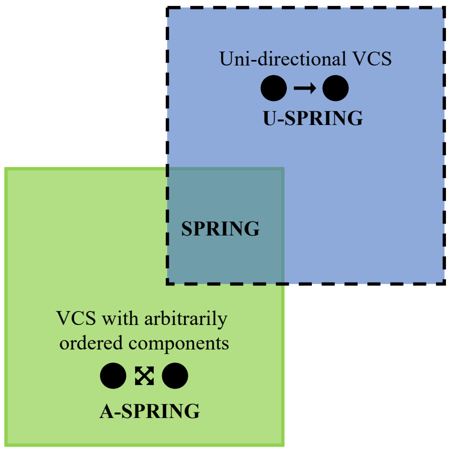


Figure 6.13: Relation of U-SPRING to A-SPRING

I have presented the problem as an MILP that can be used with an appropriate solver to find optimal solutions to the U-SPRING problem. As the problem is NP-hard, the optimization approach is not applicable for large problem instances, e.g., when a large number of templates should be embedded in the network, when the data rate injected into the sources is large, or when the substrate network has a large number of nodes and links.

I have also presented a heuristic that can find close-to-optimal solutions to the problem in a considerably shorter time than the optimization approach.

Using simulation-based evaluations, I have shown that the allocated node capacity closely follows the increasing and decreasing load, using both approaches. The heuristic cannot consider the requirements of the upcoming steps while deciding the scaling, placement, and routing for each individual instance. This is reflected in the values of the link-related metrics like the total latency and total used link capacity, which are much higher in the solutions delivered by the heuristic, compared to the optimal results.

These approaches show the feasibility of defining pliable VCSs. Such a flexible definition, paired with an accurate resource demand profile for the VCS, is a powerful tool for service providers as well as network operators in a typical service management and orchestration scenario. The service providers can define VCSs without the need for the (usually inaccurate) estimation of the exact number of required instances and the exact resource demands for each instance. Network operators also gain additional degrees of freedom in optimizing the overall state of the resources and the running services. For example, they can (re-)adjust the structure of all or some of the pliable services to influence the values of their performance metrics of interest, e.g., balancing the resource allocation among different services while keeping the agreed performance levels, changing the number of instances for a certain service component as a means of re-routing the traffic away from a congested link, etc.

As shown in Figure 6.13, the core functionality provided by the A-SPRING and U-SPRING approaches is the joint scaling, placement, and routing for pliable VCSs. These models and problem definitions, however, have fundamental differences. The A-SPRING approaches have a limited support for vertical scaling (only downscaling is possible) and horizontal scaling (only a simple linear relationship is assumed between resource consumption and load). The U-SPRING approaches have full-fledged horizontal and vertical up- and down-scaling capabilities but cannot handle pliable VCSs with arbitrarily ordered instances.

7 Embedding Bi-Directional Services with Load-Proportional Structures

In this chapter, I propose solutions to the joint scaling, placement, routing problem for bi-directional pliable Virtualized Composed Services (VCSs) (the B-SPRING problem). I define the model and assumptions used for the B-SPRING problem in Section 7.1. I summarize the problem formulation in Section 7.2 and describe its complexity in Section 7.3. In Section 7.4 and Section 7.5, I present the optimization and heuristic approaches to this problem, respectively, which I have developed together with S. Schneider and H. Karl. An initial version of these approaches were developed in the course of the master's thesis of S. Schneider [40]. However, we have made significant modifications to the model and

Table 7.1: B-SPRING Substrate Network Parameters

Symbol	Definition
$G_{\text{sub}} = (V, L)$	Substrate network graph.
$v \in V$, $l \in L$	Substrate network nodes, links.
$\text{cap}_{\text{cpu}}(v)$, $\text{cap}_{\text{mem}}(v)$	CPU, memory capacity of v.
$\text{cap}(l)$, $d(l)$	Capacity, delay of l.

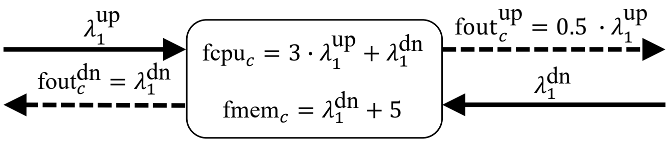

λ_1^{up}
$\text{fout}_c^{\text{dn}} = \lambda_1^{\text{dn}}$
$\text{fcpu}_c = 3 \cdot \lambda_1^{\text{up}} + \lambda_1^{\text{dn}}$
$\text{fmem}_c = \lambda_1^{\text{dn}} + 5$
$\text{fout}_c^{\text{up}} = 0.5 \cdot \lambda_1^{\text{up}}$
λ_1^{dn}

Figure 7.1: Resource demands and data rates of an example component

solution approaches, afterwards. I show the evaluation results for the approaches in Section 7.6. Section 7.7 includes the conclusion from the chapter. This chapter partially includes figures and verbatim copies of the text from my paper [32].

7.1 Model

I describe each network service by a *service template*. Based on the data rate resulting from different flows in multiple *source* locations, each service template is scaled to create an *overlay*. Overlays include all required instances per service component as well as their location in the substrate network, their required resources, and their ingoing and outgoing data rates. In this section, I describe the model for each of these entities. Parts of these definitions, e.g., basic definition of templates and components, are similar to the U-SPRING model described in Chapter 6, therefore, I only focus on the new aspects here.

7.1.1 Substrate Network

The substrate network for the B-SPRING model is a connected, directed graph similar to the model described in Section 6.1.1. Table 7.1 shows a summary of the network-related parameters used in this chapter.

7.1.2 Service Template

A service template is a connected, directed graph $G_T = (C_T, A_T)$ that describes the structure of a bi-directional pliable VCS. Figure 5.2a and 5.2b show example templates for bi-directional services. I define the service templates for the B-SPRING problem similar to those of the U-SPRING problem (Section 6.1.2), with the following differences to model the bi-directional service flows.

Each component $c \in C_T$ has a given number of *upstream and downstream* inputs $n_{\text{in}}^{\text{up}}(\text{c}) \leq 0$ and $n_{\text{in}}^{\text{dn}}(\text{c}) \leq 0$ as well as a given number of upstream and downstream outputs $n_{\text{out}}^{\text{up}}(\text{c}) \leq 0$ and $n_{\text{out}}^{\text{dn}}(\text{c}) \leq 0$, representing the ingoing and outgoing connection points. Uni-directional VCSs can be modeled if connection points are created only in one direction and the number of connection points in the opposite direction is set to zero.

By explicitly distinguishing upstream and downstream inputs and outputs, the bi-directional service template can be modeled as a graph that is acyclic in each direction. In this way, the amount of load that needs to be forwarded over each output can be calculated based the load on the corresponding input in the right direction.

The resource consumption for each component c depends on the data rates at the upstream and downstream inputs and is defined by the pre-defined functions $\text{fcpu}_c(\Lambda) : \mathbb{R}_{\geq 0}^{n_{\text{in}}^{\text{up}}(\text{c}) + n_{\text{in}}^{\text{dn}}(\text{c})} \rightarrow \mathbb{R}_{\geq 0}$ and $\text{fmem}_c(\Lambda) : \mathbb{R}_{\geq 0}^{n_{\text{in}}^{\text{up}}(\text{c}) + n_{\text{in}}^{\text{dn}}(\text{c})} \rightarrow \mathbb{R}_{\geq 0}$, representing the required amount of CPU and memory, respectively. Λ is the vector of input data rates on upstream and downstream inputs of the component.

Similarly, the upstream and downstream outgoing data rates of a component c are relative to the incoming data rates (in the respective direction) and are specified by the pre-defined functions $\text{fout}_c^{\text{up}}(\Lambda) : \mathbb{R}_{\geq 0}^{n_{\text{in}}^{\text{up}}(\text{c}) + n_{\text{in}}^{\text{dn}}(\text{c})} \rightarrow \mathbb{R}_{\geq 0}^{n_{\text{out}}^{\text{up}}(\text{c})}$ and $\text{fout}_c^{\text{dn}}(\Lambda) : \mathbb{R}_{\geq 0}^{n_{\text{in}}^{\text{up}}(\text{c}) + n_{\text{in}}^{\text{dn}}(\text{c})} \rightarrow \mathbb{R}_{\geq 0}^{n_{\text{out}}^{\text{dn}}(\text{c})}$, respectively. If there are multiple outputs in one direction, the traversing flows can be split across these outputs as defined by these functions.

Figure 7.1 shows example functions for a component that receives an expected data rate of λ_1^{up} and λ_1^{dn} on its upstream and downstream inputs. The functions define the resource demands and the outgoing data rates of the component using the ingoing data rates.

To correctly define bi-directional VCSs and distinguish upstream and downstream traffic of the flows in the model, I associate the following different roles with the components in the templates.

Each template has a mandatory single *source* component (SRC, for short), e.g., S^A and S^B in Figures 5.2a,5.2b. Source components have zero resource demands, no upstream inputs, and a single output with unspecified data rate. They may have downstream inputs, e.g., for receiving results back.

I refer to components with only upstream inputs and only downstream outputs, e.g., the SRV components in Figure 5.2a,5.2b) as *END* components. Each bi-directional template has at least one END component.

I refer to components that are neither SRC nor END as *intermediate* components (INT, for short). These components send out the flows received at an upstream (or downstream) input through at least one upstream (or downstream) output, with a data rate that may be lower than, higher than, or equal to its incoming data rate.

Assigning the roles SRC, INT, and END simplifies the notations and the problem formulation in Section 7.4. The components can be marked automatically, if the upstream and downstream inputs and outputs are distinguished. This can be defined as part of the service template for each component by the service provider or the developer, who is familiar with the expected functionality of the service components.

Table 7.2: B-SPRING Template Parameters

Symbol	Definition
$G_T = (C_T, A_T)$	Template graph.
$c \in C_T$, $a \in A_T$	Components and arcs of template T.
$n_{\text{in}}^{\text{up}}(\text{c})$, $n_{\text{in}}^{\text{dn}}(\text{c})$	Number of upstream and downstream inputs of c.
$n_{\text{out}}^{\text{up}}(\text{c})$, $n_{\text{out}}^{\text{dn}}(\text{c})$	Number of upstream and downstream outputs of c.
$\text{fcpu}_c(\Lambda)$, $\text{fmem}_c(\Lambda)$	CPU, memory demands of component c based on Λ, the vector of data rates on upstream and downstream inputs of c.
$\text{fout}_c^{\text{up}}(\Lambda)$, $\text{fout}_c^{\text{dn}}(\Lambda)$	Data rates on upstream, downstream outputs of component c, calculated based on Λ, the vector of data rates on upstream and downstream inputs of c.
src_a, dst_a	Component where arc a begins, ends.
$d_a^{\max}$	Maximum delay for a.

Additionally, components can be specified as *stateful*, indicating that their instances maintain some internal state for each traversing flow. If both upstream and downstream traffic of a flow traverse an instance of a stateful component, they have to traverse *exactly the same instance* in both directions. This is not required for upstream and downstream traffic of a flow traversing the same *stateless* component; in that case, the traffic may be routed over different instances. When traversing a component, each flow is handled by exactly one instance of that component. In this way, no state inconsistency can occur for flows that traverse a stateful component only in one direction. For load balancing, different flows may be assigned to different instances of a component. In the U-SPRING model, the whole traffic initiating from a source of the template is considered as one flow. In the B-SPRING model, different flows with a different data rate can initiate from a source location, as I describe in Section 7.1.3.

Fixed components are also special components that are pinned to a certain location in the substrate network, e.g., to model the end points (sinks) of service flows or legacy Physical Network Functions (PNFs). They cannot be re-located or scaled. I also assume their resource demands are zero, as they are pre-defined and fixed and are not calculated by the SPRING solution.

A directed arc $a \in A_T$ connects exactly one output of a component to exactly one input of another component in the service template. To simplify the formulation of the optimization problem, I annotate each arc either as a upstream or a downstream arc. Upstream (or downstream) arcs can only connect upstream (or downstream) outputs to upstream (or downstream) inputs. Each arc a is additionally annotated with a delay bound $d_a^{\max}$, specifying the maximum delay that can be tolerated between the corresponding components.

Table 7.2 shows a summary of the parameters related to the service templates.

7.1.3 Template Embedding

The *template embedding* process for bi-directional VCSs involves deciding:

- how many instances of each component (*horizontal scaling*),
- with how many resources (*vertical scaling*),
- need to be instantiated in which locations (*placement*),
- and how the upstream and downstream traffic should be routed among them (*routing*).

Considering both upstream and downstream traffic in the last point differentiates the template embedding process for bi-directional VCSs from that of uni-directional VCSs (Section 6.1.3).

The outcome of the template embedding process is an overlay, which I describe in Section 7.1.4. The required inputs for this process are similar to the U-SPRING template embedding model, described in Section 6.1.3.

I define the source instances in a slightly different way from the U-SPRING model. Each instance of a source component (e.g., representing different populations of users in different geographic locations) has a fixed location and a given outgoing data rate for each flow starting at this source. For every source location $v \in V$, I assume that a set of flows together with their data rates $(f, r_f) \in F_{T,v}$ is given. $S_T = \{F_{T,v} | v \text{ a source location of } T\}$ collects this information (which typically changes over time). This fine-grained control over individual flows is required to ensure different flows can traverse the right stateful instances in upstream and downstream paths and return to the right source locations.

In addition to the inputs required for the template embedding model in the U-SPRING problem, a template T might also include fixed components. In this case, they should also be given as a part of the input. Fixed components are given as a set X_T of tuples (c, v). $c \in C_T$ shows the fixed component and $v \in V$ is the network node where it is located.

Table 7.3 includes an overview of the parameters related to the template embedding process for the B-SPRING problem.

7.1.4 Overlay

Based on the sources of each template as well as the capacities of the substrate network, an overlay $G_{\text{OL}}(T) = (I_{\text{OL}}(T), E_{\text{OL}}(T))$ for each service template T is created. I define the overlays in the B-SPRING model similar to those of the U-SPRING model, described in Section 6.1.4.

Figure 5.2 shows example overlays of the templates shown in Figure 5.2a and 5.2b embedded in the substrate network. After the services have been scaled in Figure 5.2d, there are two instances of the source component S^{A} and one instance of S^{B}. Accordingly, the FW component is instantiated twice.

Table 7.3 includes an overview of the overlay-related parameters for the B-SPRING problem.

Table 7.3: B-SPRING Template Embedding and Overlay Parameters

Symbol	Definition
$(c, F_{T,v}) \in S_T$	Source c of template T with its flows at node v
$(f, r_f) \in F_{T,v}$	Flow f with data rate r_f
$(c, v) \in P_T$	An existing instance of component c previously embedded at node v.
$(c, v) \in X_T$	An instance of component c from template T fixed to node v.
$\mathcal{T}$	All templates to be embedded.
$\mathcal{C} = \bigcup_{T \in \mathcal{T}} C_T$	All components from templates in $\mathcal{T}$.
$\mathcal{C}_{\text{SRC}}, \mathcal{C}_{\text{INT}}, \mathcal{C}_{\text{END}} \subset \mathcal{C}$	All SRC, INT, END components
$\mathcal{C}_{\text{FIX}}, \mathcal{C}_{\text{STF}} \subset \mathcal{C}$	All fixed, stateful components
$\mathcal{A} = \bigcup_{T \in \mathcal{T}} A_T$	All arcs of templates in $\mathcal{T}$.
$\mathcal{A}_{\text{up}}, \mathcal{A}_{\text{dn}} \subset \mathcal{A}$	All upstream, downstream arcs
$\mathcal{S} = \bigcup_{T \in \mathcal{T}} S_T$	All sources of templates in $\mathcal{T}$.
$\mathcal{X} = \bigcup_{T \in \mathcal{T}} X_T$	All fixed instances of templates in $\mathcal{T}$.
$G_{\text{OL}}(T) = (I_{\text{OL}}(T), E_{\text{OL}}(T))$	Overlay graph corresponding to template T.
$\mathcal{F}$	All flows from all sources of all templates
$i \in I_{\text{OL}}(T)$, $e \in E_{\text{OL}}(T)$	Instances, edges of overlay.
$M_T^C(i)$	The corresponding component of instance i.
$M_T^V(i)$	The node where instance i is mapped to.
$M_T^A(e)$	The corresponding arc of edge e.

7.2 Problem Formulation

B-SPRING is the problem of finding the optimal embedding for a set of bi-directional pliable VCSs in the substrate network. For service template embedding, the following inputs are required:

- Substrate network
- A set of bi-directional service templates, possibly with stateful components, possibly sharing some components
- For each template, a set of flows and sources
- A set of instances pinned to fixed locations (optional, can be empty)
- A previous embedding of the service templates (optional, can be empty)

The template embedding process can be applied either for the initial embedding of VCSs into an empty network or for adjusting an existing embedding.

I consider a valid *system configuration* similar to the definition provided for the U-SPRING problem in Section 6.2. The B-SPRING problem has the additional constraint that the total delay of the paths created for an arc cannot exceed the maximum tolerable delay defined for the arc.

I define the following metrics of interest:

- The maximum amount of over-subscription for node and link resources
- The number of instances that should be added or removed
- The total resource consumption of all overlays
- The total delay over all created paths for the overlays

The values of these metrics should be minimized to give valid solutions that are desirable for the service providers and the network providers.

This model is flexible with respect to priorities for the optimization objectives. For example, over-subscription of resources could be forbidden or allowed; over-subscription avoids rejecting requests and improves utilization but jeopardizes service performance objectives. During the embedding process, one of the objectives is to minimize the maximum over-subscription over all node and link resources. Alternatively, strict limits for the amount of over-subscription can be set.

If multiple service templates include the same component c (e.g., specified using the same identifier), instances of c can be reused in the overlays of these service templates (if this is undesirable, one can easily create a copy of c with another identifier). In this case, the traffic belonging to each network service has to be separated to ensure the correct forwarding of the corresponding flows. For this, each shared component c should be adapted to create the required inputs and outputs by each service template for the shared component. The specific outputs created for each template should be annotated with the $\mathrm{fout}_c^{\mathrm{up}}$ and $\mathrm{fout}_c^{\mathrm{dn}}$ defined in that template. In this model, the number and order of inputs and outputs for the shared components must be the same in all templates that share the component. With this assumption, the fcpu_c, fmem_c functions can be adjusted automatically. For this, each variable in the function representing a specific input should be replaced with the sum of variables for all equivalent inputs from all templates. The constant values of the combined functions remain unchanged after adaptation. Each overlay can use its dedicated inputs and outputs on the instances of the shared components. Figure 7.2 shows how the CPU, memory, and data rate functions of the example component from Figure 5.2c are adapted to allow two service templates to share this component. Reusing the component results in a lower idle resource consumption than the case where separate components are used for each template.

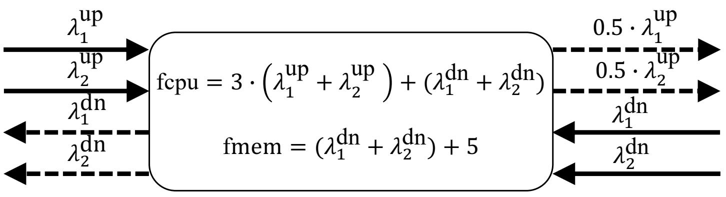

λ_1^{up}
λ_2^{up}
λ_1^{dn}
λ_2^{dn}
fcpu = $3 \cdot (\lambda_1^{up} + \lambda_2^{up}) + (\lambda_1^{dn} + \lambda_2^{dn})$
fmem = $(\lambda_1^{dn} + \lambda_2^{dn}) + 5$
$0.5 \cdot \lambda_1^{up}$
$0.5 \cdot \lambda_2^{up}$
λ_1^{dn}
λ_2^{dn}

Figure 7.2: Resource demands and data rates of the example component in Figure 5.2, adapted to be shared between two templates

7.3 Problem Complexity

Using polynomial-time reduction, I show that for an instance of the B-SPRING problem, it is an NP-complete problem to decide if a solution exists where the over-subscription of (node and link) resources is zero. Based on the given load and resource capacities, it is possible to check in time polynomial in the size of the problem input whether an embedding of a set of templates results in any over-subscription. The size of the solution is also polynomial in the size of the input, so the problem is in NP.

The B-SPRING problem is an extension of the U-SPRING problem, which has been proven to be NP-complete [34]. I show a reduction of U-SPRING to B-SPRING, proving its NP-hardness. Given an instance of U-SPRING, I construct an instance of B-SPRING as follows.

As U-SPRING only considers uni-directional service templates, for every component in the U-SPRING model, I consider the inputs/outputs as upstream inputs/outputs. Similarly, I consider every template arc as an upstream arc. U-SPRING does not include any delay bound for arcs, so for every arc I set the maximum delay to infinity. In U-SPRING, every instance of a source component c at node v with data rate r is specified as (c, v, r). There is no stateful component model in U-SPRING, so the flows from sources can be distributed freely over different instances of each component.

To create a corresponding scaling and load balancing behavior in B-SPRING, I transform every such source instance of every template T into a source instance with M flows, $F_{T,v} = \{(f_1, r/M), (f_2, r/M), \cdots, (f_M, r/M)\}$. To get a limited number of digits after the decimal point, I assume all input parameters are rational numbers and M is a sufficiently large number to create data rate values with the desired precision. E.g., if an implementation of U-SPRING supports values with 2 digits after the decimal point for data rates of overlay edges, I translate each source instance with data rate r into $r/0.01$ flows starting from this source, each flow having a data rate of 0.01.

Using the remaining input parameters directly as provided for U-SPRING, this is now a complete instance of the B-SPRING problem. If there is a solution for a U-SPRING problem instance with no violation of capacity constraints, then the corresponding B-SPRING problem instance also has a solution without over-subscription of the substrate network resources. Similarly, combining the data rates of different flows from each source instance into a joint data rate, a solution with no over-subscription found for B-SPRING is also a solution with no violations for U-SPRING.

