Technische Universität Berlin
Fakultät für Elektrotechnik und Informatik
Lehrstuhl für Intelligente Netze
und Management Verteilter Systeme

Dynamic Aspects of Network Virtualization
Algorithmic and Economic Opportunities

vorgelegt von
Arne Ludwig (Dipl.-Inf.)
geb. in Berlin

Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 4. Februar 2016

Berlin 2016
D 83

Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum Arne Ludwig (Dipl.-Inf.)

Abstract

The virtualization trend decouples services from the constraints of the underlying physical infrastructure. This decoupling facilitates more flexible and efficient resource allocations: the service can be realized at *any* place in the substrate network which fulfills the service specification requirements.

This thesis studies such flexibilities in the context of *virtual networks (VNets)*. A VNet describes a set of virtual nodes which are connected by virtual links; both nodes and links provide certain QoS or resource guarantees. The network virtualization paradigm envisions an Internet where customers can request arbitrary VNets from one or multiple substrate providers (e.g., an ISP or even indirectly via a broker). Indeed, network virtualization not only introduces flexibilities but also economical opportunities or even VNet markets: an aspect which has hardly been studied so far.

We, in the first part of this thesis, investigate opportunities and challenges of such a possible market. We identify a fundamental tradeoff between specification details given by the customer and embedding efficiency on the provider side. In particular, we introduce a formal framework, called the *Price of Specificity (PoS)*, to reason about and quantify this tradeoff. Subsequently, we study buying and pricing strategies for VNets: taking into account the multi-resource nature of VNets as well as possible discounts for larger quantities. We observe that the problem can be seen as multi-dimensional parking permit problem and present online algorithms as well as a competitive analysis, showing that our algorithms are asymptotically optimal.

Network virtualization not only introduces challenges in terms of embedding and pricing. Especially long-lived VNets also need to be adapted and reconfigured over time (e.g., due to maintenance, new specifications, or load balancing). However, we observe that changing VNets adaptively in a consistent manner is non-trivial, as changes typically need to be communicated to and implemented at different and distributed components simultaneously. At the same time, ensuring consistency during updates is critical in virtualized environments where isolation needs to be provided between users and services sharing the same resources.

Hence, in the second part of the thesis we study how to dynamically update VNets in a consistent way. In particular, we present a formal model and devise efficient algorithms to update SDN (Software Defined Networking)-based VNets such that security-critical functionality (like firewalls) are always traversed and loops avoided. We formally prove the correctness and efficiency of these algorithms, but also prove computational hardness results.

Zusammenfassung

Virtualisierung ermöglicht es mehreren Services gemeinsam, auf derselben physikalischen Infrastruktur bereitgestellt zu werden und diese somit effizienter zu nutzen. Nachdem die Virtualisierung von Endsystemen, beispielsweise in der Cloud, schon erfolgreich umgesetzt wird, findet dieser Trend jetzt auch vermehrt im Netzwerk statt. Virtuelle Netzwerke (VNetze) definieren ein Netzwerk aus virtuellen Knoten und Kanten mit Ressourcengarantien, welche flexibel von den Nutzern spezifiziert und direkt von Providern oder indirekt via Broker gemietet werden können.

Gegenstand dieser Arbeit sind die ökonomischen und algorithmischen Herausforderungen eines solchen VNetz-Marktes. Im ersten Teil untersuchen wir den Einfluss von Anforderungen unterschiedlichen Detailgrads auf das Einbettungsproblem. Dieses beschreibt das Problem von ressourceneffizienten Abbildungen der virtuellen auf die physikalischen Netze. Als Metrik, um die Auswirkungen der Flexibilität auf die Effizienz zu bestimmen, führen wir hierzu den Price of Specificity (PoS) ein. Wir analysieren, basierend auf den flexiblen Anforderungen, verschiedene Miet- und Preismechanismen unter Berücksichtigung von möglichen Rabatten auf größere Verträge. Wir zeigen, dass das Ressourcenmietproblem bei unbekannter Nachfrage einer mehrdimensionalen Variante des Parking-Permit-Problems entspricht, für die wir einen asymptotisch optimalen Algorithmus präsentieren.

VNetze bieten allerdings nicht nur Herausforderungen bezüglich der Einbettung und dem Ressourcenhandel. Insbesondere VNetze von langer Dauer müssen beispielsweise aufgrund von Anforderungsänderungen seitens des Kundens oder aufgrund von Lastverteilung seitens des Providers zur Laufzeit geändert werden. Diese Änderungen sind nicht trivial, da sie viele Netzwerkkomponenten betreffen, die jeweils individuell angepasst werden müssen. Im zweiten Teil der Arbeit untersuchen wir daher Algorithmen, um diese Netzwerkänderungen effizient und konsistent durchzuführen. Wir betrachten dabei SDN (Software Defined Networking) basierte VNetze und gehen von einem logisch zentralisierten Controller aus, der eine globale Übersicht über das Netzwerk hat. Trotz dieser globalen Sicht ist es nicht trivial, diese Netzwerkänderungen durchzuführen, da es zu Verzögerungen bei der Umsetzung der Änderungen auf den einzelnen Netzwerkkomponenten kommen kann. Wir betrachten konsistente Netzwerkänderungen mit einem Fokus auf Zyklen-Freiheit und Wegpunkt-Garantie. Zu diesem Zweck präsentieren wir eine Komplexitätsanalyse der Probleme, sowie effiziente Algorithmen zu deren Lösung.

Papers

Parts of this thesis are based on the following papers. All my collaborators are among my co-authors.

Pre-Published Papers

International Conferences

A. Ludwig, S. Schmid, and A. Feldmann. **The Price of Specificity in the Age of Network Virtualization.** In *Proceedings of the 5th IEEE/ACM International Conference on Utility and Cloud Computing (UCC),* pages 187–190, 2012

Extended version: A. Ludwig, S. Schmid, and A. Feldmann. **Specificity vs. Flexibility: On the Embedding Cost of a Virtual Network.** In *Proceedings of the 25th IEEE International Teletraffic Congress (ITC),* pages 1–9, 2013

X. Hu, A. Ludwig, A. Richa, and S. Schmid. **Competitive Strategies for Online Cloud Resource Allocation with Discounts.** In *Proceedings of the 35th IEEE International Conference on Distributed Computing Systems (ICDCS),* pages 93–102, 2015

A. Ludwig, J. Marcinkowski, and S. Schmid. **Scheduling Loop-Free Network Updates: It's Good to Relax!** In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC),* pages 13–22, 2015

Workshops

A. Ludwig, M. Rost, D. Foucard, and S. Schmid. **Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies.** In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets),* pages 15:1–15:7, 2014

A. Ludwig and S. Schmid. **Distributed Cloud Market: Who Benefits from Specification Flexibilities?** In *Presented at ACM DCC, published in ACM Performance Evaluation Review (PER) 43.3,* 2015

Technical Reports

A. Ludwig, C. Fuerst, A. Henze, and S. Schmid. **Opposites Attract: Virtual Cluster Embedding for Profit.** *CoRR, arXiv abs/1511.02354* 2015

Un-Published Papers

A. Ludwig, S. Dudycz, M. Rost and S. Schmid. **Transiently Secure Network Updates.** Accepted to *ACM SIGMETRICS / IFIP Performance 2016*, (to appear)

S. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid. **Transiently Consistent SDN Updates: Being Greedy is Hard.** *Under submission*

Contents

1 Introduction

The Internet has become an integral part of people's life and it is hard to imagine a world without the omnipresent services offered by, e.g., Amazon, Apple, Google, Microsoft. But, despite the Internet's rapid growth, the architecture and protocols underlying the Internet have hardly changed over the last decades. While given the many new services on the Internet, this can be seen as a huge success, we now slowly start to realize the limitations of the Internet. Especially the ossification of the Internet hinders necessary innovations. The Internet was not designed for its current scale and its evolution has led to a situation where its multiprovider nature of several different *Internet Service Providers (ISPs)* makes it hard to introduce changes into the architecture, as all of the ISPs would have to agree. However, the variety of services offered in the Internet, and hence, their different requirements, e.g., in terms of performance, mobility support, or security make an architectural change necessary. Emails contain sensitive data and the cloud is used to store private data, introducing critical security requirements. Moreover, the ever-increasing usage of mobile devices like smartphones and tablets can benefit from better mobility support.

A way to enable innovation is virtualization. In datacenters, virtualization is already a reality, as computing services are being used by companies for example as a spillover resource in times of high demands (cloud bursting). Even some service providers rely for their businesses (e.g., streaming services like Spotify [5]) on cloud providers which offer storage as well as frontends to deliver applications to the users. The benefits of using cloud services of, e.g., Amazon AWS and Microsoft Azure, are the low investment costs and that it is not necessary to get all the expertise to maintain and administrate the infrastructure into the company. This also makes the cloud attractive for startups. Given all of these use cases, it is not surprising that nowadays cloud computing is a huge market. A recent Goldman Sachs study [8] assumes a roughly 30% CAGR (compound annual growth rate) from 2013 until 2018 for the cloud, whereas the same study predicts only a 5% growth rate for the overall enterprise IT.

Even though datacenters are often under the control of a single entity, there is a need for virtualization on the network as well. There are several different kinds of jobs running within a datacenter imposing different loads on the network. This impacts the variance of the runtime even if the compute resources are isolated [100], as there are no guarantees on network performance. Thus, planning is difficult and the variance can also increase the cost, which makes network performance guarantees useful. While end-system virtualization has already been very successful in the context of datacenters and cloud computing, the virtualization trend now spills over to the network. A recent example is Google's *B4* [63], a *Software Defined Networking (SDN)* implementation which allows to flexibly and securely manage wide-area network traffic. The *network virtualization* paradigm envisions a world where entire *virtual networks (VNets)*, with QoS and isolation guarantees *on nodes as well as links*, can be requested on demand. Network virtualization decouples the applications and services from the constraints of the underlying physical network, and where and when resources are allocated only depends on the VNet specification itself.

Thus, even the traditionally more conservative ISPs have become interested in the opportunities of network virtualization. The opportunities they envision, are ways for ISPs to innovate their network as well as to offer novel services: many ISPs do not only have a large network, but also many geographically distributed resources such as storage and computation (e.g., in their "micro-datacenters", the *Points-of-Presence*, and the *street cabinets*). If these resources can be used (and/or leased) for new services, the monetization of the infrastructure is improved as well as its efficiency. Note that in contrast to other players in the Internet, e.g., *Content Distribution Network (CDN)* providers, ISPs have the advantage of knowing their infrastructure and the customers demand. This knowledge can be exploited to use the resources where and when they are most useful.

However, the introduced flexibilities must be exploited with care, as a dynamic and adaptive network operation may also introduce inconsistencies. The different allocations and increased dynamics cause more updates in the network, which are sensitive tasks as small mistakes can lead to large outages. Even tech-savvy companies often struggle with correct network operations. GitHub, for example, reported on a planned addition of switches into their network in November 2012 [2]. During this operation, they not only discovered a bridge loop, which caused the disabling of some links but also a broadcast storm caused by an undetected bridge loop. Overall, this incident led to 18 minutes of downtime of the entire service. A similar incident occurred at Amazon EC2 (Amazon Elastic Compute Cloud) in April 2011 [1]. A planned capacity upgrade in the US-East region went wrong, where instead of rerouting the traffic within the main network, it was routed via the low-capacity network of the storage system (EBS - Elastic Block Store). The traffic overload on this network made it unusable for the usual read/write accesses isolating many EBS nodes. This event triggered cascading events, as, e.g., the EBS nodes tried to re-mirror their data on other than the isolated nodes, which led to a several days outage.

To get a better handle on these issues, the SDN paradigm introduces a principled and formally verifiable way to specify and change policies, and hence offers ways to control traffic more fine-grained. Thus, SDN can be seen as an enabler for VNets [24, 37].

1.1 Problem Statement

The goal of the thesis is to provide the tools and insights needed to enable a *VNet market*. On this market, customers can specify resources on the end systems as well as on the network. Such a market may not only be attractive in the context of datacenter scenarios, where customers can then rely on their performance guarantees and hence, enable optimized planning, but also includes WAN (wide are network) scenarios where customers are potential service providers who want to have guaranteed performance to their services for a certain range of users.

To create such a VNet market we need to address two problems:

1. The economical aspects need to be understood.
2. The increased load on the network control caused by the VNets need to be handled efficiently.

The first part of the thesis studies the economic aspects. Previous research focused on designing efficient virtual network architectures and technologies as well as devising algorithms for exploiting the flexibilities introduced by virtualization, e.g., to flexibly embed VNets, i.e., finding a resource efficient mapping of the virtual network on a physical substrate network. Surprisingly, only little is known about the economical possibilities of VNets. Without understanding the economical implications, a VNet market is impossible, as the addition of specifiable network resources and the resulting provisioning of performance guarantees will change the way virtual resources are being sold. There are two major questions in this regard: 1. "What is the impact of different specifications on the resource cost?", as the degree of specificity of a VNet clearly impacts the amount of possible embeddings. And 2. "How can a pricing and buying scheme look like?", given that network guarantees cannot be given free of charge, this is a crucial question.

The diverse routing and forwarding patterns of different VNets sharing the same substrate, introduce additional challenges on the control plane. Especially long-lived VNets will likely be reconfigured due to changing specifications or load balancing. The second part of the thesis studies efficient ways to handle this increased network control load. SDN is a promising approach to solve this issue, as it can be used to provide a logically-centralized control of its virtualized network to each customer. This fine-grained control enables the customer to perform network updates, i.e., changing the forwarding behavior of a set of switches. This is a critical task as it influences the performance, e.g., due to load balancing as well as the security of the network. However, even though SDN provides a logically centralized view on the network and hence a better control to execute the network updates, it does not inherently prevent a network update from violating consistency properties leading to network outages in the worst case. Hence, we do address the following issue: "How to update a network in an efficient manner while adhering network consistency?"