The reduction can be performed in time polynomial in the size of the input, so B-SPRING is an NP-hard problem. Together with the fact that it is in NP, it follows that this problem is an NP-complete problem.

7.4 Optimization Approach

In this section, I present an Mixed-Integer Program (MIP) formulation for B-SPRING, which is an extension of the MIP formulation of the U-SPRING problem (Section 6.4). Table 7.1, 7.2, and 7.3 show an overview of the input parameters to this problem.

All constraints are linear. The problem is a Mixed-Integer Linear Program (MILP) if the functions fcpu_c, fmem_c, $\mathrm{fout}_c^{\mathrm{up}}$, and $\mathrm{fout}_c^{\mathrm{dn}}$ are linear for each component c.

Given the inputs described in Section 7.2, the following MIP formulation can be used to find optimal solutions to the B-SPRING problem.

The decision variables are presented in Table 7.4. In the rest of this chapter, $\mathcal{M}$ represents a constant that is sufficiently large, used in the so-called Big-M formulations. $(W)_k$ shows the k-th component of a vector W. $\underline{0}$ is a zero vector of appropriate length. The constant values $x^*_{c,v}$ defined as follows are also a part of the problem input, showing previous embeddings of service components:

$$\forall (c,v) \in P_T : x^*_{c,v} = 1$$
$$\forall c \in \mathcal{C}, \forall v \in V, \text{if } (c,v) \notin P_T : x^*_{c,v} = 0$$

7.4.1 Constraints

In this section, I describe the constraints for the B-SPRING optimization approach.

Mapping Consistency Rules

$$\forall v \in V, \forall c \in \mathcal{C}_{\mathrm{SRC}} : \qquad x_{c,v} = \begin{cases} 1 & \text{if } \exists (c, F_{T,v}) \in \mathcal{S} \\ 0 & \text{else} \end{cases} \tag{7.1}$$

$$\forall v \in V, \forall c \in \mathcal{C}_{\mathrm{fixed}} : \qquad x_{c,v} = \begin{cases} 1 & \text{if } \exists (c, v) \in \mathcal{X} \\ 0 & \text{else} \end{cases} \tag{7.2}$$

Sources and fixed components are assigned to their pre-defined locations using Constraint 7.1 and Constraint 7.2, respectively. For each source instance, the data rate of the flows starting at that source are assigned to the corresponding outputs, using Constraint 7.3.

Table 7.4: B-SPRING Decision Variables

Variable	Definition
$x_{c,v}$	1 iff an instance of component c is mapped to node v
$\delta_{c,v}$	1 iff $x_{c,v} \neq x^*_{c,v}$, i.e., an instance of component c is added or removed at node v
$\text{cpu}_{c,v}$, $\text{mem}_{c,v}$	CPU, memory demands of the instance of component c at node v, or 0 if no such instance exists
$t^{\text{up}}_{c,v,f}$, $t^{\text{dn}}_{c,v,f}$	1 iff upstream, downstream traffic of a flow f traverses an instance of component c at node v
$\text{in}^{\text{up}}_{c,v,f}$, $\text{in}^{\text{dn}}_{c,v,f}$	Vector of data rates at inputs of the instance of component c at node v, corresponding to flow f, or an all-zero vector
$\text{out}^{\text{up}}_{c,v,f}$, $\text{out}^{\text{dn}}_{c,v,f}$	Vector of data rates at outputs of the instance of component c at node v, corresponding to flow f, or an all-zero vector
$e_{a,v,v',f}$	1 iff for an arc a, an overlay edge between nodes v and v' corresponding to flow f is created
$z_{a,v,v',l}$	Data rate on link l corresponding to an arc a that connects an instance of component c at node v to an instance of component c' at node v', or 0
$\zeta_{a,v,v',l}$	1 iff $z_{a,v,v',l} > 0$
ψ_{cpu}, ψ_{mem}, ψ_{dr}	Maximum CPU, memory, link capacity over-subscription

$\forall v \in V, \forall f \in \mathcal{F}, \forall c \in \mathcal{C}_{\text{SRC}}, \forall T \in \mathcal{T}$:

$$\text{out}^{\text{up}}_{c,v,f} = \begin{cases} r_f & \text{if } (c, F_{T,v}) \in S_T, \exists (f, r_f) \in F_{T,v} \\ 0 & \text{else} \end{cases} \tag{7.3}$$

When going from one solution to another, the added and removed instances on each node are tracked (Constraint 7.4). If an instance of a component is created on a node, the right number of upstream (Constraint 7.5,7.7) and downstream (Constraint 7.6,7.8) inputs and outputs should be created on that instance.

$$\forall c \in \mathcal{C}, \forall v \in V : \qquad \delta_{c,v} = \begin{cases} x_{c,v} & \text{if } x^*_{c,v} = 0 \\ 1 - x_{c,v} & \text{if } x^*_{c,v} = 1 \end{cases} \tag{7.4}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n^{\text{up}}_{\text{in}}(c)] : \qquad (\text{in}^{\text{up}}_{c,v,f})_k \leq \mathcal{M} \cdot x_{c,v} \tag{7.5}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n^{\text{dn}}_{\text{in}}(c)] : \qquad (\text{in}^{\text{dn}}_{c,v,f})_k \leq M \cdot x_{c,v} \tag{7.6}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n^{\text{up}}_{\text{out}}(c)] : \qquad (\text{out}^{\text{up}}_{c,v,f})_k \leq \mathcal{M} \cdot x_{c,v} \tag{7.7}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n^{\text{dn}}_{\text{out}}(c)] : \qquad (\text{out}^{\text{dn}}_{c,v,f})_k \leq \mathcal{M} \cdot x_{c,v} \tag{7.8}$$

Flow and Data Rate Rules

$$\forall c \in \mathcal{C}, \forall v \in V, \forall f \in \mathcal{F}: \quad \text{if } c \in \mathcal{C}_{\text{INT}}: \text{out}^{\text{up}}_{c,v,f} = \text{func}^{\text{up}}_{c}(\text{in}^{\text{up}}_{c,v,f}) - (1 - x_{c,v}) \cdot \text{func}^{\text{up}}_{c}(\underline{0}) \tag{7.9}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall f \in \mathcal{F}: \quad \text{if } c \in \mathcal{C}_{\text{INT}}: \text{out}^{\text{dn}}_{c,v,f} = \text{func}^{\text{dn}}_{c}(\text{in}^{\text{dn}}_{c,v,f}) - (1 - x_{c,v}) \cdot \text{func}^{\text{dn}}_{c}(\underline{0}) \tag{7.10}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall f \in \mathcal{F}: \quad \text{if } c \in \mathcal{C}_{\text{END}}: \text{out}^{\text{dn}}_{c,v,f} = \text{func}^{\text{dn}}_{c}(\text{in}^{\text{up}}_{c,v,f}) - (1 - x_{c,v}) \cdot \text{func}^{\text{dn}}_{c}(\underline{0}) \tag{7.11}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall f \in \mathcal{F}: \quad \mathcal{M} \cdot t^{\text{up}}_{c,v,f} \geq \sum_{k \in [1, n^{\text{up}}_{\text{in}}(c)]} (\text{in}^{\text{up}}_{c,v,f})_k + \sum_{k \in [1, n^{\text{up}}_{\text{out}}(c)]} (\text{out}^{\text{up}}_{c,v,f})_k \tag{7.12}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall f \in \mathcal{F}: \quad t^{\text{up}}_{c,v,f} \leq \mathcal{M} \cdot \sum_{k \in [1, n^{\text{up}}_{\text{in}}(c)]} (\text{in}^{\text{up}}_{c,v,f})_k + \sum_{k \in [1, n^{\text{up}}_{\text{out}}(c)]} (\text{out}^{\text{up}}_{c,v,f})_k \tag{7.13}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall f \in \mathcal{F}: \quad \mathcal{M} \cdot t^{\text{dn}}_{c,v,f} \geq \sum_{k \in [1, n^{\text{dn}}_{\text{in}}(c)]} (\text{in}^{\text{dn}}_{c,v,f})_k + \sum_{k \in [1, n^{\text{dn}}_{\text{out}}(c)]} (\text{out}^{\text{dn}}_{c,v,f})_k \tag{7.14}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall f \in \mathcal{F}: \quad t^{\text{dn}}_{c,v,f} \leq \mathcal{M} \cdot \sum_{k \in [1, n^{\text{dn}}_{\text{in}}(c)]} (\text{in}^{\text{dn}}_{c,v,f})_k + \sum_{k \in [1, n^{\text{dn}}_{\text{out}}(c)]} (\text{out}^{\text{dn}}_{c,v,f})_k \tag{7.15}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall f \in \mathcal{F}: \text{if } c \in \mathcal{C}_{\text{state}}: t^{\text{up}}_{c,v,f} = t^{\text{dn}}_{c,v,f} \tag{7.16}$$

The data rate of upstream and downstream traffic of flows entering an instance of a component determines the data rate of the upstream and downstream traffic that leaves that instance (Constraint 7.9,7.10). As instances of END components (Section 7.1.2) can only have upstream inputs and downstream outputs, they are treated by a special rule (Constraint 7.11). Constraint 7.12–7.15 keep track of the instances of components that upstream or downstream traffic of each flow traverse. This is required to make sure that each flow traverses exactly the same instance of a stateful component in both directions, ensured by Constraint 7.16.

A single flow cannot be split over multiple instances of a component (Section 7.1.2). For this, if a flow traverses an instance at node v, it is allowed to enter and exit this instance using exactly one edge per corresponding arc (Constraint 7.18, 7.20). Instances of END components again need an special treatment (Constraint 7.17). For each edge created this way, the data rate is also calculated on the corresponding input/output of its source/destination instances (Constraint 7.19, 7.21).

$$\forall a \in \mathcal{A}, \forall v \in V, \forall f \in \mathcal{F} : \text{if } \mathrm{src}(a) \in \mathcal{C}_{\mathrm{END}} : \sum_{v' \in V} e_{a,v,v',f} = t^{\mathrm{up}}_{c,v,f} \tag{7.17}$$

$$\begin{aligned}&\forall a \in \mathcal{A}, \forall v \in V, \forall f \in \mathcal{F},\\ &\quad \text{if } \mathrm{src}(a) \in \mathcal{C}_{\mathrm{END}}, \text{if } a \in \mathcal{A}_{\mathrm{up}} \text{ from output } k \text{ of } \mathrm{src}(a) \text{ to input } k' \text{ of } \mathrm{dst}(a) :\\ &\qquad \sum_{v' \in V} e_{a,v,v',f} = t^{\mathrm{up}}_{c,v,f}\end{aligned} \tag{7.18}$$

$$\begin{aligned}&\forall a \in \mathcal{A}, \forall v \in V, \forall f \in \mathcal{F},\\ &\quad \text{if } \mathrm{src}(a) \in \mathcal{C}_{\mathrm{END}}, \text{if } a \in \mathcal{A}_{\mathrm{up}} \text{ from output } k \text{ of } \mathrm{src}(a) \text{ to input } k' \text{ of } \mathrm{dst}(a) :\\ &\qquad \sum_{v' \in V} (\mathrm{out}^{\mathrm{up}}_{\mathrm{src}(a),v',f})_k \cdot e_{a,v',v,f} = (\mathrm{in}^{\mathrm{up}}_{\mathrm{dst}(a),v,f})_{k'}\end{aligned} \tag{7.19}$$

$$\begin{aligned}&\forall a \in \mathcal{A}, \forall v \in V, \forall f \in \mathcal{F},\\ &\quad \text{if } \mathrm{src}(a) \in \mathcal{C}_{\mathrm{END}}, \text{if } a \in \mathcal{A}_{\mathrm{dn}} \text{ from output } k \text{ of } \mathrm{src}(a) \text{ to input } k' \text{ of } \mathrm{dst}(a) :\\ &\qquad \sum_{v' \in V} e_{a,v,v',f} = t^{\mathrm{dn}}_{c,v,f}\end{aligned} \tag{7.20}$$

$$\begin{aligned}&\forall a \in \mathcal{A}, \forall v \in V, \forall f \in \mathcal{F},\\ &\quad \text{if } \mathrm{src}(a) \in \mathcal{C}_{\mathrm{END}}, \text{if } a \in \mathcal{A}_{\mathrm{dn}} \text{ from output } k \text{ of } \mathrm{src}(a) \text{ to input } k' \text{ of } \mathrm{dst}(a) :\\ &\qquad \sum_{v' \in V} (\mathrm{out}^{\mathrm{dn}}_{\mathrm{src}(a),v',f})_k \cdot e_{a,v',v,f} = (\mathrm{in}^{\mathrm{dn}}_{\mathrm{dst}(a),v,f})_{k'}\end{aligned} \tag{7.21}$$

The data rates of individual flows over created edges are mapped to links in the substrate network; Constraint 7.22–7.25 ensure flow conservation over the path(s) that the flows take. During path creation, the total delay of the links corresponding to an edge cannot exceed the maximum delay specified for its arc (Constraint 7.26). Constraint 7.27 prevents an overlay edge being mapped to a path with a loop.

$$\begin{aligned}&\forall a \in \mathcal{A}, a \text{ starts at output } k \text{ of } \mathrm{src}(a), \forall v, v_1, v_2 \in V :\\ &\qquad \sum_{vv' \in L} z_{a,v_1,v_2,vv'} - \sum_{v'v \in L} z_{a,v_1,v_2,v'v} =\\ &\begin{cases} 0 & \text{if } v \neq v_1, v \neq v_2\\ 0 & \text{if } v = v_1 = v_2\\ \sum_{f \in \mathcal{F}} e_{a,v_1,v_2,f} \cdot (\mathrm{out}^{\mathrm{up}}_{\mathrm{src}(a),v_1,f})_k & \text{if } v = v_1, v_1 \neq v_2, a \in \mathcal{A}_{\mathrm{up}}\\ \sum_{f \in \mathcal{F}} e_{a,v_1,v_2,f} \cdot (\mathrm{out}^{\mathrm{dn}}_{\mathrm{src}(a),v_1,f})_k & \text{if } v = v_1, v_1 \neq v_2, a \in \mathcal{A}_{\mathrm{dn}} \end{cases}\end{aligned} \tag{7.22}$$

$$\forall a \in \mathcal{A}, \forall v_1, v_2 \in V, \forall l \in L: \quad z_{a,v_1,v_2,l} \leq \mathcal{M} \cdot \zeta_{a,v_1,v_2,l} \tag{7.23}$$

$$\forall a \in \mathcal{A}, \forall v_1, v_2 \in V, \forall l \in L: \quad \zeta_{a,v_1,v_2,l} \leq \mathcal{M} \cdot z_{a,v_1,v_2,l} \tag{7.24}$$

$$\forall a \in \mathcal{A}, \forall v_1, v_2 \in V, \forall l \in L: \quad \zeta_{a,v_1,v_2,l} \leq \sum_{f \in \mathcal{F}} e_{a,v_1,v_2,f} \tag{7.25}$$

$$\forall a \in \mathcal{A}, \forall v_1, v_2 \in V: \quad \sum_{l \in L} \zeta_{a,v_1,v_2,l} \cdot d(l) \leq d_{\max}(a) \tag{7.26}$$

$$\forall a \in \mathcal{A}, \forall v_1, v_2 \in V, \forall v'v'' \in L, \text{if } v''v' \in L: \quad \zeta_{a,v_1,v_2,v'v''} + \zeta_{a,v_1,v_2,v''v'} \leq 1 \tag{7.27}$$

Calculation of Resource Consumption

$\forall c \in \mathcal{C}, \forall v \in V:$

$$\text{cpu}_{c,v} = \text{func}_c^{\text{cpu}}\Big(\sum_{f \in \mathcal{F}} \text{in}_{c,v,f}^{\text{up+dn}}\Big) - (1 - x_{c,v}) \cdot \text{func}_c^{\text{cpu}}(\underline{0}) \tag{7.28}$$

$\forall c \in \mathcal{C}, \forall v \in V:$

$$\text{mem}_{c,v} = \text{func}_c^{\text{mem}}\Big(\sum_{f \in \mathcal{F}} \text{in}_{c,v,f}^{\text{up+dn}}\Big) - (1 - x_{c,v}) \cdot \text{func}_c^{\text{mem}}(\underline{0}) \tag{7.29}$$

The combined data rate of the flows on upstream and downstream inputs of each created instance determines the resource demands of that instance (Constraint 7.28, 7.29).

Constraint 7.30–7.32 keep track of the maximum over-subscription of node and link resources to be able to minimize or bound it as required.

$$\forall c \in \mathcal{C}, \forall v \in V: \quad \sum_{c \in \mathcal{C}} \text{cpu}_{c,v} - \text{cap}_{\text{cpu}}(v) \leq \psi_{\text{cpu}} \tag{7.30}$$

$$\forall c \in \mathcal{C}, \forall v \in V: \quad \sum_{c \in \mathcal{C}} \text{mem}_{c,v} - \text{cap}_{\text{mem}}(v) \leq \psi_{\text{mem}} \tag{7.31}$$

$$\forall l \in L: \quad \sum_{a \in \mathcal{A}, v, v' \in V} z_{a,v,v',l} - \text{cap}_{\text{dr}}(l) \leq \psi_{\text{dr}} \tag{7.32}$$

7.4.2 Optimization Objective

Based on the problem formulation in Section 7.2, we have defined the following objective functions for the MIP:

- obj_1: Minimize the maximum node and link resource over-subscription

$$\text{min. } \psi_{\text{cpu}} + \psi_{\text{mem}} + \psi_{\text{dr}}$$

- obj_2: Minimize the number of added and removed instances

$$\text{min. } \sum_{j \in \mathcal{C}, v \in V} \delta_{j,v}$$

- obj$_3$: Minimize the total node and link resource consumption

$$\min. \sum_{j \in \mathcal{C}, v \in V} (\text{cpu}_{j,v} + \text{mem}_{j,v}) + \sum_{a \in \mathcal{A}, v, v' \in V, l \in L} z_{a,v,v',l}$$

- obj$_4$: Minimize the total delay

$$\min. \sum_{a \in \mathcal{A}, v, v' \in V, l \in L} d(l) \cdot \zeta_{a,v,v',l}$$

In practice, jointly optimizing all four objectives is necessary to serve the requirements of service providers and network operators. Therefore, we have defined the following lexicographical combination of the four objectives:

$$\text{minimize } w_1 \cdot \text{obj}_1 + w_2 \cdot \text{obj}_2 + w_3 \cdot \text{obj}_3 + w_4 \cdot \text{obj}_4$$

For each objective to have a clear priority, the weights $w_1, \ldots, w_4$ should be selected such that the ranges of values each of the objective functions can assume do not overlap with others. The actual weights depend on the use case.

7.5 Heuristic Approach

The resource demands of VCSs and the available capacity on the substrate network nodes and links change frequently, requiring quick reactions. While the optimization approach can be used with appropriate solvers to find optimal embeddings for the templates, it is too slow to be used for large problem instances. In this section, I give an overview of the heuristic approach, developed as part of my joint work with S. Schneider and H. Karl [32], which finds good solutions for the B-SPRING problem quickly, either from scratch or adapting an existing solution. S. Schneider had a major role in the implementation of this algorithm. I use this algorithm together with the optimization approach for analyzing the B-SPRING problem in Section 7.6.

Table 7.1, 7.2, and 7.3 summarize the required inputs to this problem.

The heuristic approach consists of initialization, sequential embedding, and iterative improvement steps, which I briefly describe in this section.

During the initialization step, the heuristic computes the shortest paths between all pairs of nodes in the substrate network, using an implementation of the Floyd-Warshall algorithm that favors paths with low delay and high capacity.

Afterwards, the algorithm creates an initial solution using the embedding procedure shown in Algorithm 7.1. The main workflow of the embedding procedure is similar to the U-SPRING heuristic approach (Algorithm 6.1) but we have modified it substantially to support bi-directional service flows, stateful and fixed components, and to allow instances of a component to be shared among different VCSs, if requested.

As flows are mapped to these precomputed paths, the load on the links increases, possibly resulting in over-subscription. Iterative improvements then aim at reducing or avoiding over-subscription. While the embedding procedure ensures correct embeddings and tries to optimize the embedding for the objectives defined in Section 7.2, its sequential nature can also lead to sub-optimal results. For example, when embedding bi-directional templates with stateful components, flows in the downstream direction return to the same stateful instances they traversed in the upstream direction. When data rate is propagated over the instances one by one, such cases cannot be anticipated when mapping the stateful instance in the upstream direction. The additional resource demand imposed later at the downstream direction can therefore lead to over-subscription.

Therefore, in the improvement step, the overlay of each template is modified iteratively based on tabu search [111] by picking a random instance that is neither source nor fixed and declaring it as *tabu*. The overlay is then re-created using the embedding procedure again but disallowing to place the tabu instance at the same location as before. This leads to a different distribution of instances and the load, possibly decreasing or avoiding over-subscription.