1.2 Contributions

The contributions of the thesis can be grouped accordingly.

Economic Implications of a VNet Market

We analyze how flexible specifications of WAN scenarios can be exploited to improve the embedding of virtual networks. We define a measure for specificity and introduce the notion of the *Price of Specificity (PoS)* which captures the resource cost of the embedding under a given specification.

We present a demand-specific pricing model called DSP for virtual clusters (a special form of VNets in datacenters) which is fair in the sense that customers only need to pay for what they asked. In addition, we present an algorithm called TETRIS that efficiently embeds virtual clusters arriving in an online fashion, by jointly optimizing the node and link resources.

We study cost-effective cloud resource allocation algorithms under *price discounts*, using a competitive analysis approach. We show that for a single resource, the online resource-renting problem can be seen as a two-dimensional variant of the classic online parking permit problem, and we formally introduce the PPP^2 problem accordingly. We present an online algorithm for PPP^2 that achieves a deterministic competitive ratio of k, where k is the number of resource bundles, which is almost optimal, as we also prove a lower bound of $k/3$ for any deterministic online algorithm.

Dynamic Network Updates

We provide a model which vastly simplifies the problem size of finding consistent network update schedules. We show that for fast dynamic network updates, i.e., sending out updates in rounds, where in each round only a subset of switches are updated, it is important to minimize the number of rounds: the interactions between the controller and the switches. We first prove that this problem while ensuring loop-freedom (LF) is difficult in general: The problem of deciding whether a k-round schedule exists is NP-complete already for $k = 3$, and there are problem instances requiring $\Omega(n)$ rounds, where n is the network size. Given these negative results, we introduce an attractive, relaxed notion of loop-freedom. We prove that $O(\log n)$-round relaxed loop-free schedules always exist, and can also be computed efficiently.

After studying loop-freedom, we introduce a most basic safety property, *Waypoint Enforcement* (WPE): each packet is required to traverse a certain checkpoint (for instance, a firewall) and show that WPE can easily be violated during network updates, even though both the old and the new policy ensure WPE. We show that WPE may even conflict with LF and evaluate the computational complexity: In particular, we show it is NP-hard to determine if a scenario is solvable.

1.3 Structure of the Thesis

The thesis is structured in two parts. The first part focuses on the economic implications of a VNet market, while the second part studies fast and consistent network updates as an enabler for the VNets.

Chapter 2 gives an overview of the background regarding VNet embeddings and a potential economic architecture. Although the embedding problem is a central aspect of the network virtualization paradigm and has already been studied frequently in related work, we throughout the thesis focus on the economic implications.

Depending on the needs of a customer, a VNet's specification can differ drastically from a fully specified VNet to a VNet which only specifies some key parameters. In Chapter 3, we first consider the most flexible scenario, i.e., a WAN scenario where customers can choose freely how detailed they specify their VNets. We study the impact of the degree of their specifications on the embedding costs. This is done by defining a measure for specificity and the Price of Specificity as the ratio of increased resource cost compared to a fully flexible specification. To avoid bias introduced by suboptimal embeddings, we use a mixed integer program to compute optimal embeddings in terms of resource cost. Since datacenters are often under the control of a single entity, in such settings, VNets are likely to be introduced first. Many jobs within a datacenter are running under tight timing constraints and have deadlines. We evaluate the impact of their time constraints and different pricing levels in a competitive distributed datacenter setting.

While pricing schemes nowadays mainly focus on VM-pricing only, Chapter 4 studies ways to include network resources into the pricing. Given the heterogeneity of jobs within a datacenter we specifically focus on jobs where network and compute resources might differ in between the requests and indicate that given embedding algorithms do not efficiently include this heterogeneity. The proposed pricing scheme can result in discounts depending on the remaining resource capacities inside a datacenter. Hence, we study how to buy resources under discounts from a resource broker perspective given unpredictable demand. We show that this problem is similar to the classic online parking permit problem and present an online algorithm which is almost optimal.

In the second part of the thesis, we are interested in how to efficiently enable VNets and, therefore, a VNet market. The SDN paradigm offers ways to control the traffic in a more fine-grained way and is hence suited to help VNets becoming a reality. As the introduction of flexible VNets may lead to a lot of dynamics and routing changes in the network it needs to be updated fast and consistently. We give an overview of the network update background in Chapter 5, where we explain the basic principle of SDN and its most prominent representative *OpenFlow*. Furthermore, we discuss the strengths and weaknesses of different ways of updating a network and some of the consistency issues. We give a detailed overview of the model for dynamic networks in Chapter 6, where we introduce different ways of representing the problem as well as possibilities to reduce the (visual) complexity during the update.

In Chapter 7, we focus on consistent network updates. Here we are specifically interested in loop-free updates in a dynamic fashion, without having to wait for all switches to be updated before we see first effects on the network. This also has the benefit that we do not need tags to differentiate between old and new forwarding rules. We identify two different notions of loop-freedom (LF): We differentiate between (the classical) strong loop-freedom where no topological loops are allowed and the relaxed loop-freedom where loops are forbidden on paths, which a newly injected packet might take. We mainly study two different objectives for efficiently updating networks under both definitions of loop-freedom: 1. Maximizing the updates per round and 2. Minimizing the total number of rounds. We study their complexities as well as heuristics to solve the update problem efficiently.

The trend of adding virtualized middleboxes to the network also imposes some additional challenges regarding consistent network updates which we study in Chapter 8. Here we introduce the consistency property waypoint enforcement (WPE): that a middlebox, e.g., a firewall, needs to be traversed from every packet. We study the implications of WPE on a network update and how to perform fast updates solely adhering to WPE. Performing a dynamic network update without violating either WPE or LF is not always possible. We prove that it is even NP-hard to determine if a scenario is solvable and evaluate on small- to mid-sized scenarios to what extent these conflicts occur.

Finally, in Chapter 9 we summarize our work and provide directions for future research and open problems.

Part I

Economics of Virtual Networks

2 Network Virtualization Overview

After revamping the server business, the virtualization trend has also started to spill over to networking. The network virtualization paradigm envisions an Internet where customers can request arbitrary VNets from a substrate provider (e.g., an ISP). A VNet describes a set of virtual nodes, which are connected by virtual links; both nodes and links provide certain QoS or resource guarantees. Early ideas on how to ensure bandwidth guarantees across shared links have been studied in QoS routing, e.g., in [30, 107].

The virtualization trend in today's Internet decouples services from the constraints of the underlying physical infrastructure. This decoupling facilitates more flexible and efficient resource allocations: the service can be realized at *any* place in the substrate network, which fulfills the service specification requirements. Thus, computation and storage have become a utility and can be scaled elastically: geographically distributed resources can flexibly be aggregated and shared by multiple customers.

However, network virtualization is about more than just bandwidth: It also facilitates to virtualize the network stack, allowing to experiment or use clean slate network protocols, and hence to overcome the *ossification* of the Internet [13]. Given the flexibilities, virtual networks also introduce new economic opportunities and challenges. This chapter gives an overview of the economic roles within a VNet market. We present abstractions and algorithms for the virtual network embedding problem and discuss some pricing schemes. Khan et al. [67] provide a broader view on the network virtualization concept. For an overview of different flavors of network virtualization, see surveys [32, 46]. Next, we briefly define terms commonly used in the context of virtual networks:

Common Terms.

- *VNet (virtual network):* Set of virtual nodes connected by virtual links.
- *Specification:* Resource requirements for a VNet. These can be, e.g., CPU cores for virtual nodes or, e.g., bandwidth guarantees for virtual links.
- *Substrate:* Physical infrastructure hosting several VNets in parallel.
- *Embedding:* A mapping of virtual nodes and links onto the substrate, meeting the VNets requirements. The quality of an embedding is often evaluated by resource consumption (less is better) or acceptance ratio (higher is better).
- *Provider:* Owns and operates the substrate.

- *Customer:* Requests VNets at the provider or at a broker. Can potentially be a service provider wanting to offer QoS guarantees to its customers.
- *Broker:* Buys (or leases) resources (or VNets) from providers and resells them towards the customers.
- *Datacenter:* Geographically centered network, where the infrastructure is under the control of a single provider. Customers nowadays already rent VMs (virtual machines) in datacenters.
- *WAN (wide area network):* Geographically distributed network, where the infrastructure is often under the control of multiple providers. It connects several networks, e.g., datacenters with each other.
- *VNet Market:* A market where customers can request VNets for any needs. This includes VNets for datacenters, e.g., a deadline relevant compute job, as well as VNets for WAN scenarios, e.g., requiring minimal connectivity between several businesses datacenters. The VNets will be hosted by different providers and either sold by the providers or a broker.

2.1 Economic Roles

Network virtualization has the potential to change Internet networking by allowing multiple, potentially heterogeneous and service-specific virtual networks (VNets) to cohabit a shared substrate network. This also impacts the roles participating in a network virtualization market.

We envision a network virtualization environment where services are offered and realized by different *economic roles*, as for example proposed in [103]. In a nutshell, the authors assume that the physical network, or more generally: the *substrate network* is owned and managed by one or several *Physical Infrastructure Providers* (PIP) while virtual network abstractions are offered by so-called *Virtual Network Providers* (VNP). VNPs can be regarded as resource *brokers*, buying and combining resources from different PIPs. The virtual network is operated by a so-called *Virtual Network Operator* (VNO). Finally, there is a *Service Provider* (SP) specifying and offering a flexible service.

In such an environment, a service is realized in multiple steps, and the application or service specifications are communicated from the SP down the hierarchy to the PIP. While the SP may specify the service on a high level (e.g., regarding maximally tolerable latencies experienced by users accessing the service), the specification is transformed by the VNP to a VNet topology describing which virtual node resources should be realized by which PIP and how to connect; in other words, the VNet is embedded by the VNP on a graph consisting of PIPs. The PIP then transforms the specification into a concrete VNet allocation / embedding on its substrate network.

We simplify this model a bit, by having a *provider* owning and managing its physical infrastructure, a *broker*, which buys and sells resources as well as a *customer* who is interested in the resources to deploy a service as a service provider or run, e.g., datacenter jobs. Throughout the thesis, we study both, scenarios where a broker exists and scenarios where customers interact directly with the provider. For our economical analysis in the first part of the thesis, we assume that the provider is operating the networks and hence, leave out the roles of the VNP and VNO.

2.2 VNet Embeddings

At the heart of network virtualization lies the promise of a more efficient utilization of the given infrastructure and its resources by sharing it among multiple VNets. The problem of finding good VNet embeddings [87] has been studied intensively over the last few years. Even though the focus of the thesis is not to enhance VNet embedding algorithms, we give an overview of the most relevant models and results since a VNet market cannot become reality without efficient embeddings.

A VNet consists of virtual nodes and virtual links, which require certain amount of resource guarantees, e.g., CPU, RAM and disk capacity for end systems and a maximal latency or minimal bandwidth guarantees on the network. Throughout this thesis, it is sufficient to associate virtual nodes with virtual machines.

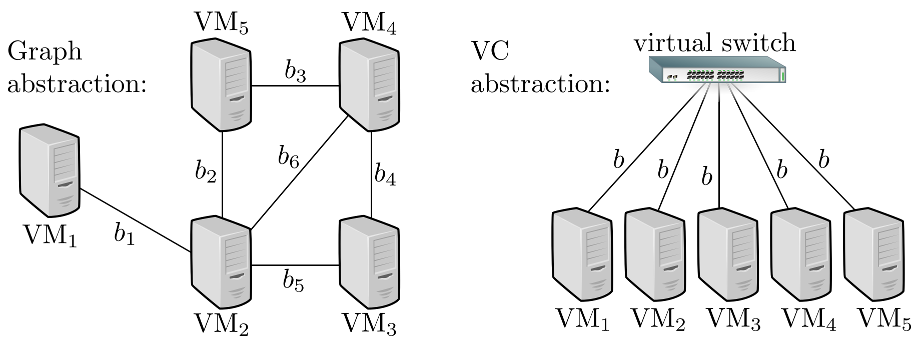

Figure 2.1: VNet abstractions. *Left:* VNet specified according to the graph abstraction. *Right:* (Different) VNet specified according to the VC abstraction.

The physical infrastructure or substrate network hosts the virtual resources. In the context of virtual nodes one can typically think of an $n:1$ mapping of virtual nodes to physical nodes, as the physical nodes are typically large machines with several VM slots enhancing the idea of efficient resource usage. The situation is different for virtual links. A single virtual link has to be mapped to a path of physical links, shared with other virtual links and hence, renders it an $n:m$ mapping.

There are several different abstractions to specify a virtual network from which we discuss three next:

1. *Graph abstraction.* The most general approach is to define network connectivity on a per node pair base in a *graph abstraction*, which is used in most WAN scenarios [57]. In the datacenter context this is used in, e.g., *Secondnet* [55]. Each node and each link has to be specified individually, providing the maximum amount of customization. See Figure 2.1 (*left*) for an example. We refer to this as the graph abstraction.

2. *Hose abstraction.* A simplified version to specify VNets is to only define the bandwidth b per single node, i.e., a node's aggregated incoming and outgoing traffic cannot exceed b. This abstraction is introduced in *Gatekeeper* [97] and is based on the *hose model* initially introduced in the context of VPNs (virtual private networks) [38]. In the hose model, each node requires capacities in the amount of the sum of the in/out traffic to each other node.