Algorithm 7.1 Main procedure of the B-SPRING heuristic

```
// Remove old overlays with no templates
if $\exists G_{\mathrm{OL}}(T)$ with $T \notin \mathcal{T}$ then
    remove $G_{\mathrm{OL}}(T)$
remove all fixed instances of all templates
for all $T \in \mathcal{T}$ do
    // Add/remove/update source instances and flows
    if $\nexists G_{\mathrm{OL}}(T)$ then
        create empty overlay $G_{\mathrm{OL}}(T)$
    for all $(c, F_{T,v}) \in S_T$ do
        if $\nexists i \in I_{\mathrm{OL}}(T)$ with $M_T^C(i) = c$ and $M_T^V(i) = v$ then
            create $i \in I_{\mathrm{OL}(T)}$ with $M_T^C(i) = c$ and $M_T^V(i) = v$
        assign/update the data rate of flow $(f, r_f)$ on $i$
    if $\exists i \in I_{\mathrm{OL}(T)}$, $M_T^C(i) \in \mathcal{C}_{\mathrm{SRC}}$, $\nexists (f, \lambda) \in F_{T,M_T^V(i)}$ for any $\lambda$ then
        remove $i$
    // Add fixed instances
    for all $(c, v) \in X_T$ do
        if $\nexists i \in I_{\mathrm{OL}(T)}$ with $M_T^C(i) = c$ and $M_T^V(i) = v$ then
            create $i \in I_{\mathrm{OL}(T)}$ with $M_T^C(i) = c$ and $M_T^V(i) = v$
    set current direction to upstream
    // Process instances in topological order according to template
    for all $i \in I_{\mathrm{OL}}(T)$ in topological order do
        if $i$ has no input data rate, $M_T^C(i) \notin \mathcal{C}_{\mathrm{FIX}}$ and $M_T^C(i) \notin \mathcal{C}_{\mathrm{SRC}}$ then
            remove $i$ and continue with the next instance
        if $M_T^C(i) \in \mathcal{C}_{\mathrm{END}}$ then
            set current direction to downstream
        compute output data rates of $i$
        for all output $k$ of $i$ in current direction do
            get arc $a$ and flows leaving $k$
            if $a \notin A_T$ then
                continue with next output
            update mapping of flows to edges
```

7.6 Evaluation

In this section, I show the evaluation results of the optimization approach, compared to the heuristic approach. As the heuristic approach achieves close-to-optimal results in a significantly shorter time than the optimization approach, we used the heuristic for further analyzing the B-SPRING problem in larger scenarios.

We have used Python implementations of the optimization and the heuristic approach and Gurobi Optimizer [21]7.0.2 as a solver for the MILP. The implementation of the approaches is available online [37]. All simulations have been performed on machines with Intel Xeon E5-2695 v3 CPUs running at 2.30 GHz and using GNU Parallel [112] to automatically assign jobs to available cores.

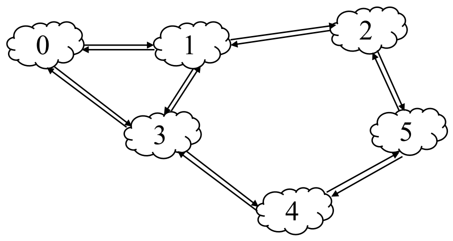


Figure 7.3: Substrate network used for evaluations

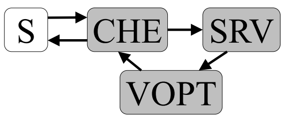


Figure 7.4: Video streaming template used for evaluations

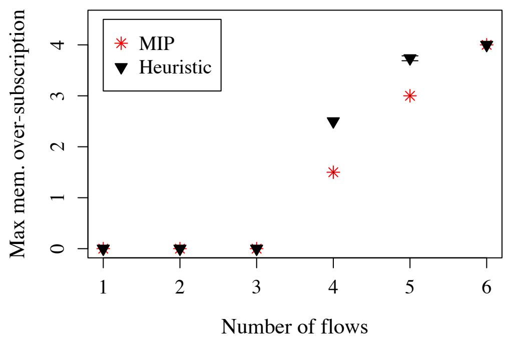
Due to the long run time of the optimization approach, for comparing the performance of both approaches, we have used a small network consisting of 6 nodes and 14 directed links with uniform capacities. Figure 7.3 shows the topology of the network, which is a subset of the Abilene network from the SNDlib test instances [94]. Additionally, we have evaluated the heuristic using much larger networks with hundreds of nodes. We have calculated the link delay $d(l)$ for each link l based on the distances between the geographical locations of the nodes.

Based on common Network Function Virtualization (NFV) use cases [113], we have chosen an example video streaming VCS as the bi-directional template to be embedded. Figure 7.4) shows this template. In this VCS, the users (represented by the source component S) request videos. The requests go through a cache (CHE) and are forwarded to the server (SRV) if they are not found in the cache. Before streaming videos from the server, they are transcoded by a video optimizer (VOPT), which reduces their data rate by 50 % [114].

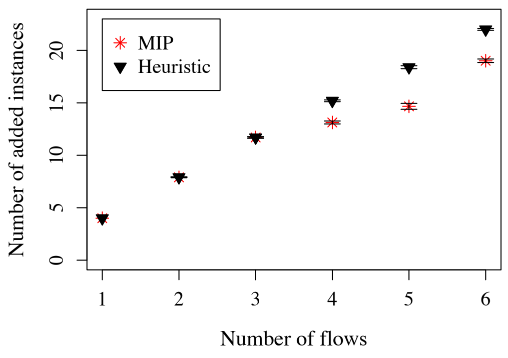
Using the substrate network and the video streaming template, we have created different problem instances with 1 to 6 flows leaving sources at randomly varying locations in the network. We have used the optimization and heuristic approaches to create initial embeddings on an empty network, minimizing the lexicographical combination of the four objectives defined in Section 7.4.2: Minimizing (1) the maximum link and node resource over-subscription (obj_1) to avoid violations of Service-Level Agreement (SLA) or even infeasible embeddings, (2) the overhead of adding or removing instances (obj_2), (3) the total resource consumption (obj_3), and (4) the total delay (obj_4).

7.6.1 Comparison of Optimization and Heuristic Approaches

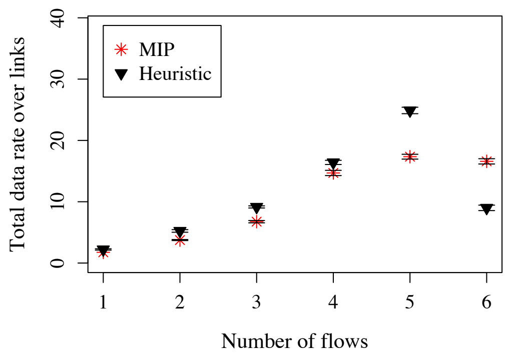
Considering obj_1, both approaches can completely avoid memory over-subscription for 1 to 3 flows by distributing the load over different instances and nodes. For problem instances with more flows, the available memory no longer suffices and is over-subscribed relative to the number of flows (Figure 7.5a). While being in the same order of magnitude, the heuristic creates embeddings with significantly



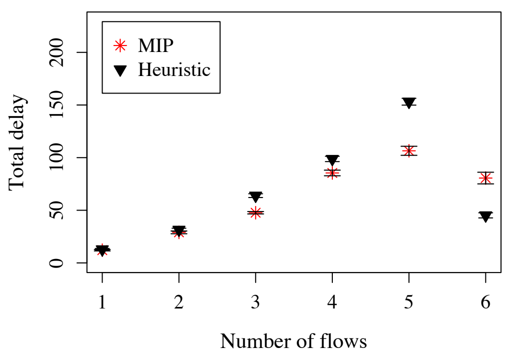
(a) Maximum memory over-subscription over all nodes



(b) Total number of added instances for all service components



(c) Total data rate over all network links



(d) Total delay

Figure 7.5: Comparison of the results delivered by optimization and heuristic approaches with increasing load

higher maximum memory over-subscription compared to the optimal results (up to 67 % higher average). The results are similar for CPU resources. In these experiments, the link capacities are never over-subscribed.

These observations are similar for the other three objectives: The heuristic can closely approximate the results of the optimization approach with just small deviations in the values of important metrics, as shown in Figure 7.5.

An exception occurs for problem instances with 6 flows; the heuristic approach cannot find solutions with as few instances as the optimization approach (against obj_2). However, with these additional instances, the paths created between the pairs of instances are shorter, with a lower data rate and smaller delay (in line with obj_3 and obj_4) than the results of the optimization approach. For the optimization approach, these objectives have a lower priority than obj_1 and obj_2.

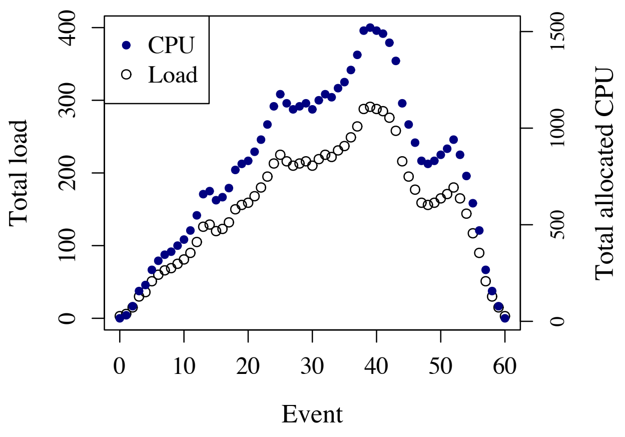
7.6.2 Scalability

Where the optimization approach needs minutes to hours to solve the problem instances, the heuristic finds solutions in milliseconds to seconds. For example, the worst-case run times for problem instances described in Section 7.6.1 with 4 flows were 137.1 hours with the optimization approach and 6.7 seconds with the heuristic approach (not the same problem instance). Using this advantage of the heuristic approach, we have further evaluated the performance of our approaches by solving additional problem instances using the heuristic.

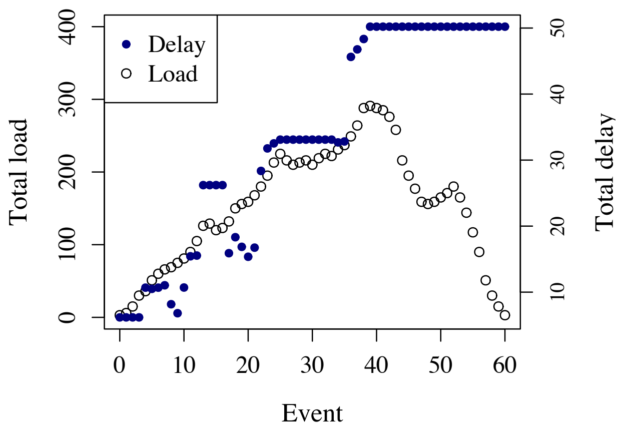
We have used the largest substrate network in the SNDlib test instances [94], consisting of 161 nodes and 664 directed links (Brain network), with 10 sources and 30 flows. The simulations start with an empty network, embedding the template shown in Figure 7.4 and adapting the embeddings based on the increasing and decreasing load in the form of 60 events. For these problems, the heuristic found embeddings in 42.31 s on average.

Figure 7.6a shows the total allocated CPUs over these events, based on the total load from all flows at all source locations. Resource allocation clearly adapts to the load; with increasing load, more CPUs are allocated to the service components and when the load decreases, the allocated resources are decreased. A similar trend can be observed for the total used link capacity in Figure 7.6c.

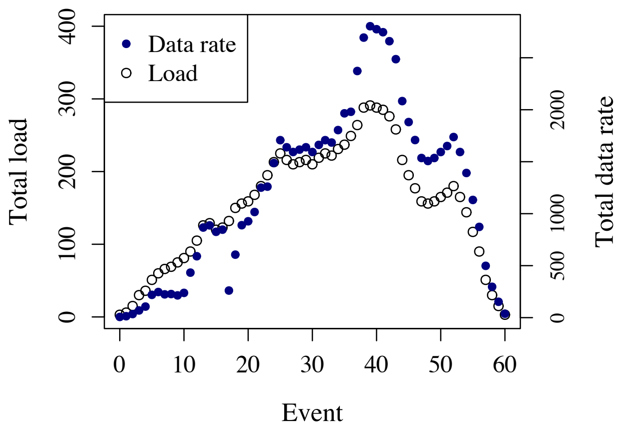
Figure 7.6b shows an important consequence of the choice of priorities for the four objectives in these simulations. The second-highest priority (after minimizing resource over-subscription) was minimizing the number of added and removed instances to limit of the costs and overheads associated with starting and stopping these instances. Minimizing the delay has the lowest priority among the objectives, as both approaches ensure that the delay remains within the maximum tolerable delay bound for each arc, as defined in the templates. Looking at the total delay for the embedded template, it can be observed that after the peak load around event 40, although the load decreases, the delay does not decrease. Adhering to the priority of minimizing the number of added/removed instances, the existing instances (possibly placed farther from the sources) are still used with a lower data rate forwarded to them, rather than removing or migrating them. This shows that reducing the delay in an already deployed service is difficult using these priorities and could be seen as an argument to increase the priority of obj_4 in a practical setting. This can be done, for example, when the situation is stable and instances can be added or removed without risk.



(a) Changes to the allocated CPU based on the changes in load



(b) Changes to the delay based on the changes in load



(c) Changes to the used link capacity based on the changes in load

Figure 7.6: Analysis of the solutions for a large problem instance with a series of events that change the overall load of the network

7.7 Conclusion

In this chapter, I have described the B-SPRING problem for joint scaling, placement, and routing pliable VCSs with bi-directional service flows. I have presented optimization and heuristic approaches to the problem that take into account upstream and downstream flows returning to their sources as well as stateful service components. Both approaches can reuse service components across different VCSs, if requested. They can also handle VCSs that consist of fixed components, like legacy PNFs that might still be needed while network softwarization is developing.

The presented MILP can be used for finding optimal embeddings in small substrate networks. For larger networks, the heuristic approach can find close-to-optimal solutions within seconds. In practice, this short execution time allows quick adaptation of the embeddings to ongoing load fluctuations in the network.

Simulation-based evaluations have shown that the quality of the solutions provided by the heuristic approach is close to the optimal results. Depending on the priorities of the service provider or the network operator, the solutions can be adjusted to deliver the required results, e.g., by using different objective functions in the MILP. I have presented the results of a configuration where over-subscription of node and link capacities is minimized. I have shown that the amount of allocated resources adapts to the load and the number of added or removed instances are kept as low as possible to avoid unnecessary costs in a frequently changing load setup.

Using the B-SPRING model, the desired structure of bi-directional services can be defined as service templates in form of directed, acyclic graphs (in each of the upstream and downstream directions) and embedded into the substrate network with the required amount of resources. The B-SPRING approaches support the vertical and horizontal scaling of service components, even if they are stateful and ensure the correct routing and processing of flows in upstream and downstream directions.

As shown in Figure 7.7, the B-SPRING problem is an extension of the U-SPRING problem (Chapter 6). A-SPRING (Chapter 3) and B-SPRING approaches allow service components to be shared among different VCSs. All three problems deal with the joint optimization of scaling, placement, and routing for VCSs.

The U-SPRING and B-SPRING problems are similar in the sense that the structure of the pliable VCSs are described using *service templates* in both cases and the service templates are embedded into the substrate network according to the location of *sources* and the load originating from them for each service. However, the U-SPRING and B-SPRING approaches differ in the following aspects.

In U-SPRING approaches, the service components required for upstream and downstream flows need to be embedded separately, requiring separate capacity planning. Using these approaches, to ensure that a certain stateful component is traversed by the same flows in both directions, the VCSs might need to be divided into smaller parts with fixed endpoints. For example, the service in Figure 5.2b should be broken into three sub-templates S^{B}→Fwl→vSrv, vSrv→ParCtrl→Fwl, and Fwl→S^{B}. These sub-templates should then be embedded sequentially in sepa-

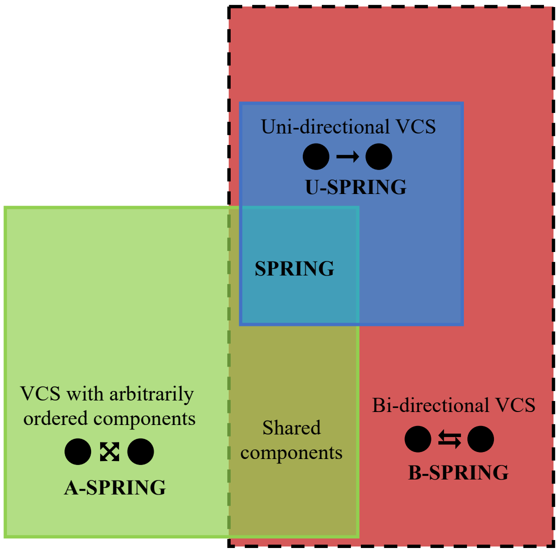


Figure 7.7: Relation of B-SPRING to A-SPRING and U-SPRING

rate steps, requiring to solve the problem multiple times with additional complexities compared to the B-SPRING approaches that embed bi-directional pliable VCSs in a single step. One important complexity is that in each step of such a sequential process, the requirements of the next steps cannot be considered, possibly resulting in more resource consumption, higher delay, and even capacity violations.

Moreover, B-SPRING approaches allow reusing instances of service components across multiple VCSs, if so required and desired. For this, the shared service components can be included (with the same unique identifier) in all of the service templates that need it. For embedding the VCSs, the shared service components across different templates are detected and handled as requested, e.g., by adapting the inputs and outputs. While the formulation of the U-SPRING approaches does not prevent this, this capability has not been integrated in the solutions presented in Chapter 6. As the B-SPRING approaches can also support uni-directional services, they can be used instead of the U-SPRING approaches if the component sharing capability is required for uni-directional VCSs.

Finally, unlike the U-SPRING approaches, the B-SPRING solution approaches can handle VCSs that are composed of virtual service components as well as PNFs with fixed locations and resources.

8 Embedding Heterogeneous Services with Load-Proportional Structures

In this chapter, I propose solutions to the problem of jointly scaling, placing, and routing heterogeneous pliable Virtualized Composed Services (VCSs) (the M-SPRING problem). In Section 8.1, I describe the model and assumptions used in the problem formulation and solution approaches. Section 8.2 and Section 8.3 include a summary of the problem formulation and description of the problem complexity, respectively. Afterwards, I present the optimization and heuristic

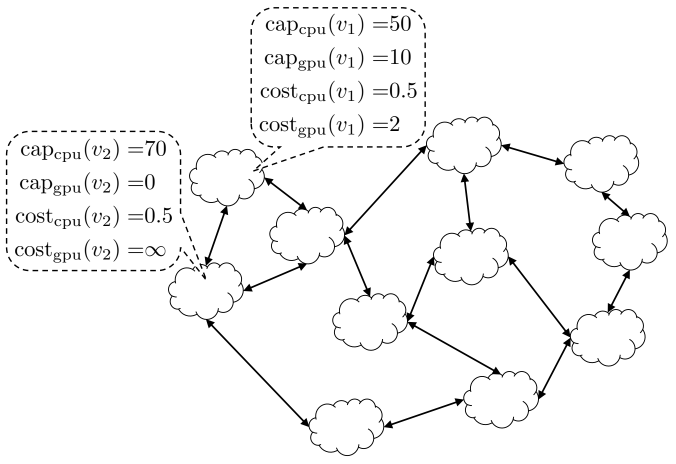

$\text{cap}_{\text{cpu}}(v_1) = 50$
$\text{cap}_{\text{gpu}}(v_1) = 10$
$\text{cost}_{\text{cpu}}(v_1) = 0.5$
$\text{cost}_{\text{gpu}}(v_1) = 2$
$\text{cap}_{\text{cpu}}(v_2) = 70$
$\text{cap}_{\text{gpu}}(v_2) = 0$
$\text{cost}_{\text{cpu}}(v_2) = 0.5$
$\text{cost}_{\text{gpu}}(v_2) = \infty$

Figure 8.1: Example substrate network. v_1 offers CPU and GPU resources but v_2 has only CPU resources.

approaches to the M-SPRING problem in Section 8.4 and Section 8.5, respectively. In Section 8.6, I evaluate the presented approaches. Section 8.7 concludes this chapter. This chapter partially includes figures and verbatim copies of the text from my paper [36].

8.1 Model

In this section, I describe the assumptions and the model and, based on them, formalize the M-SPRING problem.

8.1.1 Substrate network

The substrate network is a connected, directed graph, $G_{\text{sub}} = (V, L)$, similar to the substrate network model of the U-SPRING (Section 6.1.1) and B-SPRING (Section 7.1.1) problems, with differences in the network nodes.

Each network *node* $v \in V$ has a limited capacity of general-purpose processing resources $\text{cap}_{\text{cpu}}(v) \geq 0$ and special-purpose processing (e.g., Graphics Processing Unit (GPU)) resources $\text{cap}_{\text{gpu}}(v) \geq 0$. This can be extended to express demands for other types of resources, e.g., memory, FPGAs, etc. These resources are available at a pre-defined cost on each node. I denote the cost of using a unit of CPU and GPU resources for one time unit on node v by $\text{cost}_{\text{cpu}}(v)$ and $\text{cost}_{\text{gpu}}(v)$, respectively. If a certain resource type is not available on a node, I assume the cost of using it is infinitely large.

Figure 8.1 shows an example substrate network, including example resource capacities and costs for two nodes.

Table 8.1 summarizes the network-related parameters used in the rest of this chapter.

Table 8.1: M-SPRING Substrate Network Parameters

Symbol	Definition
$G_{\text{sub}} = (V, L)$	Substrate network graph.
$v \in V$, $l \in L$	Substrate network nodes, links.
$\text{cap}_{\text{cpu}}(v)$, $\text{cap}_{\text{gpu}}(v)$	CPU, GPU capacity of v.
$\text{cost}_{\text{cpu}}(v)$, $\text{cost}_{\text{gpu}}(v)$	Cost of a CPU, GPU unit on node v.
$\text{cap}(l)$, $d(l)$	Capacity, delay of l.

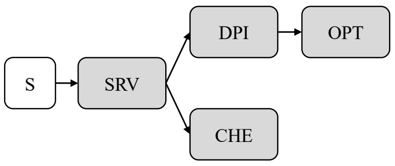


Figure 8.2: Example service template including a source, a server (SRV), a deep packet inspector (DPI), a video optimizer (OPT), and a cache (CHE).

8.1.2 Service Template

Service deployment requests are given as templates that describe the general structure of a service. Each service template is a connected, directed, acyclic graph $G_T = (C_T, A_T)$, e.g., as shown in Figure 8.2. Each *component* $c \in C_T$ in the template represents a Virtual Network Function (VNF), cloud service component, etc., possibly with different deployment versions. I describe the details of components and deployment versions in Section 8.1.3. Similar to the U-SPRING and B-SPRING models, each *arc* $a \in A_T$ of the template represents the connectivity among two components.

Table 8.2 shows a summary of the parameters related to the templates.

8.1.3 Components and Deployment Versions

Each component $c \in C_T$ has a given number of inputs n_c^{in} and outputs n_c^{out}, representing the number of ingoing and outgoing connection points, respectively. The outgoing data rate of a component depends on the data rate on all its inputs. This is calculated using a given function $\text{fout}_c(\Lambda) : \mathbb{R}_{\geq 0}^{n_c^{\text{in}}} \to \mathbb{R}_{\geq 0}^{n_c^{\text{out}}}$, where Λ is the vector of data rates on all inputs of the component.

Each component may optionally be deployable using different resource types, e.g., as a Virtual Machine (VM) version that can only use CPUs or an accelerated version that needs special-purpose processing resources (in this model, GPU) in addition to CPUs. Each such *deployment version* of a component requires a specific software version, which is made available by the service provider. Each component description must include at least one deployment version. I refer to the components with more than one deployment version as *multi-version components.*

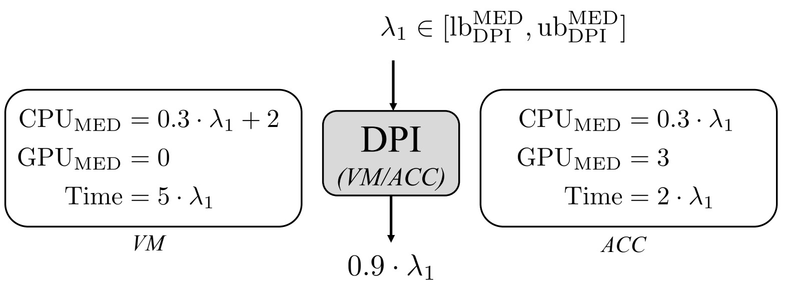
Each deployment version may consume different types and different number of resources and can result in a different cost and a different Time in System (TiS)

Table 8.2: M-SPRING Template Parameters

Symbol	Definition
$G_T = (C_T, A_T)$	Template graph.
$c \in C_T$, $a \in A_T$	Components and arcs of template T.
n_c^{in}, n_c^{out}	Number of inputs, outputs of c.
VER	Set of supported deployment versions for components: virtual machine, container, or GPU-accelerated version.
LEV	Set of load levels at components: low, medium, high, and infinite load.
$\mathrm{lb}_c^{\mathrm{lev}}(\mathrm{ver})$, $\mathrm{ub}_c^{\mathrm{lev}}(\mathrm{ver})$	Lower and upper bounds for data rates that are considered as a certain load level lev $\in$ LEV for each component c for the corresponding deployment version ver $\in$ VER.
$\mathrm{fpt}_c(\mathrm{ver}, \lambda)$	Time in system of requests using version ver $\in$ VER of component c, when the sum of incoming data rates on all its inputs equals λ.
$\mathrm{fcpu}_c^{\mathrm{lev}}(\mathrm{ver}, \Lambda)$, $\mathrm{fgpu}_c^{\mathrm{lev}}(\mathrm{ver}, \Lambda)$	CPU, GPU demands of component c at a certain load level lev $\in$ LEV, for a certain deployment version ver $\in$ VER, calculated based on Λ, the vector of data rates on inputs of c.
$\mathrm{ccon}_c^{\mathrm{lev}}(\mathrm{ver})$, $\mathrm{gcon}_c^{\mathrm{lev}}(\mathrm{ver})$	Idle CPU, GPU resource consumption of the deployment version ver $\in$ VER of component c at load level lev $\in$ LEV (the constant term of the corresponding polynomial functions $\mathrm{fcpu}_c^{\mathrm{lev}}(\mathrm{ver}, \Lambda)$), $\mathrm{fgpu}_c^{\mathrm{lev}}(\mathrm{ver}, \Lambda)$.
$\mathrm{fout}_c(\Lambda)$	Data rates on outputs of component c, calculated based on Λ, the vector of data rates on inputs of c.
src_a, dst_a	Component where arc a begins, ends.
$d_a^{\max}$	Maximum delay for a.

for the requests. I assume a resource cost model based on the usage duration of a unit of a resource in the substrate network. I describe the specification of required resource units in the rest of this section. Different versions of a component might be appropriate in different scenarios, involving a trade-off between TiS and resource costs. Available deployment versions for a component are expressed as the set VER, for example including: a VM version (VM) or a container version (CON) that only need CPUs as processing units, and an accelerated version (ACC) that needs CPUs and GPUs. To simplify the notations in the rest of this chapter, I include only these versions in the problem formulation but other deployment versions requiring other types of resources can easily be added to the model.

In practice, deploying each such version for a component requires invoking a specific Virtualized Infrastructure Manager (VIM), e.g., an OpenStack [115] instance for VMs or a Kubernetes [116] instance for containers. So, a clear pointer to the instance type in the template makes further processing steps easier.



$\lambda_1 \in [\mathrm{lb}_{\mathrm{DPI}}^{\mathrm{MED}}, \mathrm{ub}_{\mathrm{DPI}}^{\mathrm{MED}}]$
$\mathrm{CPU}_{\mathrm{MED}} = 0.3 \cdot \lambda_1 + 2$
$\mathrm{GPU}_{\mathrm{MED}} = 0$
$\mathrm{Time} = 5 \cdot \lambda_1$
VM
DPI
(VM/ACC)
$\mathrm{CPU}_{\mathrm{MED}} = 0.3 \cdot \lambda_1$
$\mathrm{GPU}_{\mathrm{MED}} = 3$
$\mathrm{Time} = 2 \cdot \lambda_1$
ACC
$0.9 \cdot \lambda_1$

Figure 8.3: Example DPI as a VM and as an accelerated version (ACC).

Each deployment version ver $\in$ VER of component c is accompanied by a function $\mathrm{fpt}_c(\mathrm{ver}, \lambda)$ that shows the *maximum* expected TiS given a total input data rate of λ. This description is, of course, only meaningful if accompanied by the specification of the attributes of the processing unit that has been used to profile the component. For simplicity, I leave out these details from the component description model. Different deployment versions can be defined for different processing unit architectures, resulting in different TiS values for a given load.

Figure 8.3 shows the Deep Packet Inspector (DPI) component from the example template of Figure 8.2, defined with two different deployment versions: a VM version and an accelerated version. For this DPI, if the input data rate is λ_1, the outgoing data rate from its only output is expected to be at most $0.9 \cdot \lambda_1$, for example because the service provider expects 10 % of video streaming requests to be unauthorized. The TiS using the VM version is at most 2.5 times more than the case where the accelerated version is used to process the same amount of load.

As shown in Section 5.2, the processing resource demands of components can have complex, non-linear relationships with the input load. To overcome computational difficulties in the problem formulation (e.g., in the Mixed-Integer Program (MIP) described in Section 8.4), I assume the CPU and GPU demands are given as piecewise linear functions that approximate non-linear dependencies of the resource demands on the load. Figure 8.4a shows how such a piecewise linear function could represent the CPU demands of an example component based on its incoming data rate.

In case a resource type like GPU cannot be shared among different processes, piecewise constant functions can be used, representing the stepwise increase of the number of required resource units by increasing load, e.g., as shown in Figure 8.4b. The piecewise linear functions $\mathrm{fcpu}_c(\mathrm{ver}, \Lambda)$, $\mathrm{fgpu}_c(\mathrm{ver}, \Lambda)$ specify the CPU and GPU demands of the deployment version ver of component c. The demand is calculated based on the vector Λ of ingoing data rates on inputs of the component.

I define the condition for selecting the right linear function to calculate the resource demand based on the *total* load that should be handled, i.e., the sum of data rates on all inputs of the component. For example, the function shown in Figure 8.4b can be expressed as in Equation 8.1, where $\lambda = \sum_{i \in n_c^{\mathrm{in}}} (\Lambda)_i$ is the sum of data rates on all inputs of the accelerated deployment version of component c and $(\Lambda)_i$ is the i-th element of vector Λ.

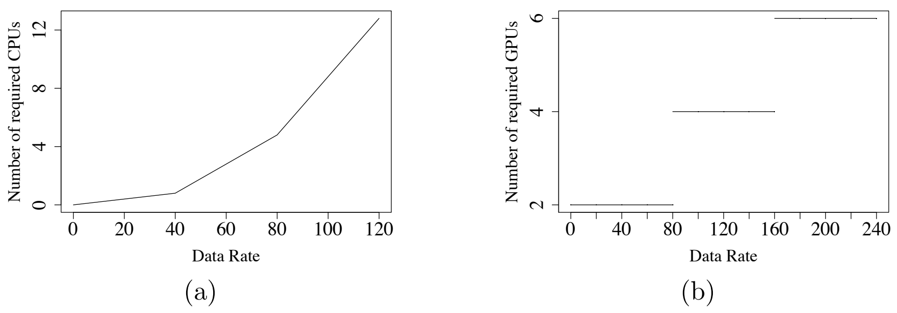


Figure 8.4: Example CPU and GPU demands based on incoming data rate

$$\mathrm{fgpu}_c(\mathrm{ACC}, \Lambda) = \begin{cases} 2 & \text{if } \lambda \in (0, 80] \\ 4 & \text{if } \lambda \in (80, 160] \\ 6 & \text{if } \lambda \in (160, 240] \\ \infty & \text{else} \end{cases} \tag{8.1}$$

For simplifying the notations, I assume the resource demands of all components are defined for the same *number* of load levels, i.e., the same number of non-overlapping intervals defining the domain of the function. I express this with a set LEV, consisting of four possible load levels, i.e., low (LOW), medium (MED), and high (HIG) that can be handled by a specific deployment version of a component and infinite load (INF) above those. As the actual data rate that is received at each input of a component is known only *after the template embedding* process, an explicit modeling of the *sum* of these input data rates in this way is necessary. This model, however, can be adapted to any arbitrary number of load levels. Infinite load is any amount of total data rate that cannot be handled efficiently by the corresponding deployment version using any reasonable (as defined by the service provider) amount of resources in a reasonable amount of time.

I use the notations $\mathrm{lb}_c^{\mathrm{lev}}(\mathrm{ver})$, $\mathrm{ub}_c^{\mathrm{lev}}(\mathrm{ver})$ to show the lower and upper bounds of a load level lev for the deployment version ver of a component c. For example, considering an accelerated version of c, if the sum of input data rates to c is between $\mathrm{lb}_c^{\mathrm{MED}}(\mathrm{ACC})$ and $\mathrm{ub}_c^{\mathrm{MED}}(\mathrm{ACC})$, its CPU and GPU demands are calculated by the linear functions $\mathrm{fcpu}_c^{\mathrm{lev}}(\mathrm{ACC}, \Lambda)$, $\mathrm{fgpu}_c^{\mathrm{lev}}(\mathrm{ACC}, \Lambda)$. The values of the lower and upper bounds are given as part of the service template, e.g., generated using service profiling methods.

The resource demand functions are described in the template. Figure 8.3 shows example functions for CPU and GPU demands of the VM and ACC versions of the DPI function, at medium load level.

Source components are special components that represent the starting point of the flows in the service, e.g., end users or content distribution servers. Source components have zero resource demands (and, therefore, no cost), no input, exactly one outgoing connection to another component with unspecified data rate, and impose no additional TiS. S is the source component of the template in Figure 8.2. Each template has exactly one source component. Several instances of

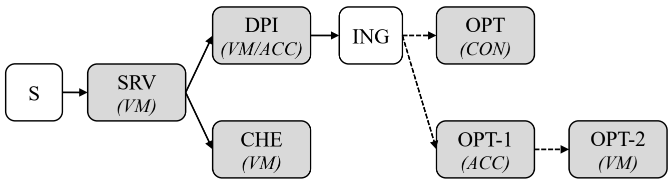


Figure 8.5: Example multi-structure video streaming service template including multi-version components.

each source component can be mapped to different nodes of the network where flows are initiated, e.g., to model different locations where users of a service are located.

Fixed components are also special components at a given location in the substrate network. Their resource demands are not considered for embedding the templates, as they are pre-defined and fixed.

Table 8.2 shows an overview of the parameters and the related symbols to the components and deployment versions.

8.1.4 Multi-Structure Templates

By specifying multiple deployment versions for components of a template, services can be deployed with the best version of each component considering the trade-offs between the TiS for the requests and the resource costs.

In a more complex scenario, a certain functionality might even be realized in different variations that consist of more than one component. For example, the video optimizer (OPT) functionality in Figure 8.5 can either be deployed as one single container or as a chain of one accelerated component and one VM that need to work together. Selecting one version or another of such a service component not only affects the resource demands of the component and the TiS for the requests but may also influence the structure of the whole VCS. I refer to such templates as *multi-structure templates*.

To support this, I introduce special *ingress* components in the model. These components have at least two outputs, among which only one output can be active. Any component of the template (except the source component) can be marked as an ingress component. For this, the component must be equipped with a load balancing or classification functionality. If no such component is available in the VCS, additional placeholder components with zero resource demands, zero cost, and zero additional TiS can be defined and added to the template. This might be needed, e.g., if version selection is required directly at the beginning of the template or for keeping the rest of the components (e.g., the DPI in Figure 8.5) in the template unaware of this optional branching. The placeholder ingress components will not be a part of the actual deployment of the service.

Multi-structure templates may include multi-version components, modeling heterogeneous services. For example, the video streaming template in Figure 8.5 includes a multi-version DPI and can take multiple structures depending on which version of the video optimizer (OPT) is selected.

8.1.5 Template Embedding

The *template embedding* process for the M-SPRING problem involves deciding:

- how many instances (*horizontal scaling*),
- of which components (*structure selection*)
- using which deployment version of each component for each instance of it (*version selection*),
- with how many resources (*vertical scaling*),
- need to be instantiated in which locations (*placement*),
- and how the traffic should be routed among them (*routing*).

The outcome is an overlay, described in Section 8.1.6.

The required inputs for this process are similar to the U-SPRING template embedding model, described in Section 6.1.3.

In addition to the description of source instances, if a template T includes fixed components, they should also be given as a part of the input. Fixed components are described as a set X_T of tuples (c, v). $c \in C_T$ shows the fixed component and $v \in V$ is the network node where it is located. Fixed components influence the embedding process of the template, e.g., because the template contains non-fixed components that can only be placed at locations with a given maximum delay to the fixed component.

Previous embeddings of components of template T are defined with a small difference to the U-SPRING model; they are given as a set P_T of tuples (c, v, ver). Such a tuple specifies that an instance of component $c \in C_T$ exists on node $v \in V$ with the deployment version ver.

Table 8.3 gives an overview of symbols and parameters related to the template embedding.

8.1.6 Overlay

The overlay model for the M-SPRING problem is similar to the overlays of the U-SPRING problem, described in Section 6.1.4. One difference is related to the fact that for each component, there can be multiple instances (in all SPRING approaches). In M-SPRING approaches, if there are several deployment versions of a component, each instance of it can have a different version, if required. For simplicity, I assume only one instance of each component can be mapped to one network node, which also means two different deployment versions of one component cannot be mapped to one network node. The M-SPRING optimization problem can, of course, be formulated differently to allow multiple instances of a component on one node. As in the U-SPRING problem, if one template is embedded multiple times, e.g., each for a different service provider, there are no limitations for embedding multiple instances of the same component from different service deployment requests into the same node.

Table 8.3: M-SPRING Template Embedding and Overlay Parameters

Symbol	Definition
$(c, v, \lambda) \in S_T$	Data rate λ of source component c from template T at node v.
$(c, v) \in X_T$	An instance of c fixed to node v.
$(c, v, \text{ver}) \in P_T$	An existing instance of component c with deployment version ver previously embedded at node v.
$\mathcal{T}$	All templates to be embedded.
$\mathcal{C} = \bigcup_{T \in \mathcal{T}} C_T$	All components from templates in $\mathcal{T}$.
$\mathcal{C}_{\text{FIX}}, \mathcal{C}_{\text{SRC}}, \mathcal{C}_{\text{ING}} \subset \mathcal{C}$	All fixed, source, ingress components.
$\mathcal{C}_N \subset \mathcal{C}$	All normal processing components that are not fixed, source, or ingress.
$\mathcal{A} = \bigcup_{T \in \mathcal{T}} A_T$	All arcs of templates in $\mathcal{T}$.
$\mathcal{S} = \bigcup_{T \in \mathcal{T}} S_T$	All sources of templates in $\mathcal{T}$.
$\mathcal{X} = \bigcup_{T \in \mathcal{T}} X_T$	All fixed components of templates in $\mathcal{T}$.
$G_{\text{OL}}(T) = (I_{\text{OL}}(T), E_{\text{OL}}(T))$	Overlay graph corresponding to template T.
$i \in I_{\text{OL}}(T)$, $e \in E_{\text{OL}}(T)$	Instances, edges of overlay.
$M_T^C(i)$	The corresponding component of instance i.
$M_T^V(i)$	The node where instance i is mapped to.
$M_T^A(e)$	The corresponding arc of edge e.

As described in Section 8.1.4, multi-structure templates might include optional branches in their structure that may not be embedded. Therefore, there may not necessarily be an edge in the overlay for each arc of the corresponding template. The maximum allowed latency for each edge is defined for its corresponding arc.

Table 8.3 shows the overlay-related parameters and additional related symbols.

8.2 Problem Formulation

As defined in Section 8.1.5, this problem involves version and structure selection, placement, scaling, and routing decisions. The inputs and outputs of the problem can be summarized as follows.

- Inputs:
  - Substrate network
  - A template for each VCS
  - Location and data rate of the sources for each VCS
  - Location and deployed version of previously embedded components (optional, can be empty)
  - Location of fixed components (optional, can be empty)

- Outputs:
  - For newly requested VCSs: An overlay mapped to the substrate network
  - For already deployed VCSs: Modified overlay and its modified mapping to the substrate network

I define a valid *system configuration* similar to the U-SPRING model defined in Section 6.2, with the additional constraint that the total delay of the paths created for an arc cannot exceed the maximum tolerable delay defined for the arc.

For the M-SPRING problem, I define the following metrics of interest:

- The total number of processing resources allocated to all instances on all nodes
- The total TiS for all overlays
- The total link capacity consumption in the network
- The total number of instances that are added, removed, re-located, or switched to another deployment version

The desired solution to this problem *minimizes* the values of these metrics. In practice, considering all four metrics is necessary for jointly serving the requirements of service providers and network operators. Service providers need their services to perform quickly and they want their costs to be as low as possible, matching the performance the service achieves. For each component, at least one instance is created, with the suitable deployment version selected based on the trade-off between the resource cost and the maximum expected TiS that would be required for processing the service requests with the specified set of resources. The objectives of the service providers can be partly achieved by minimizing the required number of processing resources and the TiS imposed by the selected deployment versions for service components.

Network operators require remaining capacity after each embedding to be able to accept additional service deployment requests. Minimizing the total number of the required processing resources contributes to minimizing the amount of used network node resources. Minimizing the total link capacity consumption results in favoring solutions that place components of a service as close as possible to each other (ideally, on the same network node, using zero link capacity). This also results in a lower total latency for the service.

Minimizing the number of instances that are added, removed, or modified (i.e., by changing the deployment version of a component on the node it was previously embedded) reduces the management and state handling overheads and contributes to keeping the running services as stable as possible.

Over-subscription of node and link capacities is not allowed in this problem, as the version and structure selection behavior of the M-SPRING solutions can be influenced if there are no hard capacity limits. For example, instead of switching to a more expensive deployment version to be able to keep the TiS for service requests in an acceptable level, the algorithm might choose to over-subscribe the previously used resources to avoid changes. This is not desirable, as it does not allow observing the actual power of defining heterogeneous pliable VCSs.

8.3 Problem Complexity

Using polynomial-time reduction, I show that for an instance of the M-SPRING problem deciding whether a solution with no violation of capacity constraints exists is an NP-complete problem. It is possible in time polynomial in the size of inputs of the problem to check whether the embedding is valid. The output of the problem has also a polynomial relation to the size of the problem inputs. Therefore, the problem is in NP.

This problem is an extension to the U-SPRING problem, which is proven to be NP-Complete [34]. Given an instance of U-SPRING, I construct an instance of the current problem as follows.

I assume every component in every template to be embedded has exactly one deployment version that only consumes CPUs, e.g., a VM version and has zero memory demand. I also assume the templates have only one possible structure. I set the TiS for the service requests using all components to zero and the maximum tolerable delay for each arc to infinity, as the U-SPRING problem formulation does not include arc delays. For a similar reason, I assume the templates do not include any fixed components. I can complete the inputs to the M-SPRING problem using the rest of the input provided for an instance of the U-SPRING. A solution without violation of capacity constraints for the U-SPRING problem is then also a valid solution for the current problem and vice versa.

The reduction can be performed in polynomial time in the size of the problem input. Therefore, the M-SPRING problem is NP-hard. From that, I can conclude that the problem is NP-complete.

8.4 Optimization Approach

In this section, I formalize the SPRING problem for heterogeneous services as an MIP. All constraints and objective functions in this formulation are linear or can be linearized. As the link and node resource demands are also specified using piecewise linear functions, we are dealing with a Mixed-Integer Linear Program (MILP).

Tables 8.4 and 8.5 show an overview of the binary and continuous decision variables in the MILP, respectively. Input parameters of this problem are summarized in Tables 8.1, 8.2, and 8.3.