3. *VC abstraction.* A subversion of the hose abstraction, the *VC* (virtual cluster) abstraction is presented in *Oktopus* [18]. A VC is defined by a set of virtual nodes, which are connected with a guaranteed bandwidth to a single switch. This model provides a simplification, as the requested resources in this abstraction are typically identical over all the nodes and their connectivity, which is motivated by multiplexed jobs of larger scale. An example of a VNet specified according to the VC abstraction can be seen in Figure 2.1 (*right*).

Mogul et al. [86] give a good overview of different approaches that can increase datacenter utilization including proposals such as Secondnet, Gatekeeper and Oktopus. They all have in common that they address the problem of unpredictable network performance. This unpredictability is caused by other customers within the datacenter, whose jobs can impact the jobs of others and, therefore, their performance. Since bandwidth is usually not guaranteed by current cloud providers, Li et al. have shown that intracloud performance can differ significantly between different cloud providers [74].

In Section 3.1, we focus on the graph abstraction, where we study the impact of such a detailed specification. In Sections 3.2, and 4.1 we turn towards VNets in datacenters. We study the specificity impacts on a distributed datacenter market and propose a pricing scheme based on the VC abstraction. See Figure 2.1 for an example of VNets in both models. In the graph abstraction, different resources can be specified on the VMs as well as on the links, e.g., bandwidth $b_1, \cdots, b_5$. In the classical VC abstraction, a number of identical VMs is connected to a virtual switch via uniform bandwidth b. The same distinction can be made in terms of compute resources, where all VMs of a single virtual cluster have a uniform demand, whereas in the graph abstraction those can be independently specified.

Finding efficient VNet embeddings is an important aspect of network virtualization. More efficient embeddings yield a higher resource utilization and hence, a higher acceptance ratio. This can ultimately lead to higher revenues for providers and, thus, presumably lower prices for customers. Unfortunately, the general VNet embedding problem and even some simplified subproblems where nodes are already embedded are NP-hard [46, 57]. The problem is related to network design [108], virtual circuit planning [16], and minimal linear arrangement [29]. A broad overview of related work for the general VNet embedding problem can be found in [20].

However, Rost et al. have shown that the classical VC embedding problem, which was assumed to be NP-hard, e.g., in [18], is not NP-hard and they provide an optimal embedding algorithm for general topologies [98]. Based on this, Fuerst et al. [48] extend the VC embedding problem and include data locality. This is beneficial for large datacenter applications which are working on a distributed file system. The authors evaluate the complexity of different scenarios and show that several variations can be solved efficiently.

2.3 VNet Economics

To facilitate VNets, QoS-guarantees on the networks must be provided. Early ideas to QoS-enabled Internet pricings can be found in [93]. Most WANs nowadays are realized over MPLS (Multiprotocol Label Switching) which is up to two orders of magnitude more expensive per Mbit than residential Internet access [3]. This highlights the difficulty of providing such services.

Charging based on network resources is not an easy task, as it is not solely based on the usage of the customer. The load of the network is an important factor as well as the burstiness of the service. Reichl et al. [94] identify key pricing parameters, which are access fees, setup fees and usage fees. These parameters lead to several different pricing schemes for the Internet with the classical flat rate pricing as the most prominent way to charge private customers. In the following, we present two additional exemplary pricing approaches besides MPLS that deal with QoS in the Internet:

- *Paris Metro Pricing* was initially proposed as a pricing scheme for the metro in Paris. It has been adapted to the Internet and offers indistinguishable service classes (*"logically separated channels"*) which are charged differently [89]. The fact that only fewer customers are willing to pay a higher price for the same service leads to less traffic on the more expensive channels. Thus, it provides better service simply based on the pricing.
- *Smart Market* was introduced in [81]. The authors argue for a congestion based pricing system. In addition to a fixed connection charge, a packet that is sent with little to no load on the network has only a very small price. However, as load increases, the price does as well. This can be realized in an auction based market where packets include bids. Customers exceeding the marginal bid are then charged not according to their bid, but to the marginal bid, essentially a *Vickrey auction* (or second price auction).

Nowadays, datacenters pricing schemes are simpler, as they are typically solely based on a per VM basis and do not charge for network resources. This of course also means that there are no resource guarantees on the network, which leads to unpredictable performance and hence, unpredictable cost [100]. In the following we discuss some concepts used in most of the dominant cloud provider pricing schemes:

Pay-as-You-Go: Most provider offer VMs on a per VM per hour base, while some also started to reduce this period to a per minute base, e.g., Microsoft Azure [6]. The customers do not need to opt in long-term contracts and only pay for the VMs they used (on a per hour granularity), which is called a *pay-as-you-go* model.

Discounts: If customers choose to use more resources, they are usually charged a smaller price per unit, e.g., for storage and data transfer. Some providers offer their customers also the choice to reserve instances for a longer time period. The customers are charged an upfront fee, which enables them to have a discount on the per hour usage price and ensures availability to the time they reserved the instances.

Spot market: On Amazon AWS [4] it is also possible to rent spot instances. These instances are typically less expensive than standard instances, but on the downside give no usage guarantee. Customers bid on the spot market with a fixed price that they are willing to pay for VMs. As long as the spot market price for these instances is below the bid, customers will keep their reservation. Once the bid is exceeded, they lose control of the VMs immediately without being charged for the begun hour and thus, making this a low priced alternative for rather uncritical jobs, which do not cause any harm if they fail.

How can these concepts be applied towards VNets? Regarding general VNets (according to the graph abstraction), Hu et al. compare the pay-as-yo-go pricing with a pay-as-you-come pricing (customers specify their demand in advance) in the context of service migration and they give optimal algorithms in an offline setting where the demand patterns are known [61]. Their results indicate that the pay-as-you-go pricing is beneficial for the customers especially in scenarios with only moderate discounts. In the datacenter context, Bazaar [64] introduces a principle where customers have a simpler way to express their needs. Instead of specifically describing their requirements in terms of compute and networking resources, they express their requirements in high-level goals as, e.g., job completion time, as many jobs have deadlines [109]. In addition, Ballani et al. [17] presented a first pricing scheme *DRP* (Dominant Resource Pricing) for VCs which offers a guaranteed minimal base bandwidth for every VM and charges the customer if the base bandwidth is exceeded.

Throughout Chapter 3, we study the impact of specifications such as deadlines and in Chapter 4 we discuss the limitation of DRP and provide a pricing scheme for datacenters as well as buying strategies for brokers under discounts.

3 Harnessing Specification Flexibilities

The network virtualization paradigm enables customers to specify their requirements freely. However, it is likely that not every customer has the detailed knowledge required to specify his VNet completely or to his advantage. In fact, while some customers can completely specify their VNets, others might just give some high level requirements, such as deadlines or some form of QoE (Quality of Experience) description.

As a result, VNets are specified at different levels of details. This creates flexibilities in terms of, e.g., location in WANs or deadlines in datacenters. This flexibility gives the provider the opportunity to embed VMs in different places and on different time schedules and, thus, further improve the efficient resource usage. Less specific means larger degree of freedom and potentially better embedding strategies

This chapter studies the impact of different specifications of VNet on the VNet market. Since our focus is not on the embedding algorithms themselves in this work, we compute optimal embeddings to evaluate the solutions to disregard the penalty of suboptimal embeddings.