I define the parameters $m^*_{c,v,\text{ver}}$ to capture previous embeddings of components. For every component c that was previously embedded into node v with version ver, represented by a tuple $(c, v, \text{ver}) \in P_T$ (Section 8.1.5), I set $m^*_{c,v,\text{ver}}$ to 1. For all other components, nodes, and versions, I set it to 0. $\mathcal{M}$ represents a constant that is sufficiently large, used in the so-called Big-M formulations. I represent the

Table 8.4: M-SPRING Binary Decision Variables

Variable	Definition
$x_{c,v}$	1 iff an instance of c is mapped to v.
$m_{c,v,\text{ver}}$	1 iff an instance of c is mapped to v with version ver.
$\delta_{c,v}$	1 iff $m_{c,v,\text{ver}} \neq m^*_{c,v,\text{ver}}$, i.e., an instance of c is added, removed, or switched to another version at v.
$l^{\text{cpu}}_{c,v,\text{ver},\text{lev}}$, $l^{\text{gpu}}_{c,v,\text{lev}}$	Helper variable for calculating the CPU, GPU demand, which indicates whether the sum of data rates on the inputs of c on v is larger than or equal to the lower bound that defines the load level lev for version ver.
$u^{\text{cpu}}_{c,v,\text{ver},\text{lev}}$, $u^{\text{gpu}}_{c,v,\text{lev}}$	Helper variable for calculating the CPU, GPU demand, which indicates whether the sum of data rates on the inputs of c on v is smaller than or equal to the lower bound that defines the load level lev for version ver.
$b^{\text{cpu}}_{c,v,\text{ver},\text{lev}}$, $b^{\text{gpu}}_{c,v,\text{lev}}$	Helper variable for calculating the CPU, GPU demand, which indicates whether the sum of data rates on the inputs of c on v is in the range that defines the load level lev for version ver.
$r_{c,v,k}$	1 iff output k of the instance of ingress component c at v is activated.
$u_{a,v,v',l}$	1 iff the link is used for the path created for arc a with source and destination on v and v'.

k-th element of a vector W by $(W)_k$. $\underline{0}$ is a zero vector of appropriate length.

8.4.1 Constraints

In this section, I describe the constraints for the M-SPRING optimization approach.

Mapping Consistency Rules

$$\forall c \in \mathcal{C}_{\text{SRC}}, \forall v \in V: \qquad x_{c,v} = \begin{cases} 1 & \exists (v,c,\mu) \in \mathcal{S} \\ 0 & \text{else} \end{cases} \tag{8.2}$$

$$\forall v \in V, \forall c \in \mathcal{C}_{\text{FIX}}: \qquad x_{c,v} = \begin{cases} 1 & \text{if } \exists (c,v) \in \mathcal{X} \\ 0 & \text{else} \end{cases} \tag{8.3}$$

$$\forall c \in \mathcal{C}_{\text{SRC}}, \forall v \in V: \qquad \text{out}_{c,v} = \begin{cases} \mu & \exists (v,c,\mu) \in \mathcal{S} \\ 0 & \text{else} \end{cases} \tag{8.4}$$

I assign fixed components and sources to their pre-defined locations (Constraint 8.2, 8.3). The data rate of each source is assigned to its output (Constraint 8.4).

I track the added/removed/modified instances (Constraint 8.5). If an instance is created, the right number of inputs (Constraint 8.6) and outputs (Constraint 8.7)

Table 8.5: M-SPRING Continuous Decision Variables

Variable	Definition
$\text{cpu}_{c,v}$, $\text{gpu}_{c,v}$	CPU, GPU demand of the instance of c if mapped to v.
$\text{time}_{c,v}$	TiS for an instance of c if mapped to v.
$p_{c,v,\text{ver}}$	Potential CPU demand of c at v for version ver.
$g_{c,v}$	Potential GPU demand of c at v, defined for its GPU-accelerated instances.
$t_{c,v,\text{ver}}$	Potential TiS for c at v using version ver.
$s_{c,v}$	Total CPU and GPU resource cost of c on v per time unit.
$\text{in}_{c,v}$	Vector of length n_c^{in} of data rates at inputs of the instance of c at v, or an all-zero vector
$\text{out}_{c,v}$	Vector of length n_c^{out} of data rates at outputs of the instance of c at v, or an all-zero vector
$o_{c,v}$	Vector of length n_c^{out} of potential data rates at outputs of the instance of ingress component c at v
$\text{dr}^e_{a,v,v'}$	Data rate of the edge corresponding to an arc a that connects an instance of c at v to an instance of c' at v'.
$\text{dr}^l_{a,v,v',l}$	Data rate on link l corresponding to an arc a that connects an instance of c at v to an instance of c' at v'.

are created for it. At most one instance of each component can be mapped to a node (Constraint 8.8, 8.9).

$$\forall c \in \mathcal{C}_N, \forall v \in V: \qquad \delta_{c,v} = \begin{cases} m_{c,v,\text{ver}} & \text{if } m^*_{c,v,\text{ver}} = 0 \\ 1 - m_{c,v,\text{ver}} & \text{if } m^*_{c,v,\text{ver}} = 1 \end{cases} \tag{8.5}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n_c^{\text{in}}]: \qquad (\text{in}_{c,v})_k \le \mathcal{M} \cdot x_{c,v} \tag{8.6}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n_c^{\text{out}}]: \qquad (\text{out}_{c,v})_k \le \mathcal{M} \cdot x_{c,v} \tag{8.7}$$

$$\forall c \in \mathcal{C}_N, \forall v \in V: \qquad \sum_{\text{ver} \in \text{VER}} m_{c,v,\text{ver}} \le 1 \tag{8.8}$$

$$\forall c \in \mathcal{C}_N, \forall v \in V: 0 \le |\,\text{VER}\,| \cdot x_{c,v} - \sum_{\text{ver} \in \text{VER}} m_{c,v,\text{ver}} \le |\,\text{VER}\,| - 1 \tag{8.9}$$

Flow and Data Rate Rules

$$\forall c \in \mathcal{C}_N \setminus \mathcal{C}_{\text{ING}}, \forall v \in V: \qquad \text{out}_{c,v} = \text{fout}_c(\text{in}_{c,v}) - (1 - x_{c,v}) \cdot \text{fout}_c(\underline{0}) \tag{8.10}$$

$$\forall c \in \mathcal{C}_{\text{ING}}, \forall v \in V: \qquad o_{c,v} = \text{fout}_c(\text{in}_{c,v}) - (1 - x_{c,v}) \cdot \text{fout}_c(\underline{0}) \tag{8.11}$$

$$\forall c \in \mathcal{C}_{\text{ING}}, \forall v \in V: \qquad \text{out}_{c,v} = r_{c,v,k} \cdot o_{c,v} \tag{8.12}$$

$$\forall c \in \mathcal{C}_{\text{ING}}, \forall v \in V: \qquad \sum_{k \in [1, n_c^{\text{out}}]} r_{c,v,k} = 1 \tag{8.13}$$

The data rate entering an instance determines its outgoing data rate (Constraint 8.10). The data rates are set only if the instance is mapped to a certain node. I assume all deployment versions for an instance use the same function for calculating the outgoing data rate. This realizes the assumption that all functions for data rate, CPU, and GPU demands of a component are specifically created for that component (e.g., using a profiling system), based on target performance metrics, including throughput. These functions can be used to calculate the amount of resources required for each instance to perform at the specified performance level under the current input data rate. For ingress components, only one of the outputs can have a data rate (Constraint 8.11–8.13).

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n_c^{\text{in}}] : \quad (\text{in}_{c,v})_k = \sum_{\substack{a \in \mathcal{A} \text{ ends in input } k \text{ of } \text{src}_a(a), \\ v' \in V}} \text{dr}^e_{a,v',v} \tag{8.14}$$

$$\forall c \in \mathcal{C}, \forall v \in V, \forall k \in [1, n_c^{\text{out}}] : \quad (\text{out}_{c,v})_k = \sum_{\substack{a \in \mathcal{A} \text{ starts at output } k \text{ of } \text{src}_a(a), \\ v' \in V}} \text{dr}^e_{a,v,v'} \tag{8.15}$$

I assign a data rate to each input of the instances on each node, which is calculated as the sum of data rates of the overlay edges that end in that input (Constraint 8.14). Similarly, I assign a data rate to the outputs of the instances (Constraint 8.15).

$$\forall a \in \mathcal{A}, \forall v, v_1, v_2 \in V :$$

$$\sum_{vv' \in L} \text{dr}^l_{a,v_1,v_2,vv'} - \sum_{v'v \in L} \text{dr}^l_{a,v_1,v_2,v'v} = \begin{cases} 0 & \text{if } v{\neq}v_1, v{\neq}v_2 \\ 0 & \text{if } v{=}v_1{=}v_2 \\ \text{dr}^e_{a,v_1,v_2} & \text{if } v{=}v_1, v_1{\neq}v_2, a{\in}\mathcal{A} \end{cases} \tag{8.16}$$

$$\forall a \in \mathcal{A}, \forall v_1, v_2 \in V, \forall l \in L : \quad \text{dr}^l_{a,v_1,v_2,l} \leq \mathcal{M} \cdot u_{a,v_1,v_2,l} \tag{8.17}$$

$$\forall a \in \mathcal{A}, \forall v_1, v_2 \in V, \forall l \in L : \quad u_{a,v_1,v_2,l} \leq \text{dr}^l_{a,v_1,v_2,l} \tag{8.18}$$

$$\forall a \in \mathcal{A}, \forall v_1, v_2 \in V : \quad \sum_{l \in L} u_{a,v_1,v_2,l} \cdot d(l) \leq d_a^{\max} \tag{8.19}$$

The data rates of the edges are mapped to network links, ensuring flow conservation over the path(s) (Constraint 8.16). The total delay of the used network links must not exceed the maximum delay (Constraint 8.17–8.19).

Calculation of Resource Consumption

$$\forall c \in \mathcal{C} \setminus \mathcal{C}_{\text{SRC}}, \forall v \in V, \forall \, \text{ver} \in \text{VER}, \forall \, \text{lev} \in \text{LEV} : \quad \text{ub}_c^{\text{lev}}(\text{ver}) - \sum_{k \in [1, n_c^{\text{in}}]} (\text{in}_{c,v})_k \leq \mathcal{M} \cdot u^{\text{cpu}}_{c,v,\text{ver},\text{lev}} \tag{8.20}$$

$$\forall c \in \mathcal{C} \setminus \mathcal{C}_{\text{SRC}}, \forall v \in V, \forall \,\text{ver} \in \text{VER}, \forall \,\text{lev} \in \text{LEV} :$$
$$\sum_{k \in [1, n_c^{\text{in}}]} (\text{in}_{c,v})_k - \text{lb}_c^{\text{lev}}(\text{ver}) \leq \mathcal{M} \cdot l^{\text{cpu}}_{c,v,\text{ver},\text{lev}} \tag{8.21}$$

$$\forall c \in \mathcal{C} \setminus \mathcal{C}_{\text{SRC}}, \forall v \in V, \forall \,\text{ver} \in \text{VER} :$$
$$\sum_{\text{lev} \in \text{LEV}} (l^{\text{cpu}}_{c,v,\text{ver},\text{lev}} + u^{\text{cpu}}_{c,v,\text{ver},\text{lev}}) = |\,\text{LEV}\,| + 1 \tag{8.22}$$

$$\forall c \in \mathcal{C} \setminus \mathcal{C}_{\text{SRC}}, \forall v \in V, \forall \,\text{ver} \in \text{VER}, \forall \,\text{lev} \in \text{LEV} :$$
$$0 \leq l^{\text{cpu}}_{c,v,\text{ver},\text{lev}} + u^{\text{cpu}}_{c,v,\text{ver},\text{lev}} - 2 \cdot b^{\text{cpu}}_{c,v,\text{ver},\text{lev}} \leq 1 \tag{8.23}$$

$$\forall c \in \mathcal{C} \setminus \mathcal{C}_{\text{SRC}}, \forall v \in V, \forall \,\text{ver} \in \text{VER}, \forall \,\text{lev} \in \text{LEV} :$$
$$\begin{aligned} p_{c,v,\text{ver}} + \mathcal{M} \cdot (1 - b^{\text{cpu}}_{c,v,\text{ver},\text{lev}}) \geq \\ \text{fcpu}_c^{\text{lev}}(\text{ver}, \text{in}_{c,v}) - (1 - m_{c,v,\text{ver}}) \cdot \text{ccon}_c^{\text{lev}}(\text{ver}) \end{aligned} \tag{8.24}$$

The data rate on inputs of each instance is used for calculating its *minimum* resource demands. For selecting the right piece of the piecewise linear function, I determine the load level for every potential deployment version. For this, I compare the sum of all input data rates of the instance to the pre-defined upper and lower bounds for each load level (Constraint 8.20, 8.21). Exactly one load level is indicated as the right one (Constraint 8.22, 8.23). Based on the load level, I calculate the *potential* minimum CPU demand of each potential version (Constraint 8.24).

I repeat the same process to determine the *potential* minimum GPU resource demands (Constraint 8.25–8.29). To reduce the number of decision variables, I calculate the resource demands only for the resource types that are actually specified in the templates. For example, in the case of a VM deployment version, I set the actual GPU resource demand of it to 0, without creating the intermediate variables holding the potential GPU demands of it.

$$\forall c \in \mathcal{C}_N, \forall v \in V, \forall \,\text{lev} \in \text{LEV} :$$
$$\text{ub}_c^{\text{lev}}(\text{ACC}) - \sum_{k \in [1, n_c^{\text{in}}]} (\text{in}_{c,v})_k \leq \mathcal{M} \cdot u^{\text{gpu}}_{c,v,\text{lev}} \tag{8.25}$$

$$\forall c \in \mathcal{C}_N, \forall v \in V, \forall \,\text{lev} \in \text{LEV} :$$
$$\sum_{k \in [1, n_c^{\text{in}}]} (\text{in}_{c,v})_k - \text{lb}_c^{\text{lev}}(\text{ACC}) \leq \mathcal{M} \cdot l^{\text{gpu}}_{c,v,\text{lev}} \tag{8.26}$$

$$\forall c \in \mathcal{C}_N, \forall v \in V: \quad \sum_{\text{lev} \in \text{LEV}} (l^{\text{gpu}}_{c,v,\text{lev}} + u^{\text{gpu}}_{c,v,\text{lev}}) = |\,\text{LEV}\,| + 1 \tag{8.27}$$

$$\forall c \in \mathcal{C}_N, \forall v \in V, \forall\,\text{lev} \in \text{LEV}: \quad 0 \leq l^{\text{gpu}}_{c,v,\text{lev}} + u^{\text{gpu}}_{c,v,\text{lev}} - 2 \cdot b^{\text{gpu}}_{c,v,\text{lev}} \leq 1 \tag{8.28}$$

$$\forall c \in \mathcal{C}_N, \forall v \in V, \forall\,\text{lev} \in \text{LEV}: \quad g_{c,v} + \mathcal{M} \cdot (1 - b^{\text{gpu}}_{c,v,\text{lev}}) \geq \text{fgpu}^{\text{lev}}_c(\text{ver}, \text{in}_{c,v}) - (1 - m_{c,v,\text{ACC}}) \cdot \text{gcon}^{\text{lev}}_c(\text{ver}) \tag{8.29}$$

$$\forall c \in \mathcal{C} \setminus \mathcal{C}_{\text{SRC}}, \forall v \in V, \forall\,\text{lev} \in \text{LEV}: \quad g_{c,v} + \mathcal{M} \cdot (1 - b^{\text{gpu}}_{c,v,\text{lev}}) \geq \text{fgpu}^{\text{lev}}_c(\text{ACC}, \text{in}_{c,v}) - (1 - m_{c,v,\text{ACC}}) \cdot \text{gcon}^{\text{lev}}_c \tag{8.30}$$

Among the potential versions, only one version may be mapped to a potential location. The resource demands of the optimal versions of components (according to the objectives) are assigned as their final resource demands on the optimal nodes (Constraint 8.31, 8.32). Link and node resource consumption must not be larger than the capacity (Constraint 8.33–8.35).

$$\forall c \in \mathcal{C}_N, \forall v \in V: \quad \text{cpu}_{c,v} = \sum_{\text{ver} \in \text{VER}} p_{c,v,\text{ver}} \cdot m_{c,v,\text{ver}} \tag{8.31}$$

$$\forall c \in \mathcal{C}_N, \forall v \in V: \quad \text{gpu}_{c,v} = g_{c,v} \cdot m_{c,v,\text{ACC}} \tag{8.32}$$

$$\forall c \in \mathcal{C}, \forall v \in V: \quad \sum_{c \in \mathcal{C}} \text{cpu}_{c,v} \leq \text{cap}_{\text{cpu}}(v) \tag{8.33}$$

$$\forall c \in \mathcal{C}, \forall v \in V: \quad \sum_{c \in \mathcal{C}} \text{gpu}_{c,v} \leq \text{cap}_{\text{gpu}}(v) \tag{8.34}$$

$$\forall l \in L: \quad \sum_{a \in \mathcal{A}, v, v' \in V} \text{dr}^l_{a,v,v',l} \leq \text{cap}(l) \tag{8.35}$$

The *potential* maximum TiS for each instance of each component is calculated using the given functions if a version is mapped to a node (Constraint 8.36,8.37). The *actual* maximum TiS imposed by each component is decided based on the selected version at the target node (Constraint 8.38).

$$\forall c \in \mathcal{C}_N, \forall v \in V, \forall\,\text{ver} \in \text{VER}: \quad t_{c,v,\text{ver}} = \text{fpt}_c(\text{ver}, \sum_{k \in [1, n^{\text{in}}_c]} (\text{in}_{c,v})_k) - (1 - m_{c,v,\text{ver}}) \cdot \text{fpt}_c(\text{ver}, \sum_{k \in [1, n^{\text{in}}_c]} (\text{in}_{c,v})_k) \tag{8.36}$$

$$\forall c \in \mathcal{C}_N, \forall v \in V, \forall \text{ver} \in \text{VER}: \qquad t_{c,v,\text{ver}} \leq \mathcal{M} \cdot m_{c,v,\text{ver}} \tag{8.37}$$

$$\forall c \in \mathcal{C}_N, \forall v \in V: \qquad \text{time}_{c,v} = \sum_{\text{ver} \in \text{VER}} t_{c,v,\text{ver}} \tag{8.38}$$

I calculate the total CPU and GPU resource cost of every embedded instance on their target nodes per time unit (Constraint 8.39). By multiplying this value and the TiS of service flows at each component, the total resource usage cost of each component on each node can be calculated.

$$\forall c \in \mathcal{C}_N, \forall v \in V : s_{c,v} = \text{cpu}_{c,v} \cdot \text{cost}_{\text{cpu}}(v) + \text{gpu}_{c,v} \cdot \text{cost}_{\text{gpu}}(v) \tag{8.39}$$

8.4.2 Optimization Objective

Based on the problem formulation in Section 8.2, I define the following objective functions for the MILP:

- obj_1: Minimize the total compute resource cost

$$\text{min.} \sum_{c \in \mathcal{C}, v \in V} s_{c,v}$$

- obj_2: Minimize the total TiS for service requests

$$\text{min.} \sum_{c \in \mathcal{C}, v \in V} \text{time}_{c,v}$$

- obj_3: Minimize network resource consumption

$$\text{min.} \sum_{a \in \mathcal{A}, v, v' \in V, l \in L} \text{dr}^{l}_{a,v,v',l}$$

- obj_4: Minimize the number of added, removed, modified instances

$$\text{min.} \sum_{c \in \mathcal{C}, v \in V} \delta_{c,v}$$

To combine the benefits of using these objective functions, I define their lexicographical combination as follows:

$$\text{min. } w_1 \cdot \text{obj}_1 + w_2 \cdot \text{obj}_2 + w_3 \cdot \text{obj}_3 + w_4 \cdot \text{obj}_4$$

For the objectives to have a clear priority, the weights $w_1, \ldots, w_4$ should be selected such that the range of the values that different objective functions can take do not overlap. The desired priority among these objectives depends on the use case.

Algorithm 8.1 Main procedure of the M-SPRING heuristic

```
1:  // Remove old overlays with no templates
2:  if $\exists G_{\mathrm{OL}}(T)$ with $T \notin \mathcal{T}$ then
3:      remove $G_{\mathrm{OL}}(T)$
4:  remove all fixed instances of all templates
5:  for all $T \in \mathcal{T}$ do
6:      // Add/remove/update source instances and data rates
7:      if $\nexists G_{\mathrm{OL}}(T)$ then
8:          create empty overlay $G_{\mathrm{OL}}(T)$
9:      for all $(c, v, \lambda) \in S_T$ do
10:         if $\nexists i \in I_{\mathrm{OL}}$ with $M_T^C(i) = c$ and $M_T^V(i) = v$ then
11:             create instance $i \in I_{\mathrm{OL}}$ with $M_T^C(i) = c$ and $M_T^V(i) = v$
12:         set/update output data rate of $i$
13:     if $\exists i \in I_{\mathrm{OL}(T)}$, $M_T^C(i) \in \mathcal{C}_{\mathrm{SRC}}$, $\nexists(c, v, \lambda) \in S_T$ for any $\lambda$ then
14:         remove $i$
15:     // Add fixed instances
16:     for all $(c, v) \in X_T$ do
17:         if $\nexists i \in I_{\mathrm{OL}(T)}$ with $M_T^C(i) = c$ and $M_T^V(i) = v$ then
18:             create $i \in I_{\mathrm{OL}(T)}$ with $M_T^C(i) = c$ and $M_T^V(i) = v$
19:     // Process instances in topological order according to template
20:     for all $i \in I_{\mathrm{OL}}$ in topological order do
21:         if $i$ has no input data rate, $M_T^C(i) \notin \mathcal{C}_{\mathrm{FIX}}$ and $M_T^C(i) \notin \mathcal{C}_{\mathrm{SRC}}$ then
22:             remove $i$ and go to next iteration
23:         compute output data rates of $i$
24:         for all output $k$ of $i$ do
25:             set/update output data rate of $i$
```

8.5 Heuristic Approach

In Section 8.4, I describe the MILP formulation of the M-SPRING problem that can be used with an appropriate solver to find the optimal solution for small problem instances. In this section, I present a heuristic that quickly finds solutions and can be used for larger scenarios. This algorithm can be used for initial embedding of templates in a substrate network as well as for adapting existing embeddings.

Algorithm 8.1 shows an overview of the main procedure, which is similar to those of the U-SPRING (Algorithm 6.1) and B-SPRING (Algorithm 7.1) heuristics. I describe the important steps specific to the M-SPRING heuristic in the rest of this section.

The algorithm first processes the templates that were previously embedded but need to be removed (lines 2–3). It also removes all fixed instances from all existing overlays, to insert them correctly later on, if still needed. Afterwards, it goes through the templates to be added or modified (line 5). The templates can be sorted beforehand, e.g., according to the total input data rate from their sources.

For new templates, the algorithm creates an empty overlay (lines 7–8). It then processes the source and fixed instances for the template (lines 9–18).

Setting the output data rate of an instance i results in creating/updating outgoing edges from the output(s) of i as well as the inputs of the instances where these edge are destined. For this, the algorithm must decide how many instances of which versions of the subsequent component need to be created on which nodes. I describe this with an example.

For the example template shown in Figure 8.5 (page 127), after assigning the output data rate of instances of S, the algorithm needs to create at least one instance of SRV, which will receive the traffic from S. For every deployment version of SRV (in this example, SRV has only a VM deployment version), it looks for potential nodes. As one of objectives (described in Section 8.2) is to minimize the number of added/removed instances, the algorithm takes a greedy decision; it tries to create an instance of SRV with the maximum possible input data rate. If the total outgoing data rate of S is higher than the upper bound of the load level HIG for SRV, it creates an instance of SRV and sets its input data rate to this highest possible value. For the remaining data rate from S, it creates additional instances of SRV in the same way, until there is no more traffic left to be forwarded to a SRV instance.

At the same time, it selects the candidate nodes that can host the created instances. These nodes must have enough capacity and there must be a path to them from the node where S is located. The links over the path must have enough capacity and the total delay of the path must not be larger than the maximum tolerable delay defined in the service template. Locations with an existing instance of SRV from a previous embedding are also considered. Among the candidate nodes the algorithm can now select the best option, considering the resulting TiS of the deployment version at that load level and the resource usage cost on that node.

It then iterates over the instances of the overlay in a topological order (Line 20). That is, each instance i is processed only after all instances that have an edge to i (as specified in the template) have already been processed. The first instances that are processed are the instances after the source instance that are created while setting the output of the source instance in line 18 (instances of SRV in the previous example).

Next, it computes and propagates the data rates from outputs of the current instance towards other components (lines 23–25). In the previous example, this is the data rate towards DPI and CHE. If the data rate needs to be increased (i.e., if there are no previous embeddings of DPI or CHE, or if the previous data rate was less than the computed data rate at this step), it proceeds in the same way as described for outputs of the source instance. If the data rate needs to be decreased, a similar process is required to select the most suitable instances of the subsequent components that should get a lower data rate. To limit the range of required modifications, the algorithm selects an outgoing edge that has the smallest data rate larger than or equal to the data rate that should be decreased.

If the current instance is an instance of an ingress component, the algorithm first calculates the most suitable embedding for all of its outgoing branches. This is done sequentially for different branches. Then, comparing the total cost of each branch (calculated as the total resource usage cost during the total TiS), the cheapest branch is added to the overlay and the other branches are discarded.

8.6 Evaluation

The solution approach to the M-SPRING problem has a similar nature to those of the U-SPRING and B-SPRING problems. They all receive an abstract template (with different requirements and specifications in each problem) and embed the template into the substrate network, by scaling, placing, and selecting paths for the components and arcs in a single step. I have evaluated the U-SPRING and B-SPRING approaches from different aspects; e.g., I have shown that the amount of resources allocated to services follows the variations of load very closely (Section 6.6). I have also analyzed the scalability of the solutions as well as the behavior of the heuristic approach in large substrate networks (Section 7.6). Those results, without considering the technicalities of heterogeneous services, also apply to the approaches presented in this chapter. Therefore, in this section, I focus on evaluating the new aspects of the joint scaling, placement and routing problem that are specific to heterogeneous services.

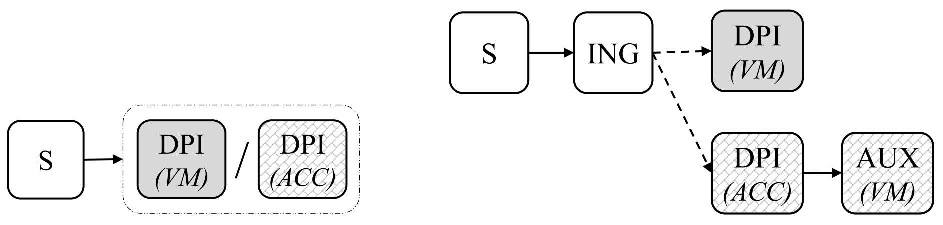
For evaluating the optimization and heuristic approaches, I have used a Python implementation of both of them. The implementation of the approaches is available online [37]. For solving the MILP, I have used the Gurobi Optimizer [21] 8.0.1. I have used the benchmarks from Inführ and Raidl [110] for the Virtual Network Mapping Problem (VNMP) [109] to build the topology of the substrate networks.

I present the results of two different experiments to compare how the heuristic solves the problem compared to the optimization approach and to show the scalability of the heuristic approach.

8.6.1 Comparison of Optimization and Heuristic Approaches

For the first set of experiments, to achieve results in a reasonable time with the optimization approach, I have used a simple template T_1 with the structure shown in Figure 8.6a. It consists of a source component and a DPI in different versions. I have set the resource demands and the resulting TiS based on the findings of Araújo et al. [117]. They have analyzed DPI VNFs with and without using GPU acceleration. Following their results, I assume an ACC version performs 20 times faster than the VM version. As I only focus on compute resource demand for simplicity, I assume other resources like memory, buffer, or disk space can be adjusted as needed similar to the compute resources.

I have used the smallest substrate network from VNMP benchmarks with 20 nodes and 44 links (substrate graph eu_20_0_prob [109]), with uniform capacities and resource costs over the network. I have set the GPU capacity of each node to 5 times less than its CPU capacity and the GPU usage cost per time unit on the same node to 50 times more than the CPU usage cost. These ratios *roughly* follow the Amazon EC2 On-Demand Pricing model [118]. Concrete information



(a) Service template T_1 (b) Service template T_2

Figure 8.6: Example heterogeneous service templates

about resource unit prices cannot be extracted from these models, as the pricing model is based on pre-defined instances with a certain group of resources reserved for them.

Figure 8.7 shows the results of the first set of experiments. In these experiments, I have increased the data rate flowing from the only source instance of the template from 1 to 70. I have captured the values of different metrics for the following cases:

1. Heuristic approach, considering both versions of DPI
2. Optimization approach, considering both versions of DPI
3. Optimization approach, considering only the VM version of DPI
4. Optimization approach, considering only the ACC version of DPI

To see the behavior of the algorithms in different load situations, I have embedded the template with different source data rates without considering the previous embedding. The results of the optimization approach have been calculated to optimality without a time limit for the solver.

As shown in Figure 8.7a, the heuristic approach starts selecting accelerated versions of the DPI early on. This is because of its greedy decision process that tries to push as much of the input data rate as possible to the first instance it creates at each step. As the instances are created one by one without considering the whole template, the required instances in the next steps cannot be considered. For this reason, the heuristic creates embeddings that are even more expensive than the ACC-only experiments with the optimization approach, as Figure 8.7b illustrates. In exchange, as shown in Figure 8.7c, the created overlays result in a very low TiS, making the heuristic approach favorable for time-sensitive VCSs.

The optimization approach creates more balanced results, favoring low-cost solutions for a larger range of input data rates. The cost of the solutions found by the optimization approach that can use both versions of the DPI lie between those of the VM-only and ACC-only experiments. Above the source data rate of 35 units, this approach starts creating more ACC versions of the DPI (in addition to VM versions) as the load increases, which consume GPUs, as shown in Figure 8.7d. This results in the gradual increase in the cost that can be observed in Figure 8.7b and the decrease in the TiS that is shown in Figure 8.7c. There is only a minor increase in the total number of CPUs, as shown in Figure 8.7e.

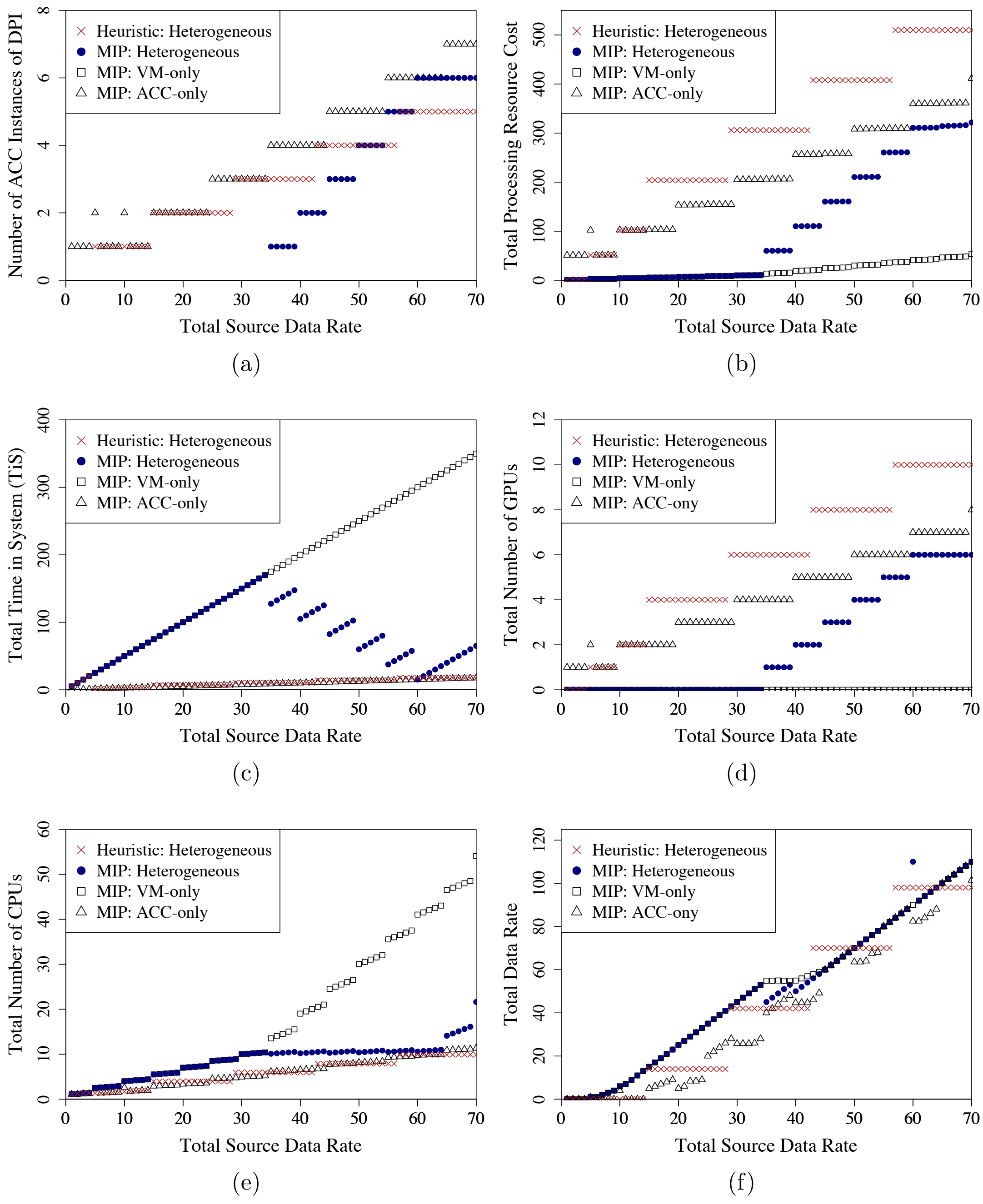


Figure 8.7: Results of the first set of experiments with template T_1 including a source and a multi-version DPI

In Figure 8.7c, above the source data rate of 35 units, the total TiS has an overall decreasing trend but increases in small intervals (e.g., between input data rate values 35 and 39). This can be explained in combination with Figures 8.7a and 8.7d. They show the number of ACC instances and the number of used GPUs in the embedding, respectively. In small intervals, the number of allocated GPUs remains constant. By increasing the load, the TiS increases, up to a point that the embedding with this set of resources is not optimal anymore. In this case, additional ACC instances are added (e.g., at input data rate 40), with additional GPUs allocated to them, which create a sudden decrease in the TiS and the corresponding increase in the resource cost that can be observed in Figure 8.7b.

Figure 8.7f shows the changes in the total data rate flowing over the network nodes as the total source data rate increases. All approaches show a similarly increasing trend. As the heuristic approach uses more ACC instances in total, it is able to handle the source data rates using the created instances in larger intervals without the need for additional instances which could be located farther than the source. Non-increasing data rate over the network links in spite of the increasing source data rate means parts of the traffic does not flow through the links; instead, it remains inside one network node, e.g., when two instances connected with an edge in the overlay are mapped to the same node. Similarly, a decrease in the total data rate over network links (e.g., when data rate increases from 34 to 35 using the optimization approach) can occur if a flow that was previously mapped to a path over the network links is now realized as an internal connection in one of the nodes.

8.6.2 Scalability

In the second set of experiments, I show the heuristic results on larger substrate networks. I have used a multi-structure template T_2, shown in Figure 8.6b. In this template, the first option for deploying a DPI is a VM and the second option is an ACC version accompanied by an auxiliary VM. A placeholder ingress component with zero resource consumption and zero TiS for requests is used for separating the two options. I have used Network 1 with 20 nodes and 44 links (substrate graph eu_20_0_prob [109]), Network 2 with 50 nodes and 124 links (substrate graph eu_50_0_prob [109]), and Network 3 with 100 nodes and 230 links (substrate graph eu_100_0_prob [109]). I have used uniform capacities for the network nodes and links, with the capacity and cost of CPU and GPU resources set as described for the previous experiment. I have set the node and link capacities of Network 2–4 to 2.5, 5, and 10 times more than those of Network 1, respectively. The templates have 4 source locations from 4 distant nodes in each substrate network, each with data rates increasing from 1 to 25. The number of required VM and ACC deployment versions of the DPI is shown in Figure 8.8.

Because of the larger distances in larger networks and delay constraints of template arcs, in Network 3 and 4, each source location requires dedicated instances of the template components to be created close to it. Therefore, in these networks, the number of required instances is higher than in the smaller networks, even for very low data rate. The instances can be shared among the sources in Network 1 and 2, resulting in a lower number of instances in total. The results for Network 3 and 4 are overlapping.

In all substrate networks, with lower source data rates, the first deployment option of the DPI (VM) is selected. As load increases, more and more of the second option of the DPI (ACC with an auxiliary VM) are created.

For the largest instances in these experiments, the heuristic finds a solution in

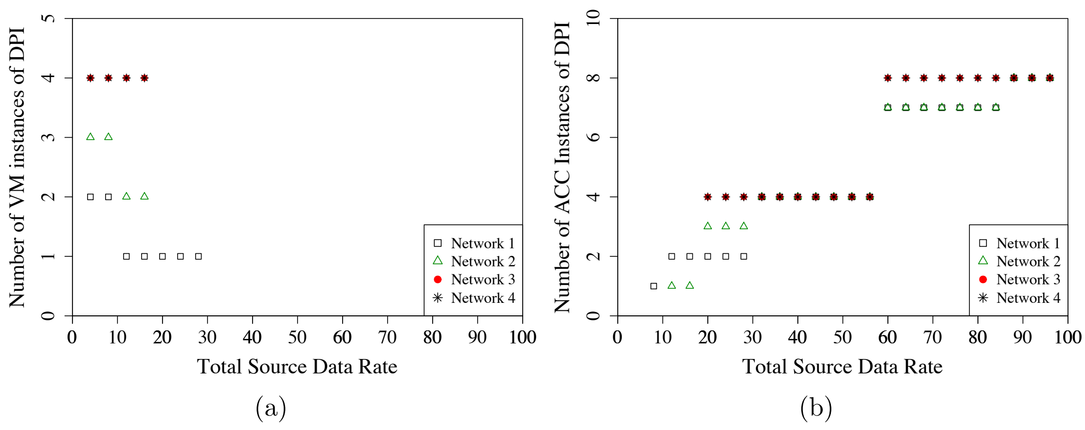


Figure 8.8: Results of the second set of experiments with the multi-structure template T_2 including four sources

less than 5 seconds. The optimization approach requires several minutes for the smallest instance and, as expected, cannot find solutions to large instances in a reasonable time.

Similar to the B-SPRING heuristic, I have implemented the path calculation in the M-SPRING heuristic based on the Floyd-Warshall algorithm, which is cubic in the number of nodes in the substrate network. For a network with 200 nodes (substrate graph eu_200_0_prob [109]), the heuristic requires 4 seconds for calculating the shortest paths. The run time of the heuristic also increases with the increasing number of sources, as the embedding process is repeated for every source. For example, for template T_1 (Figure 8.6a), the heuristic requires 4 milliseconds after the calculation of shortest paths to embed the template with one source with data rate 1 on this network with 200 nodes and 472 links. With 100 sources, each with data rate 1, the heuristic runs for 45 seconds, and with the highest number of possible source locations (200 on this network), each with data rate 1, the run time is 140 seconds. For larger networks among the VNMP instances, the path calculation was not completed even after several minutes. For such scenarios, a more time-efficient path calculation method is required.

8.7 Conclusion

In this chapter, I have shown the feasibility of defining heterogeneous pliable VCSs, including components with different deployment options. I have developed optimization and heuristic approaches for joint scaling, placement, routing, and version selection decisions for these VCSs.

Based on the evaluation results, in low-load situations, it is more cost-efficient to use general-purpose deployment versions, while for high-load situations, the hardware-accelerated versions of service components can be used for achieving a low TiS. Using these approaches, different resource types in heterogeneous infrastructures can be used efficiently to get suitable resource costs and TiS.

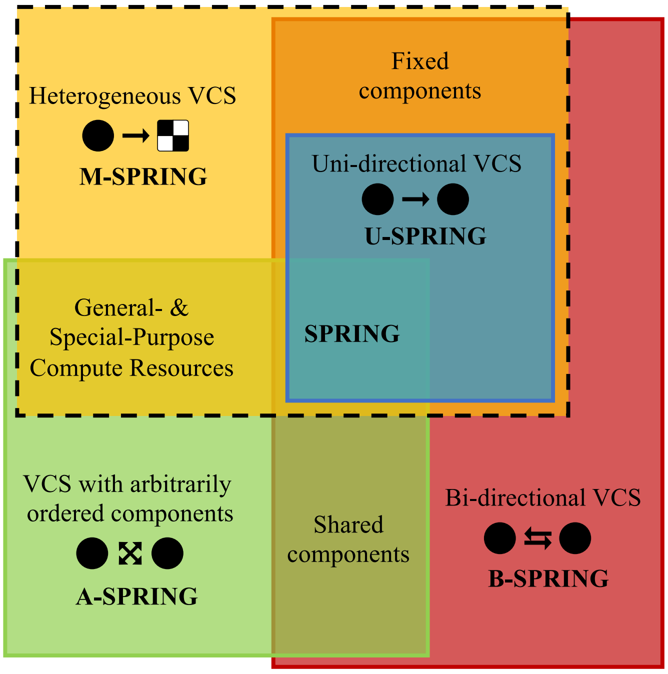


Figure 8.9: Relation of M-SPRING to A-SPRING, U-SPRING, and B-SPRING

As shown in Figure 8.9, the major difference of the templates used in the M-SPRING approaches to those of U-SPRING (Chapter 6) and B-SPRING (Chapter 7) approaches lies in the definition of multi-version service components and multi-structure VCSs, which results in different cost and performance options for VCSs. The *minimum* resource demands and the *maximum* expected TiS for service requests at each component (for each possible deployment version of it) are defined as functions of the input data rate in the service template. The actual TiS and resource demands of the instances are then determined while embedding the template into the network.

U-SPRING and B-SPRING approaches are based on the simplifying assumption that the resource demands of service components are defined as linear functions of their input data rates. With this assumption, the problems can be formulated as a MILP and solved using conventional solvers for optimization problems. To model more complex dependencies on the input data rate, in the M-SPRING model, I assume these functions are given as piecewise linear functions. The M-SPRING model is an extension of the U-SPRING model and includes all its core features.

In the M-SPRING model, I assume the heterogeneous pliable VCSs are uni-directional VCSs, as in the U-SPRING problem. The approaches can be extended to support bi-directional templates as well. Similar to the B-SPRING approaches, the M-SPRING approaches support service components fixed to a certain location, e.g., to model the endpoints of a VCS or legacy Physical Network Functions (PNFs).

The A-SPRING model (Section 3.2.1) also provides a limited support for special-purposes compute resources in addition to the general-purpose ones. However, that model does not include the resulting differences by using different versions for service components, in terms of cost or performance and, therefore, is not suitable for correctly capturing the requirements of scaling, placing, and routing of pliable VCSs with multi-version service components.

9

Results and Future Research Directions

In this chapter, first, I discuss the results of my contributions in the dissertation. I also analyze the practical applicability of my proposals based on the state of the art in service management and orchestration in the context of network softwarization. In Section 9.2, I identify future research directions to fill in the remaining gaps.

9.1 Results and Discussion

In the rest of this section, I describe how my proposed approaches can solve the six shortcomings of existing approaches that I have described in Section 1.1.

Flexible Composition and Orchestration of VCSs To address the shortcomings described in Section 1.1.1, I have extended the models from my previous research [38] to a more powerful model for describing pliable Virtualized Composed Services (VCSs) with arbitrarily ordered components. Instead of an inflexible total order among service components, this model allows to define a partial order among them. This enables composing VCSs using the order among service components that best fits the requirements of the service.

I have described a heuristic for selecting the appropriate service graphs for pliable VCS with arbitrarily ordered components. The selected service graphs are the input to the joint scaling, placement, and routing problem (A-SPRING) that I have formulated for pliable VCSs with arbitrarily ordered components. I have also presented a heuristic that provides quick and close-to-optimal solutions to the A-SPRING problem for mapping the selected combinations to the substrate network. The evaluation results for these approaches show the feasibility of defining a pliable structure for complex VCSs bypassing a combinatorial explosion.