We look at two different scenarios. In Section 3.1, we study the impact of specificity on the embedding cost in a WAN scenario where VNets can be freely specified, according to the *graph abstraction* from Section 2.2. Then we turn to a more reasonable near term scenario, VNets in datacenters. After all, there is already a flourishing market of VM leasing. Hence, we study different provider benefits in a distributed cloud market in Section 3.2 based on customer flexibility regarding deadlines, according to the *VC abstraction*.

3.1 Price of Specificity

In a fully virtualized resource infrastructure the location where a VNet is realized (or embedded) *is only restricted by the VNet request specification*. For example, if a customer insists that his VNet nodes run on a 64-bit architecture, the choice of resources is restricted and the VNet may be more expensive to realize compared to a situation where also 32-bit architectures are allowed. Similarly, if a customer requires the VNet to be realized over storage resources in Switzerland only, the VNet embedding can be more expensive than if the requirements are less restrictive and allow, e.g., to exploit Europe-wide storage sites.

This section studies the impact of specificity in the context of network virtualization. We assume a two-player setting consisting of a *customer* (who requests specific VNets) and a *provider*. The customer could for example be a startup company (or even a broker) and requests resources at a provider who operates the substrate network. We assume that the customer specifies certain requirements of the VNet, and the provider will try to realize the VNet in a most resource-efficient manner subject to the customer's specifications. We investigate the tradeoff between VNet specificity and embedding costs. In order to avoid artifacts from heuristic or approximate VNet solutions, our methodology is based on *optimal embeddings*. Accordingly, we present a simple optimal algorithm to compute VNet embeddings, which also supports migration.

Contribution

We present a formal model to measure the specificity ς of a given VNet request and introduce the notion of the *Price of Specificity (PoS)*. PoS(ς) captures the increased embedding cost of a given VNet request of specificity ς. We then identify different types of specifications (such as requirements on resource types and vendor, geographical embedding constraints, or whether migration is allowed after an initial placement), and analyze their influence on the VNet allocation cost. It turns out that the PoS depends both on the substrate size as well as the load of the substrate, while the load has a larger impact than a proportionally similar change of the VNet size; sometimes, the embedding cost can be larger than two (i.e., PoS(ς)>2), even in small settings. Our results also confirm the intuition that the relationship between the distribution of the requested and the supplied resource types is important: While skewed distributions of resources can yield better allocations, they entail the risks of a high PoS if the demand does not perfectly match the supply. Although migration is regarded as one of the advantages of network virtualization and we generally observe positive results in our experiments, we will also present a scenario where migration can also *increase* the resource costs (and hence the Price of Specificity). This is shown in a first formal analysis of the PoS.

We believe that our evaluation not only sheds light onto the resource costs of a VNet in different scenarios (and hence in some sense, the real "value" of a resource) but can also provide insights on how to structure a substrate network in order to increase the number of embeddable networks (and reduce the Price of Specificity) at minimal cost. In this sense, this section serves as a first step towards a better understanding of the economical dimension of the VNet embedding problem, and we provide a short discussion of its limitations and further directions.

3.1.1 Model

We represent the substrate network as a graph $G_S = (V_S, E_S)$ where V_S represents the substrate nodes and E_S represents the substrate links. Also a VNet request comes in the form of a graph (according to the graph abstraction in Section 2.2), represented as $G_V = (V_V, E_V)$ (V_V are the virtual nodes, E_V are the virtual links). This VNet needs to be *embedded* on G_S: each virtual node of G_V is mapped to a substrate node, and each virtual link is mapped to a *path* (or a *set of paths*). Figure 3.1 illustrates an example.

For simplicity and to focus on the specificity, throughout this chapter, we will study undirected and unweighted VNets only, i.e., we assume that each node $v \in V_V$ and each link $e \in E_V$ has a unit capacity. Our approach can easily be extended to weighted VNets.

We will consider the following embedding cost model.

Definition 1 (Embedding Costs). *Let $\Pi(e) = \{\pi_1, \pi_2, \ldots\}$ for some $e \in E_V$ denote the set of substrate paths over which e is realized (i.e., embedded). Let $\omega(\pi)$ for some $\pi \in \Pi(e)$ denote the fraction of flow over path π, and let $\lambda(e)$ denote the length of e in terms of number of hops. The cost of embedding a VNet $G_V = (V_V, E_V)$ on a substrate G_S is defined as*

$$\mathit{Cost} = \sum_{e \in E_V} \sum_{\pi \in \Pi(e)} \omega(\pi) \cdot \lambda(e)$$

In other words, the allocation cost is simply the weighted distance of the different paths used by the virtual edges.

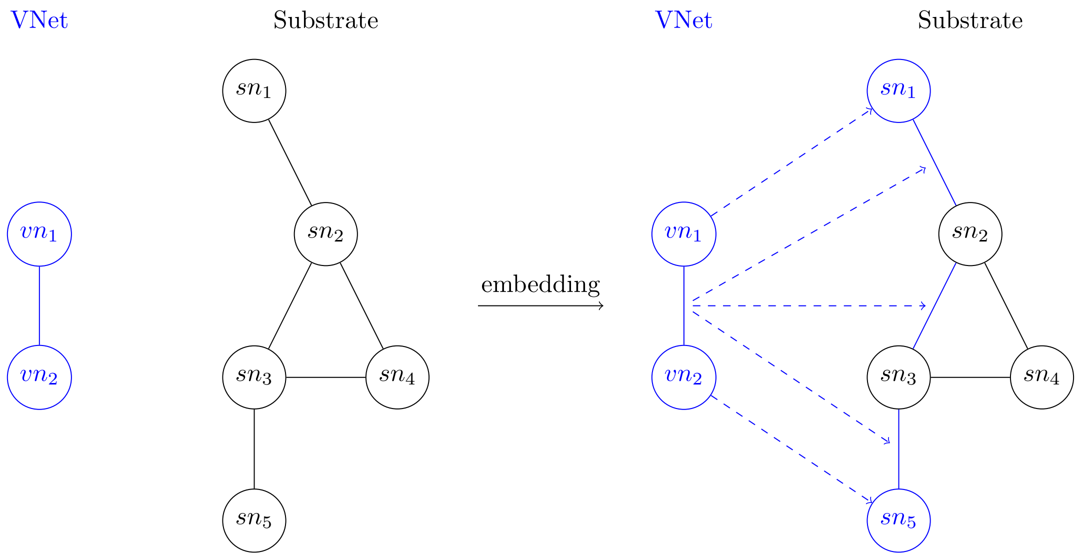

Figure 3.1: Visualization of a VNet embedding: the 2-node VNet on the left is mapped to the 5-node substrate network. Each virtual node of the VNet maps to a substrate node, and for the realization of the virtual link resources are allocated along a path.