Changing the order of service components is an additional degree of freedom, besides changing the placement of service components or starting new instances of service components to meet the service-level objectives. Changing the composition of a pliable VCS might result in changing the placement and routing for the service. It might also change the number of required instances for service components. This, in turn, might require synchronizing and migrating the state among instances (of stateful service components), flow handover, etc. These operations are also required for dynamic (re-)placement, scaling, and (re-)routing solutions for service components that work without changing the composition. Therefore, in a cloud and Network Function Virtualization (NFV) Management and Orchestration (MANO) system that is capable of dynamic service life-cycle management, changing the composition of service components does not impose any additional basic operations. The only additional requirement is the pre-processing of service deployment requests to decide the composition.

Resource Demands as a Function of Load To solve the issues described in Section 1.1.2, I have presented abstract service templates for defining the structure and resource demands of pliable VCSs. These templates include the required service components and their intended inter-connectivity. The resource demands of each service component are defined as a function of the data rate on its inputs. This allows a much more realistic modeling of the resource needs of service components than the constant resource needs assumed by the existing approaches in this context. In this way, the actual resource demands can be calculated and adapted according to the load, reducing the risk of under- or over-estimating the required resources.

For this approach to work accurately, a powerful and accurate profiling system needs to be in place, which can be used to formalize the relationships among the load, the target performance, and the required compute, storage, and networking resources for individual service components as well as the intended composition of them as a VCS, for example in the form of the functions used in the SPRING approaches.

I have shown the feasibility of defining resource demands as a function of the load using very simple examples by testing a limited set of possible configurations for the allocated resources to a VCS. Extracting such functions for realistic applications requires comprehensive profiling and benchmarking tools, methodologies, and large data sets. DevOps approaches that converge the development and operation steps of VCSs and their components are being introduced into the network softwarization context [25]. Using these approaches, automated testing [119] and profiling steps [120, 121, 122, 123] can be included into the design and development workflows of VCSs. The resulting models can be enhanced and complemented, e.g., with the help of public data sets [124] or by monitoring the VCSs in production environments, resulting in precise and realistic resource demand and performance models.

Load-Proportional Structure for Uni-Directional VCSs Using flexible service templates, I have defined pliable VCSs with load-proportional structures to address the shortcomings described in Section 1.1.3. Instead of limiting the number of required instances for service components, with this model, the structure of the VCS can be dynamically adapted to the load. One category of such VCSs that I have identified consists of uni-directional pliable VCSs with load-proportional structures. This is an enhanced model for a typical VCS, which is described as a fixed, directed acyclic service graph in existing studies on resource allocation and service mapping. A uni-directional VCS consists of a set of service components that should be traversed in a single forwarding direction, i.e., by the upstream *or* the downstream flows. This model reduces the risk of inaccurate resource demand estimations and, combined with an appropriate template embedding solution, allows adapting the structure of the VCS (i.e., the number of required instances for each service component), e.g., according to the location and request rate of its users.

Load-Proportional Structure for Bi-Directional VCSs As mentioned in Section 1.1.4, bi-directional VCSs have not been modeled and investigated sufficiently in the existing scaling, placement, or routing approaches. To fill in this gap, I have presented an extension to the uni-directional pliable VCS model that expresses bi-directional pliable VCSs. This is a comprehensive, flexible, and realistic model for VCSs that consist of the components required by upstream *and* downstream flows, where each flow returns to the source where it initiated. Using this model, stateful service components can be marked as such, which can be used by an appropriate life-cycle management solution to ensure the exact same instance of a stateful component is used for both upstream and downstream flows.

Load-Proportional Structure for Heterogeneous VCSs I have presented another category of pliable VCSs with load-proportional structures that consist of heterogeneous, multi-version components. This model exploits the opportunities offered by network softwarization and the solutions that integrate and unify large-scale multi-technology infrastructures. Using this model, different deployment versions can be specified for a single service component, each requiring a different number of different resource types (e.g., general-purpose or special-purpose compute resources). In this way, a service component can have a different cost and performance depending on the selected version. Different versions implementing a certain functionality may require different number of components. In this way, the structure of the VCS may also differ, according to the selected deployment version for the included service components. In this model, I assume uni-directional VCSs but the approaches can easily be extended to support bi-directional VCSs.

Jointly Deciding Scaling, Placement, Routing, and Deployment Versions I have presented approaches for the SPRING problem that can scale, place, and create paths for multiple pliable VCSs on a common substrate network. Unlike existing approaches to service embedding that only consider a subset of the

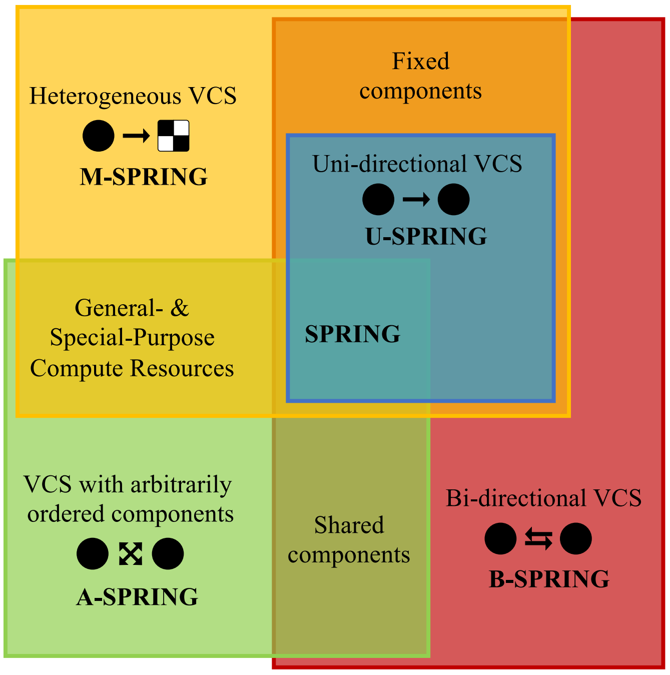


Figure 9.1: Overview of SPRING approaches

SPRING problem, these approaches consist of a joint, single-step decision process. Figure 9.1 shows an overview of these approaches and their interrelation.

Empiric tests have shown that the SPRING approaches find a balance among conflicting requirements of different VCSs, goals of different service providers, and overall objectives of a network operator that hosts the VCSs on its infrastructure. They ensure that the allocated capacity quickly follows changes in the demand.

Overall, the evaluations give evidence to the feasibility of the joint, single-step scaling, placement and routing approach for pliable VCSs defined using flexible service templates. Using these approaches, service developers and providers can specify VCSs at a high level of abstraction without needlessly limiting the structure and the risk of over-/under-estimating the required resources. At the same time, network operators can re-optimize the system configuration (i.e., the state of embedded services and the utilization of different resources in the network) after changes, ensuring the resilience of the running VCSs and meeting the service-level objectives, using a minimal set of modifications. Depending on the priorities of the service providers or network operators, the solutions can be adjusted to deliver the required results, e.g., by changing the optimization objective functions in the Mixed-Integer Linear Program (MILP) formulations or adapting the decision processes of the heuristics.

Integrating the scaling, placement, and routing steps into a single-step decision process requires cloud and NFV MANO frameworks that can support this. The leading open-source NFV MANO systems like OSM [53] and SONATA [50] have a modular design that allows customized workflows and innovative life-cycle management approaches. For example, the service platform of SONATA [27] has a customizable service life-cycle management plug-in. A network operator that uses this MANO system for its VCS can easily modify the order of life-cycle management operations (i.e., instantiation, placement, scaling, chaining, termination, etc.) if needed and customize how each of these operations are performed. Using service-specific management programs supported in SONATA, it is possible to specify when and how the scaling, placement, and routing operations are performed for each network service, making the practical application of the SPRING approaches possible.

The 5G Operating System (5G OS) [19], proposed within the 5G-PICTURE project [125], is an example of ongoing efforts [126] for integrating Software-Defined Networking (SDN), NFV, and network slicing [127] concepts and providing tools and well-defined interfaces among service components, SDN controllers, and NFV MANO systems for managing VCSs on top of heterogeneous infrastructures. These tools enable applying approaches like M-SPRING on top of a heterogeneous infrastructure.

9.2 Future Research Directions

Promising future research directions include further algorithmic enhancements to the presented solutions and development of new algorithms. For example, the heuristic approach to the B-SPRING problem embeds bi-directional services very quickly, compared to the optimization approach. Tens of seconds are, however, required for solving medium-sized problem instances because of the complexity of the problem and the used path calculation method within the heuristic. Depending on the frequency of changes that happen in the system configuration and the expected reaction time to these changes, heuristics might be required that are faster, even if the gap towards optimal embeddings might get larger in that case.

As an extension of the presented models, the option of specifying an arbitrary order among (a subset of) service components can be integrated into the abstract service templates used for pliable VCSs with load-proportional structures. In this way, service templates are created that can model all four categories of pliable VCSs introduced in this dissertation. Using such templates, the service graph generation and selection decision can also be combined with the scaling, placement, and routing decisions to be taken based on the templates, as an extension to the template embedding solutions.

Another possible research direction is to investigate the effects of different queuing models at the inputs of service components, other than the M/D/1 model that I have assumed in this dissertation, and to further analyze the proposed solutions using real-world data on the performance and resource demand behaviors in real-world services. The SPRING approaches can also be extended to consider the results of predictions and information about the future state of load while planning resource allocation. E.g., using predictions about patterns of changes in load for different VCSs, special treatment of temporary services that are designed for extremely high load (like a video streaming service from a large sports event), etc.

Additionally, the approaches can be optimized to produce reasonable embeddings within a pre-defined time limit. The time limit can be adapted to the frequency of changes in the substrate network and the running services, as well as the overhead of the actual deployment of the results produced by the SPRING approaches.

The substrate network model in the SPRING approaches can be modified to consider different domains of administration with different organizations (e.g., hierarchical or peer-to-peer), where, e.g., different parts of a VCS needs to be embedded separately based on local policies and custom SPRING approaches

While NFV, SDN, and Service Function Chaining (SFC) concepts are developing rapidly, there are still many compatibility issues in the existing solutions for each of these fields. For practical applicability of the presented models and approaches in large-scale, heterogeneous networks, there is a need for flexible and customizable cloud and NFV MANO systems that have a reliable integration with different SDN solutions designed for different networking technologies. Together, these solutions can provide a dynamic SFC solution, allowing easy and fast modifications to deployed VCSs. By resolving these issues, the true power of network softwarization can be unleashed. The additional flexibility offered by pliable VCSs can turn the service specification, management, and orchestration approaches presented in this dissertation into a viable and crucial part of next-generation networks.

Bibliography

[1] ETSI NFV ISG. *GS NFV 003 V1.1.1 Network Function Virtualisation (NFV); Terminology for Main Concepts in NFV.* Group Specification. Oct. 2013.

[2] 5G-PICTURE Project. *Deliverable 2.1: 5G and Vertical services, use cases and requirements.* URL: `https://www.5g-picture-project.eu/download/5g-picture_d21.pdf` (visited on 01/03/2019).

[3] T. Shimizu, A. Nakao, and K. Satoh. "Network Softwarization View of 5G Networks". In: *5G Networks: Fundamental Requirements, Enabling Technologies, and Operations Management.* John Wiley & Sons, Ltd, 2018. Chap. 13, pp. 499–518. ISBN: 9781119333142. DOI: `10.1002/9781119333142.ch13`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119333142.ch13`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119333142.ch13`.

[4] A. Galis, S. Clayman, L. Mamatas, J. Rubio Loyola, A. Manzalini, S. Kuklinski, J. Serrat, and T. Zahariadis. "Softwarization of Future Networks and Services. Programmable Enabled Networks as Next Generation Software Defined Networks". In: *2013 IEEE SDN for Future Networks and Services (SDN4FNS).* Nov. 2013. DOI: `10.1109/SDN4FNS.2013.6702557`.

[5] F. De Turck, J.-M. Kang, H. Choo, M.-S. Kim, B.-Y. Choi, R. Badonnel, and J. W.-K. Hong. "Softwarization of Networks, Clouds, and Internet of Things". In: *International Journal of Network Management* 27.2 (2017), e1967. DOI: `10.1002/nem.1967`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/nem.1967`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.1967`.

[6] ETSI NFV ISG. *GS NFV-MAN 001 V1.1.1 Network Function Virtualisation (NFV); Management and Orchestration.* Group Specification. Dec. 2014.

[7] B. Yi, X. Wang, K. Li, S. k. Das, and M. Huang. "A Comprehensive Survey of Network Function Virtualization". In: *Computer Networks* 133 (2018), pp. 212–262. ISSN: 1389-1286. DOI: `https://doi.org/10.1016/j.comnet.2018.01.021`. URL: `http://www.sciencedirect.com/science/article/pii/S1389128618300306`.

[8] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. "Software-Defined Networking: A Comprehensive Survey". In: *Proceedings of the IEEE* 103.1 (Jan. 2015), pp. 14–76.

[9] A. Gupta, M. F. Habib, U. Mandal, P. Chowdhury, M. Tornatore, and B. Mukherjee. "On Service-Chaining Strategies Using Virtual Network Functions in Operator Networks". In: *Computer Networks* 133 (2018), pp. 1–16. ISSN: 1389-1286. DOI: `https://doi.org/10.1016/j.comnet.2018.01.028`. URL: `http://www.sciencedirect.com/science/article/pii/S1389128618300379`.

[10] T. Taleb. "Guest Editorial First Edition of Series On Network Softwarization and Enablers". In: *IEEE Journal on Selected Areas in Communications* 36.3 (Mar. 2018), pp. 381–383. ISSN: 0733-8716. DOI: `10.1109/JSAC.2018.2827538`.

[11] Q. Zhang, L. Cheng, and R. Boutaba. "Cloud Computing: State-of-the-Art and Research Challenges". In: *Journal of Internet Services and Applications* 1.1 (May 2010), pp. 7–18. ISSN: 1869-0238. DOI: `10.1007/s13174-010-0007-6`. URL: `https://doi.org/10.1007/s13174-010-0007-6`.

[12] A. N. Toosi, R. N. Calheiros, and R. Buyya. "Interconnected Cloud Computing Environments: Challenges, Taxonomy, and Survey". In: *ACM Computing Surveys* 47.1 (July 2014), 7:1–7:47. ISSN: 0360-0300. DOI: `10.1145/2593512`. URL: `http://doi.acm.org/10.1145/2593512`.

[13] R. K. Naha, S. Garg, D. Georgakopoulos, P. P. Jayaraman, L. Gao, Y. Xiang, and R. Ranjan. "Fog Computing: Survey of Trends, Architectures, Requirements, and Research Directions". In: *IEEE Access* 6 (Aug. 2018), pp. 47980–48009. ISSN: 2169-3536. DOI: `10.1109/ACCESS.2018.2866491`.

[14] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie. "Mobile Edge Computing: A Survey". In: *IEEE Internet of Things Journal* 5.1 (Feb. 2018), pp. 450–465. ISSN: 2327-4662. DOI: `10.1109/JIOT.2017.2750180`.

[15] *OpenStack Flavors*. URL: `https://docs.openstack.org/nova/latest/user/flavors.html` (visited on 05/23/2019).

[16] T. Nadeau and Q. Quinn. *Problem Statement for Service Function Chaining*. Internet Request for Comments RFC 7498. IETF, 2015, pp. 1–13.

[17] *The American Heritage Dictionary of the English Language, Fifth Edition*. Houghton Mifflin Harcourt Publishing Company, 2019. URL: `https://ahdictionary.com/word/search.html?q=pliable` (visited on 01/02/2019).

[18] M. Keller, C. Robbert, and H. Karl. "Template Embedding: Using Application Architecture to Allocate Resources in Distributed Clouds". In: *IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*. 2014. DOI: `10.1109/UCC.2014.49`.

[19] S. Dräxler, H. Karl, H. R. Kouchaksaraei, A. Machwe, C. Dent-Young, K. Katsalis, and K. Samdanis. "5G OS: Control and Orchestration of Services on Multi-Domain Heterogeneous 5G Infrastructures". In: *2018 European Conference on Networks and Communications (EuCNC)*. June 2018. DOI: `10.1109/EuCNC.2018.8443210`.

[20] G. Bianchi, E. Biton, N. Blefari-Melazzi, I. Borges, L. Chiaraviglio, P. de la Cruz Ramos, P. Eardley, F. Fontes, M. J. McGrath, L. Natarianni, D. Niculescu, C. Parada, M. Popovici, V. Riccobene, S. Salsano, B. Sayadi, J. Thomson, C. Tselios, and G. Tsolis. "Superfluidity: A Flexible Functional Architecture for 5G Networks". In: *Transactions on Emerging Telecommunications Technologies* 27.9 (2016), pp. 1178–1186. DOI: `10.1002/ett.3082`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3082`.

[21] *Gurobi Optimizer*. URL: `http://www.gurobi.com/products/gurobi-optimizer` (visited on 05/23/2019).

[22] S. Mehraghdam, M. Keller, and H. Karl. "Specifying and Placing Chains of Virtual Network Functions". In: *3rd International Conference on Cloud Networking (CloudNet)*. IEEE. Oct. 2014, pp. 7–13. DOI: `10.1109/CloudNet.2014.6968961`.

[23] S. Mehraghdam and H. Karl. "Specification of Complex Structures in Distributed Service Function Chaining Using a YANG Data Model". In: *CoRR* abs/1503.02442 (2015). arXiv: `1503.02442`. URL: `http://arxiv.org/abs/1503.02442`.

[24] S. Mehraghdam and H. Karl. "Placement of Services with Flexible Structures Specified by a YANG Data Model". In: *2nd IEEE International Conference on Network Softwarization (NetSoft)*. IEEE. June 2016. DOI: `10.1109/NETSOFT.2016.7502412`.

[25] H. Karl, S. Dräxler, M. Peuster, A. Galis, M. Bredel, A. Ramos, J. Martrat, M. S. Siddiqui, S. V. Rossem, W. Tavernier, et al. "DevOps for Network Function Virtualisation: An Architectural Approach". In: *Transactions on Emerging Telecommunications Technologies* 27.9 (2016), pp. 1206–1215. DOI: `10.1002/ett.3084`.

[26] S. Dräxler, H. Karl, and Z. Á. Mann. "Joint Optimization of Scaling and Placement of Virtual Network Services". In: *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. CCGrid '17. Madrid, Spain: IEEE Press, May 2017, pp. 365–370. ISBN: 978-1-5090-6610-0. DOI: `10.1109/CCGRID.2017.25`. URL: `https://doi.org/10.1109/CCGRID.2017.25`.

[27] S. Dräxler, H. Karl, M. Peuster, H. R. Kouchaksaraei, M. Bredel, J. Lessmann, T. Soenen, W. Tavernier, S. Mendel-Brin, and G. Xilouris. "SONATA: Service Programming and Orchestration for Virtualized Software Networks". In: *IEEE International Conference on Communications Workshops (ICC Workshops)*. IEEE. May 2017, pp. 973–978. DOI: `10.1109/ICCW.2017.7962785`.

[28] M. Peuster, S. Dräxler, H. R. Kouchaksaraei, S. V. Rossem, W. Tavernier, and H. Karl. "A Flexible Multi-PoP Infrastructure Emulator for Carrier-Grade MANO Systems". In: *3rd IEEE International Conference*

on Network Softwarization (NetSoft) Demo Track. IEEE. July 2017. DOI: 10.1109/NETSOFT.2017.8004250.

[29] S. Dräxler and H. Karl. "Specification, Composition, and Placement of Network Services with Flexible Structures". In: *International Journal of Network Management* 27.2 (2017). DOI: 10.1002/nem.1963. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/nem.1963.

[30] S. Dräxler, M. Peuster, M. Illian, and H. Karl. "Towards Predicting Resource Demands and Performance of Distributed Cloud Services". In: *KuVS-Fachgespräch Fog Computing 2018.* Technical Report. Mar. 2018. URL: http://www.infosys.tuwien.ac.at/docs/proceedings.pdf#page=14.

[31] S. Dräxler, M. Peuster, M. Illian, and H. Karl. "Generating Resource and Performance Models for Service Function Chains: The Video Streaming Case". In: *4th IEEE International Conference on Network Softwarization (NetSoft).* IEEE. June 2018. DOI: 10.1109/NETSOFT.2018.8460029.

[32] S. Dräxler, S. Schneider, and H. Karl. "Scaling and Placing Bidirectional Services with Stateful Virtual and Physical Network Functions". In: *4th IEEE International Conference on Network Softwarization (NetSoft).* IEEE. June 2018. DOI: 10.1109/NETSOFT.2018.8459915.

[33] H. R. Kouchaksaraei, S. Dräxler, M. Peuster, and H. Karl. "Programmable and Flexible Management and Orchestration of Virtualized Network Functions". In: *2018 European Conference on Networks and Communications (EuCNC).* June 2018. DOI: 10.1109/EuCNC.2018.8442528.

[34] S. Dräxler, H. Karl, and Z. Ádám. "JASPER: Joint Optimization of Scaling, Placement, and Routing of Virtual Network Services". In: *IEEE Transactions on Network and Service Management* 15.3 (Sept. 2018), pp. 946–960. ISSN: 1932-4537. DOI: 10.1109/TNSM.2018.2846572.

[35] S. Schneider, S. Dräxler, and H. Karl. "Trade-offs in Dynamic Resource Allocation in Network Function Virtualization". In: *1st Workshop on Advanced Control Planes for Software Networks (ACPSN) at IEEE Global Communications Conference (GLOBECOM).* IEEE. Dec. 2018. DOI: 10.1109/GLOCOMW.2018.8644352.

[36] S. Dräxler and H. Karl. "SPRING: Scaling, Placement, and Routing of Heterogeneous Services with Flexible Structures". In: *5th IEEE International Conference on Network Softwarization (NetSoft).* IEEE. June 2019.

[37] *Source Code of the SPRING Approaches.* URL: https://github.com/CN-UPB/SPRING (visited on 05/14/2019).