3.1.2 Specifying VNets

VNets can be specified in several ways, and the results in this chapter do not depend on any specific language. However, in order to give a concrete example, we quickly review the approach taken in a possible network virtualization prototype architecture; the detailed resource description language called FleRD (for "flexible resource description") is described in [102].

Basically, FleRD *"uses generic description elements and is centered around basic NetworkElement (NE) objects"* (for both nodes *and links*!) *"interconnected via NetworkInterfaces (NI) objects. Keeping these objects generic has the side effect that descriptions of resource aggregations, or non-standard entities (e.g., clusters or providers) is trivially supported. They may be modeled as NetworkElements of an appropriate type and included as topological elements. This may be used, e.g., to describe mappings in the context of a reseller.*

NE properties are represented as a set of attribute-value pair objects labeled as Resource and Features. The meaning of resources here is canonic and they may be shared amongst NEs. Features represent any type of property that is not a Resource (i.e., cannot be described in an amount of units; e.g., CPU flags, wordsizes, supported virtualization mechanisms, geographic locations). Associated Feature sets are interpreted as a logic clause: Features form predicates and sets of Features with corresponding attributes alternatives (disjunctions) within the clause. Features can also be used to explicitly state mapping choices (white listing) or used together with a corresponding attribute to state forbidden mappings (black listing).

FleRD explicitly allows for omission of details irrelevant to the describing entity. (...) Omission is possible both for components and their associated properties." [102]. This kind of language allows customers with any given background to either fully specify their VNets on a very fine grained level or simply focus on some important key properties, while leaving the remaining properties unspecified.

3.1.3 Optimal VNet Allocation

To compute optimal VNet embeddings and to exploit the specification flexibilities, we developed the *FlexMIP* algorithm. FlexMIP is a *Mixed Integer Program (MIP)*, and is described in Figure 3.3. It is a compact variation of the algorithm used in the network virtualization prototype architecture [103].

Concretely, the structure of FlexMIP is as follows. The inputs describe the substrate and the VNet topologies including their capacities and demands respectively. The (originally undirected) substrate graph is given as bi-directed graph $(\mathsf{V_S}, \mathsf{E_S})$, i.e., each undirected edge $\{u,v\}$ is represented as two directed edges $(u,v),(v,u)$. Similarly, for each of the originally undirected requests $r \in \mathsf{R}$ the directed graph $(\mathsf{V}^r_\mathsf{V}, \mathsf{E}^r_\mathsf{V})$ is given. *Substrate Capacities* and *VNet Demands* introduce the functions c_S and d^r_V to denote the substrate node as well as link capacities and the virtual node and link demands for request $r \in \mathsf{R}$ respectively. Based on the bi-directed representation of the substrate $c_\mathsf{S}((u,v)) = c_\mathsf{S}((v,u))$ must hold for all substrate edges $(u,v) \in \mathsf{E_S}$. Furthermore, *Node Placements* defines for each request $r \in \mathsf{R}$ the function l^r_V, which restricts the node placement of virtual nodes $v \in \mathsf{V}^r_\mathsf{V}$ to the set of substrate nodes $l^r_\mathsf{V}(v)$. *Node Mapping* and *Flow Allocation* introduce node and link embedding variables respectively. A VNet link can be split in multiple flows and therefore, be mapped on several links while a VNet node can only be mapped to exactly one substrate node. *Each Node Mapped* ensures that all virtual nodes are embedded on suitable substrate nodes. *Embed Links* guarantees that for each VNet link the needed resources are allocated on all substrate links. Here, we use $\delta^+(s)$ to denote the set of outgoing substrate edges $\{(s,\cdot) \in \mathsf{E_S}\}$ for the substrate node $s \in \mathsf{V_S}$ and denote by $\delta^-(s)$ the set of incoming edges $\{(\cdot,s) \in \mathsf{E_S}\}$: at the substrate node where a VNet link starts, the outgoing traffic has to match exactly the *VEdge Demand*; at the substrate node where the link terminates, the incoming traffic has to match the negative *VEdge Demand*. The links in-between are forced to preserve the traffic and hence, must have the same amount of incoming and outgoing traffic concerning one VNet link. The constraints *Feasibility Nodes* and *Feasibility Links* ensure that the capacity of substrate nodes and links is not exceeded. Note with respect to the links that based on the bi-directed representation of the substrate graph the flow allocations along both edge orientations are considered. The arrow notation used here means the following: if an edge $\overrightarrow{e_s} = (u,v)$ is given, then $\overleftarrow{e_s}$ is defined as (v,u). The *Feasibility Nodes* constraint guarantees that the available resources on substrate nodes are not exceeded. The objective is to minimize the costs which are defined by the sum of all substrate edge allocations. We have additionally introduced the migration cost function m^r_S which defines the (migration) costs for mapping virtual nodes onto substrate nodes. These costs are added later to the objective in the form of the sum $\sum_{r \in \mathsf{R}, v \in \mathsf{V}^r_\mathsf{V}, s \in \mathsf{V_S}} x^r_{v,s} \cdot m^r_\mathsf{S}(v,s)$, when considering such costs.

3.1.4 Defining PoS

To study the Price of Specificity, we consider the following model. We assume that each substrate node $v_S \in V_S$ of $G_S = (V_S, E_S)$ can be described by a set of k *properties* $P = \{p_1, \ldots, p_k\}$, e.g., the geographical *location* (e.g., `datacenter in Berlin, Germany`), the hardware *architecture* (e.g., `64-bit SPARC`), the *operating system* (e.g., `Mac OS X`), the virtualization *technology* (e.g., `Xen`), and so on. The specific property $p \in P$ of v_S can be realized as a specific *base type* $t_{v_S}(p)$. For example, the set $T(p)$ of base types for an operating system property $p \in P$ may be $T(p) = \{$`Mac OS X`,`RedHat 7.3`,`Windows XP`$\}$. (Note that if not every substrate node features each property, a dummy type *not available* can be used.)

Similarly, the VNet $G_V = (V_V, E_V)$ comes with a certain *specification* of allowed types. While the substrate nodes V_S naturally are of specific base types, VNet specifications can be more vague. For example, the types $T(p)$ can often be described *hierarchically* as seen in Figure 3.2: the location `Berlin` can more generally be described by `Germany`, `Europe`, or `?` (*don't care*); or instead of specifying the operating system `Mac OS X`, a VNet may simply require a `Mac`.

Concretely, we assume that each virtual node v_V comes with a specification spec$(v_V) \subseteq T(p_1) \times \ldots \times T(p_k)$ of allowed type combinations for the different properties. The substrate node v_S to which v_V is embedded must fulfill at least one such type combination.

Definition 2 (Valid Embedding). *Let $t(v_S) = \times_{p \in P} t_{v_S}(p)$ denote the vector of types of substrate node v_S. A VNet*
$G = (V_V, E_V)$ embedding is valid if for each virtual node $v_V \in V_V$, it holds that v_V is mapped to a node v_S with $t(v_S) \in$ spec(v_V). In addition, node and link capacity constraints are respected.