[38] S. Mehraghdam. "Adaptive Placement of Programmable Virtual Network Function Chains". Master's Thesis. Paderborn University, 2014.

[39] M. Illian. "Prediction of Resource Requirements and Performance of Virtualised Network Functions in a Video Streaming Context". Bachelor's Thesis. Paderborn University, 2017.

[40] S. Schneider. "Specifying, Scaling, Placing, and Reusing Bidirectional Forwarding Graphs of Virtual Network Functions". Master's Thesis. Paderborn University, 2017.

[41] L. Sun, H. Dong, and J. Ashraf. "Survey of Service Description Languages and Their Issues in Cloud Computing". In: *IEEE 8th International Conference on Semantics, Knowledge and Grids (SKG)*. 2012. DOI: `10.1109/SKG.2012.49`.

[42] M. T. Beck and J. F. Botero. "Scalable and Coordinated Allocation of Service Function Chains". In: *Computer Communications* 102 (2017), pp. 78–88. ISSN: 0140-3664. DOI: `https://doi.org/10.1016/j.comcom.2016.09.010`.

[43] S. Schneider, A. Sharam, H. Karl, and H. Wehrheim. "Specifying and Analyzing Virtual Network Services Using Queuing Petri Nets". In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2019.

[44] W. Ma, O. Sandoval, J. Beltran, D. Pan, and N. Pissinou. "Traffic Aware Placement of Interdependent NFV Middleboxes". In: *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*. IEEE. 2017. DOI: `10.1109/INFOCOM.2017.8056993`.

[45] W. Ma, J. Beltran, Z. Pan, D. Pan, and N. Pissinou. "SDN-Based Traffic Aware Placement of NFV Middleboxes". In: *IEEE Transactions on Network and Service Management* 14.3 (2017), pp. 528–542. DOI: `10.1109/TNSM.2017.2729506`.

[46] M. Gao, B. Addis, M. Bouet, and S. Secci. "Optimal Orchestration of Virtual Network Functions". In: *Computer Networks* 142 (2018), pp. 108–127. ISSN: 1389-1286. DOI: `https://doi.org/10.1016/j.comnet.2018.06.006`. URL: `http://www.sciencedirect.com/science/article/pii/S1389128618303578`.

[47] B. Addis, D. Belabed, M. Bouet, and S. Secci. "Virtual Network Functions Placement and Routing Optimization". In: *4th International Conference on Cloud Networking (CloudNet)*. IEEE. 2015. DOI: `10.1109/CloudNet.2015.7335301`.

[48] H. Moens and F. De Turck. "VNF-P: A Model for Efficient Placement of Virtualized Network Functions". In: *IEEE 10th Conference on Network and Service Management (CNSM)*. 2014. DOI: `10.1109/CNSM.2014.7014205`.

[49] H. R. Kouchaksaraei, T. Dierich, and H. Karl. "Pishahang: Joint Orchestration of Network Function Chains and Distributed Cloud Applications". In: *4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*. IEEE, June 2018. DOI: `10.1109/NETSOFT.2018.8460134`.

[50] *SONATA Project*. URL: `http://sonata-nfv.eu` (visited on 12/22/2018).

[51] *UNIFY Project*. URL: `www.fp7-unify.eu` (visited on 12/22/2018).

[52] *T-NOVA Project*. URL: `http://www.t-nova.eu` (visited on 12/22/2018).

[53] *OSM Project.* URL: https://osm.etsi.org (visited on 12/22/2018).

[54] H. Moens and B. Volckaert. “Comparing Topology and Stream Based Strategies for Modeling Service Function Chains”. In: *IEEE 2nd Conference on Network Softwarization (NetSoft).* 2016. DOI: 10.1109/NETSOFT.2016.7502418.

[55] M. Melo, S. Nickel, and F. Saldanha-da-Gama. “Facility Location and Supply Chain Management—A Review”. In: *European Journal of Operational Research* 196 (2 July 2009), pp. 401–412. DOI: 10.1016/j.ejor.2008.05.007.

[56] G. Nagy and S. Salhi. “Location-Routing: Issues, Models and Methods”. In: *European Journal of Operational Research* 177.2 (Mar. 2007), pp. 649–672. ISSN: 03772217. DOI: 10.1016/j.ejor.2006.04.004. URL: http://linkinghub.elsevier.com/retrieve/pii/S0377221706002670.

[57] C. Prodhon and C. Prins. “A Survey of Recent Research on Location-Routing Problems”. In: *European Journal of Operational Research* (Jan. 2014).

[58] B. Awerbuch and T. Leighton. “Improved Approximation Algorithms for the Multi-Commodity Flow Problem and Local Competitive Routing in Dynamic Networks”. In: *26th ACM Symposium on Theory Of Computing (STOC).* Vol. 94. 1994, pp. 487–496.

[59] I. Houidi, W. Louati, and D. Zeghlache. “Exact Multi-Objective Virtual Network Embedding in Cloud Environments”. In: *The Computer Journal* 58.3 (2015), pp. 403–415. DOI: 10.1093/comjnl/bxu154.

[60] J. Li, N. Zhang, Q. Ye, W. Shi, W. Zhuang, and X. Shen. “Joint Resource Allocation and Online Virtual Network Embedding for 5G Networks”. In: *IEEE Global Communications Conference (GLOBECOM).* Dec. 2017. DOI: 10.1109/GLOCOM.2017.8254072.

[61] A. Baumgartner, V. S. Reddy, and T. Bauschert. “Mobile Core Network Virtualization: A Model for Combined Virtual Core Network Function Placement and Topology Optimization”. In: *1st IEEE Conference on Network Softwarization (NetSoft).* IEEE. 2015. DOI: 10.1109/NETSOFT.2015.7116162.

[62] P. T. Endo, A. V. de Almeida Palhares, N. N. Pereira, G. E. Goncalves, D. Sadok, J. Kelner, B. Melander, and J. Mangs. “Resource Allocation for Distributed Cloud: Concepts and Research Challenges”. In: *IEEE Network* 25.4 (July 2011), pp. 42–46. ISSN: 0890-8044. DOI: 10.1109/MNET.2011.5958007.

[63] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano. “A Review of Auto-Scaling Techniques for Elastic Applications in Cloud Environments”. In: *Journal of Grid Computing* 12.4 (2014), pp. 559–592. DOI: https://doi.org/10.1007/s10723-014-9314-7.

[64] Z. Á. Mann. "Interplay of Virtual Machine Selection and Virtual Machine Placement". In: *Proceedings of the 5th European Conference on Service-Oriented and Cloud Computing.* 2016, pp. 137–151. DOI: https://doi.org/10.1007/978-3-319-44482-6_9.

[65] Z. Á. Mann. "Allocation of Virtual Machines in Cloud Data Centers—A Survey of Problem Models and Optimization Algorithms". In: *ACM Computing Surveys* 48.1 (2015), 11:1–11:34. DOI: 10.1145/2797211.

[66] D. M. Divakaran and M. Gurusamy. "Towards Flexible Guarantees in Clouds: Adaptive Bandwidth Allocation and Pricing". In: *IEEE Transactions on Parallel and Distributed Systems* 26.6 (2015), pp. 1754–1764. DOI: 10.1109/TPDS.2014.2325044.

[67] E. Ahvar, S. Ahvar, N. Crespi, J. Garcia-Alfaro, and Z. Á. Mann. "NACER: a Network-Aware Cost-Efficient Resource Allocation Method for Processing-Intensive Tasks in Distributed Clouds". In: *14th IEEE International Symposium on Network Computing and Applications.* 2015, pp. 90–97. DOI: 10.1109/NCA.2015.37.

[68] M. Alicherry and T. Lakshman. "Optimizing Data Access Latencies in Cloud Systems by Intelligent Virtual Machine Placement". In: *IEEE INFOCOM.* 2013, pp. 647–655. DOI: 10.1109/INFCOM.2013.6566850.

[69] E. Ahvar, S. Ahvar, Z. Á. Mann, N. Crespi, J. Garcia-Alfaro, and R. Glitho. "CACEV: a Cost and Carbon Emission-Efficient Virtual Machine Placement Method for Green Distributed Clouds". In: *IEEE 13th International Conference on Services Computing.* 2016, pp. 275–282. DOI: 10.1109/SCC.2016.43.

[70] P. Bellavista, F. Callegati, W. Cerroni, C. Contoli, A. Corradi, L. Foschini, A. Pernafini, and G. Santandrea. "Virtual Network Function Embedding in Real Cloud Environments". In: *Computer Networks* 93 (Dec. 2015), pp. 506–517. ISSN: 13891286. DOI: 10.1016/j.comnet.2015.09.034.

[71] X. Wang, C. Wu, F. Le, A. Liu, Z. Li, and F. Lau. "Online VNF Scaling in Datacenters". In: *IEEE International Conference on Cloud Computing, CLOUD.* IEEE, June 2017, pp. 140–147. ISBN: 9781509026197. DOI: 10.1109/CLOUD.2016.26. eprint: 1604.01136.

[72] P. Cappanera, F. Paganelli, and F. Paradiso. "VNF Placement for Service Chaining in a Distributed Cloud Environment with Multiple Stakeholders". In: *Computer Communications* 133 (2019), pp. 24–40. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2018.10.008. URL: http://www.sciencedirect.com/science/article/pii/S0140366418303104.

[73] J. G. Herrera and J. F. Botero. "Resource Allocation in NFV: A Comprehensive Survey". In: *IEEE Transactions on Network and Service Management* 13.3 (2016), pp. 518–532. DOI: 10.1109/TNSM.2016.2598420.

[74] T. W. Kuo, B. H. Liou, K. C. J. Lin, and M. J. Tsai. “Deploying Chains of Virtual Network Functions: On the Relation between Link and Server Usage”. In: *IEEE/ACM Transactions on Networking (TON)* 26.4 (2018), pp. 1562–1576. DOI: 10.1109/TNET.2018.2842798.

[75] M. C. Luizelli, L. R. Bays, L. S. Buriol, M. P. Barcellos, and L. P. Gaspary. “Piecing Together the NFV Provisioning Puzzle: Efficient Placement and Chaining of Virtual Network Functions”. In: *IFIP/IEEE International Symposium on Integrated Network Management (IM)*. IEEE, 2015. DOI: 10.1109/INM.2015.7140281.

[76] S. Ahvar, H. P. Phyu, and R. Glitho. “CCVP: Cost-efficient Centrality-based VNF Placement and Chaining Algorithm for Network Service Provisioning”. In: *IEEE NetSoft*. IEEE, July 2017, pp. 1–9. ISBN: 9781509060085. DOI: 10.1109/NETSOFT.2017.8004104.

[77] F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and O. C. M. B. Duarte. “Orchestrating Virtualized Network Functions”. In: *IEEE Transactions on Network and Service Management* 13.4 (2016), pp. 725–739. DOI: 10.1109/TNSM.2016.2569020.

[78] M. Savi, M. Tornatore, and G. Verticale. “Impact of Processing Costs on Service Chain Placement in Network Functions Virtualization”. In: *IEEE 1st Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*. 2015. DOI: 10.1109/NFV-SDN.2015.7387426.

[79] S. Khebbache, M. Hadji, and D. Zeghlache. “Virtualized Network Functions Chaining and Routing Algorithms”. In: *Computer Networks* 114 (Feb. 2017), pp. 95–110. ISSN: 13891286. DOI: 10.1016/j.comnet.2017.01.008.

[80] M. C. Luizelli, W. L. da Costa Cordeiro, L. S. Buriol, and L. P. Gaspary. “A Fix-and-Optimize Approach for Efficient and Large Scale Virtual Network Function Placement and Chaining”. In: *Computer Communications* 102 (Apr. 2017), pp. 67–77. ISSN: 01403664. DOI: 10.1016/j.comcom.2016.11.002.

[81] T.-M. Nguyen, M. Minoux, and S. Fdida. “Optimizing Resource Utilization in NFV Dynamic Systems: New Exact and Heuristic Approaches”. In: *Computer Networks* (2018). ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2018.11.009. URL: http://www.sciencedirect.com/science/article/pii/S1389128618302287.

[82] S. Sahhaf, W. Tavernier, D. Colle, and M. Pickavet. “Network Service Chaining with Efficient Network Function Mapping Based on Service Decompositions”. In: *IEEE 1st Conference on Network Softwarization (NetSoft)*. Apr. 2015. DOI: 10.1109/NETSOFT.2015.7116126.

[83] A. Mehta and E. Elmroth. “Distributed Cost-Optimized Placement for Latency-Critical Applications in Heterogeneous Environments”. In: *IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, Sept. 2018. DOI: 10.1109/ICAC.2018.00022.

[84] P. Chuprikov, S. Nikolenko, and K. Kogan. “On Demand Elastic Capacity Planning for Service Auto-Scaling”. In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 2016. DOI: 10.1109/INFOCOM.2016.7524616.

[85] C. Fuerst, S. Schmid, L. Suresh, and P. Costa. “Kraken: Online and Elastic Resource Reservations for Multi-Tenant Datacenters”. In: *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*. 2016. DOI: 10.1109/INFOCOM.2016.7524466.

[86] D. Bhamare, M. Samaka, A. Erbad, R. Jain, L. Gupta, and H. A. Chan. “Optimal Virtual Network Function Placement in Multi-Cloud Service Function Chaining Architecture”. In: *Computer Communications* 102 (2017), pp. 1–16. ISSN: 0140-3664. DOI: https://doi.org/10.1016/j.comcom.2017.02.011.

[87] M. Ghaznavi, A. Khan, N. Shahriar, K. Alsubhi, R. Ahmed, and R. Boutaba. “Elastic Virtual Network Function Placement”. In: *IEEE 4th International Conference on Cloud Networking (CloudNet)*. 2015. DOI: 10.1109/CloudNet.2015.7335318.

[88] R. Mijumbi, J. Serrat, J.-L. Gorricho, N. Bouten, F. De Turck, and S. Davy. “Design and Evaluation of Algorithms for Mapping and Scheduling of Virtual Network Functions”. In: *IEEE 1st Conference on Network Softwarization (NetSoft)*. 2015. DOI: 10.1109/NETSOFT.2015.7116120.

[89] M. Bjorklund. *YANG—A Data Modeling Language for the Network Configuration Protocol (NETCONF)*. RFC 6020 (Standards Track). Internet Engineering Task Force, Oct. 2010.

[90] *Broadband Forum Liaison on Flexible Service Chaining to IETF Service Function Chaining Working Group*. 2014. URL: https://datatracker.ietf.org/documents/LIAISON/liaison-2014-02-13-broadband-forum-sfc-broadband-forum-work-on-flexible-service-chaining-sd-326-attachment-1.pdf.

[91] W. Haeffner, J. Napper, M. Stiemerling, D. Lopez, and J. Uttaro. *Service Function Chaining Use Cases in Mobile Networks*. Internet-Draft draft-ietf-sfc-use-case-mobility-03. IETF Secretariat, Jan. 2018.

[92] S. Kumar, M. Tufail, S. Majee, C. Captari, and S. Homma. *Service Function Chaining Use Cases in Data Centers*. Internet-Draft draft-ietf-sfc-dc-use-cases-02. IETF Secretariat, Jan. 2017.

[93] W. Liu, H. Li, O. Huang, M. Boucadair, N. Leymann, Q. Fu, Q. Sun, C. Pham, C. Huang, J. Zhu, and P. He. *Service Function Chaining (SFC) General Use Cases*. Internet-Draft draft-liu-sfc-use-cases-08. IETF Secretariat, Sept. 2014.

[94] S. Orlowski, M. Pióro, A. Tomaszewski, and R. Wessäly. “SNDlib 1.0 – Survivable Network Design Library”. In: *ENOG INOC*. 2007.

[95] T.-W. Shinn and T. Takaoka. “Variations on the Bottleneck Paths Problem”. In: *Theoretical Computer Science* 575 (2015). Special Issue on Algorithms and Computation, pp. 10–16. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2014.10.049.

[96] *Data from Resource Demand Modeling Experiments.* URL: https://uni-paderborn.sciebo.de/index.php/s/G9q2hmUNg4n8LEg (visited on 05/14/2019).

[97] *ffserver.* URL: https://trac.ffmpeg.org/wiki/ffserver (visited on 05/23/2019).

[98] *FFmpeg.* URL: https://ffmpeg.org (visited on 05/23/2019).

[99] *Squid.* URL: http://www.squid-cache.org (visited on 05/23/2019).

[100] *GNU Wget.* URL: https://www.gnu.org/software/wget (visited on 05/23/2019).

[101] *OpenStack Ocata.* URL: https://releases.openstack.org/ocata/index.html (visited on 05/23/2019).

[102] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. “Cuanta: Quantifying Effects of Shared On-Chip Resource Interference for Consolidated Virtual Machines”. In: *2nd ACM Symposium on Cloud Computing.* ACM. 2011. DOI: 10.1145/2038916.2038938.

[103] *Big Buck Bunny.* URL: https://peach.blender.org/download/ (visited on 12/23/2018).

[104] *TPB AFK: The Pirate Bay Away from Keyboard.* URL: http://www.imdb.com/title/tt2608732/ (visited on 12/23/2018).

[105] *The Art of Playing.* URL: http://www.imdb.com/title/tt3996164/ (visited on 12/23/2018).

[106] *Best Plays of: The International 5 Movie DOTA 2 Compilation highlights.* URL: https://www.youtube.com/watch?v=ERTIWNfx93w (visited on 12/23/2018).

[107] *TV static noise HD 1080p.* URL: https://www.youtube.com/watch?v=DH0BQtwEAsMw (visited on 12/23/2018).

[108] R. E. Korf. “Linear-Space Best-First Search”. In: *Artificial Intelligence* 62.1 (1993), pp. 41–78. DOI: https://doi.org/10.1016/0004-3702(93)90045-D.

[109] *Virtual Network Mapping Problem (VNMP) Benchmark Set.* URL: https://www.ac.tuwien.ac.at/files/resources/instances/vnmp (visited on 01/02/2019).

[110] J. Inführ and G. R. Raidl. “Solving the Virtual Network Mapping Problem with Construction Heuristics, Local Search and Variable Neighborhood Descent”. In: *Proceedings of the 13th European Conference on Evolutionary Computation in Combinatorial Optimization.* 2013, pp. 250–261. DOI: https://doi.org/10.1007/978-3-642-37198-1_22.

[111] F. Glover. “Tabu search—Part I”. In: *ORSA Journal on computing* 1.3 (1989), pp. 190–206. DOI: `https://doi.org/10.1287/ijoc.1.3.190`.

[112] O. Tange. “GNU Parallel - The Command-Line Power Tool”. In: *The USENIX Magazine* 36.1 (2011), pp. 42–47.

[113] ETSI NFV ISG. *Network Functions Virtualisation (NFV): Use Cases.* Group Specification ETSI GS NFV 001 V1.1.1. 2013.

[114] Tellabs. *Mobile Video Optimization Concept and Benefits.* White Paper. 2011. URL: `http://s3.amazonaws.com/zanran_storage/www.tellabs.com/ContentPages/2438991029.pdf` (visited on 01/02/2019).

[115] *OpenStack.* URL: `https://www.openstack.org/` (visited on 05/23/2019).

[116] *Kubernetes.* URL: `https://kubernetes.io` (visited on 05/23/2019).

[117] I. M. Araújo, C. Natalino, Á. L. Santana, and D. L. Cardoso. “Accelerating VNF-based Deep Packet Inspection with the use of GPUs”. In: *20th International Conference on Transparent Optical Networks (ICTON).* July 2018. DOI: `10.1109/ICTON.2018.8473638`.

[118] *Amazon EC2 On-Demand Pricing.* URL: `https://aws.amazon.com/ec2/pricing/on-demand/` (visited on 01/02/2019).

[119] S. Van Rossem, M. Peuster, L. Conceicao, H. Razzaghi Kouchaksaraei, W. Tavernier, D. Colle, M. Pickavet, and P. Demeester. “A Network Service Development Kit Supporting the End-to-End Lifecycle of NFV-based Telecom Services”. In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN).* 2017. DOI: `10.1109/NFV-SDN.2017.8169859`.

[120] R. V. Rosa, C. E. Rothenberg, and R. Szabo. “VBaaS: VNF Benchmark-as-a-Service”. In: *Fourth European Workshop on Software Defined Networks.* IEEE, Sept. 2015. DOI: `10.1109/EWSDN.2015.65`.

[121] L. Cao, P. Sharma, S. Fahmy, and V. Saxena. “NFV-VITAL: A Framework for Characterizing the Performance of Virtual Network Functions”. In: *IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN).* Nov. 2015. DOI: `10.1109/NFV-SDN.2015.7387412`.

[122] M. Peuster and H. Karl. “Understand Your Chains: Towards Performance Profile-based Network Service Management”. In: *Proceedings of the Fifth European Workshop on Software Defined Networks.* IEEE. 2016.

[123] M. Peuster and H. Karl. “Profile Your Chains, Not Functions: Automated Network Service Profiling in DevOps Environments”. In: *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN).* Nov. 2017. DOI: `10.1109/EWSDN.2016.9`.

[124] M. Peuster, S. Schneider, and H. Karl. “The Softwarised Network Data Zoo”. In: *arXiv preprint arXiv:1905.04962* (May 2019). URL: `https://arxiv.org/abs/1905.04962`.

[125] *5G-PICTURE Project.* URL: `https://www.5g-picture-project.eu` (visited on 01/03/2019).

[126] 5G-PICTURE Project. *Deliverable 5.1: Relationships between Orchestrators, Controllers, slicing systems.* URL: `https://www.5g-picture-project.eu/publication_deliverables.html` (visited on 01/03/2019).

[127] I. Afolabi, T. Taleb, K. Samdanis, A. Ksentini, and H. Flinck. “Network Slicing and Softwarization: A Survey on Principles, Enabling Technologies, and Solutions”. In: *IEEE Communications Surveys & Tutorials* 20.3 (thirdquarter 2018), pp. 2429–2453. ISSN: 1553-877X. DOI: `10.1109/COMST.2018.2815638`.