Of course, a customer must not specify $t(v_S)$ explicitly by enumerating all allowed combinations: this set only serves for formal presentation. Rather, a customer can specify the types of VNet nodes with an arbitrary

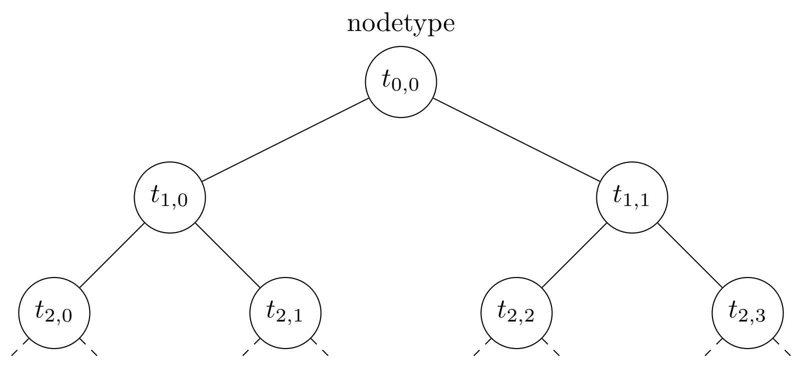

Figure 3.2: Example of a binary hierarchical specification: the VNet node type of a property can be chosen from different specificities. A type of a certain layer allows an embedding on a substrate node with a type of a descendant node. The types for each property of the substrate nodes are always chosen from the leaves.

resource description language, and use *white lists* (e.g., only `Mac`) or *black lists* (not on `Sparc`), or more complex logical formulas.

The question studied in this section revolves around the tradeoff of the VNet specificity and the embedding cost.

Definition 3 (Price of Specificity (PoS) ρ). *Given a VNet G_V, let Cost$_0$ denote the embedding cost (cf Definition 1) of G_V in the absence of any specification constraints, and let Cost$_\varsigma$ denote the embedding cost under a given specificity $\varsigma(G_V)$. Then, the* Price of Specificity $\rho(G_V)$ *(or just ρ) is defined as ρ = Cost$_\varsigma$/Cost$_0$.*

Note that the Price of Specificity ρ depends on the specific embedding algorithm. In the following, we do not assume any specific embedding algorithm, but just use the placeholder $\textsc{Alg}$ to denote an arbitrary state-of-the-art VNet embedding algorithm. (In the related work section, Section 3.1.8, we will review some candidates from the literature.) However, in the simulations, we will use an optimal algorithm *FlexMIP* that minimizes resources.

Although our definition of the Price of Specificity is generic and does not depend on a particular definition of specificity, for our evaluation, we will use the following metric.

Definition 4 (Specificity ς). *The* specificity $\varsigma(v_V)$ *of a virtual node v_V captures how many alternative type configurations are still allowed by a specification compared to a scenario where all configurations are allowed. Formally, we define $\varsigma(v_V)$ as the percentage of lost alternatives: $\varsigma(v_V) = 1 - (|t(v_S)| - 1)/(|T(p_1) \times \ldots \times T(p_k)| - 1)$. The* specificity $\varsigma(G_V)$ *of a VNet $G_V = (V_V, E_V)$ is defined as the average specificity of its nodes $v_V \in V_V$: $\varsigma(G_V) = \sum_{v_V \in V_V} \varsigma(v_V)/|V_V|$.*

Note that $\varsigma(G_V) \in [0,1]$, where $\varsigma(G_V) = 0$ and $\varsigma(G_V) = 1$ denote the minimal and the maximal specificity, respectively. We will focus on scenarios where $|T(p_1) \times \ldots \times T(p_k)| > 1$.

3.1.5 Evaluation

This section studies the Price of Specificity (PoS) in different scenarios. In order to avoid artifacts resulting from approximate or heuristic embeddings, we consider optimal embedding solutions only.

Inputs:

Requests	R	
Substrate Vertices	$\mathsf{V_S}$	
Substrate Edges	$\mathsf{E_S} \subseteq \mathsf{V_S} \times \mathsf{V_S}$	
Substrate Capacities	$c_\mathsf{S} : \mathsf{V_S} \cup \mathsf{E_S} \to \mathbb{R}^+$	
Virtual Vertices	V^r_V	$\forall r \in \mathsf{R}$
Virtual Edges	$\mathsf{E}^r_\mathsf{V} \subseteq \mathsf{V}^r_\mathsf{V} \times \mathsf{V}^r_\mathsf{V}$	$\forall r \in \mathsf{R}$
VNet Demands	$d^r_\mathsf{V} : \mathsf{V}^r_\mathsf{V} \cup \mathsf{E}^r_\mathsf{V} \to\to \mathbb{R}^+$	$\forall r \in \mathsf{R}$
Possible Placements	$l^r_\mathsf{V} : \mathsf{V}^r_\mathsf{V} \to \mathcal{P}(\mathsf{V_S})$	$\forall r \in \mathsf{R}$
Migration Costs	$m^r_\mathsf{S} : \mathsf{V}^r_\mathsf{V} \times \mathsf{V_S} \to \mathbb{R}^+$	$\forall r \in \mathsf{R}$

Variables:

Node Mapping	$x^r_{v,s} \in \{0,1\}$	$\forall r \in \mathsf{R}, v \in \mathsf{V}^r_\mathsf{V}, s \in \mathsf{V_S}$
Flow Allocation	$y^r_{v,s} \geq 0$	$\forall r \in \mathsf{R}, v \in \mathsf{E}^r_\mathsf{V}, s \in \mathsf{E_S}$

Constraints:

Each Node Mapped	$\sum_{s \in l^r_\mathsf{V}(v)} x^r_{v,s} = 1$	$\forall r \in \mathsf{R}, v \in \mathsf{V}^r_\mathsf{V}$
Embed Links	$\sum_{e_s \in \delta^+(s)} y^r_{e_v,e_s} - \sum_{e_s \in \delta^-(s)} y^r_{e_v,e_s} = d^r_\mathsf{V}(e_v) \cdot (x^r_{v_1,s} - x^r_{v_2,s})$	$\forall r \in \mathsf{R}, e_v = (v_1, v_2) \in \mathsf{E}^r_\mathsf{V}, s \in \mathsf{V_S}$
Feasibility Nodes	$\sum_{r \in \mathsf{R}, v \in \mathsf{V}^r_\mathsf{V}} x^r_{v,s} \cdot d^r_\mathsf{V}(v) \leq c_\mathsf{S}(s)$	$\forall s \in \mathsf{V_S}$
Feasibility Links	$\sum_{r \in \mathsf{R}, e_v \in \mathsf{E}^r_\mathsf{V}} y^r_{e_v,\overrightarrow{e_s}} + \sum_{r \in \mathsf{R}, e_v \in \mathsf{E}^r_\mathsf{V}} y^r_{e_v,\overleftarrow{e_s}} \leq c_\mathsf{S}(\overrightarrow{e_s})$	$\forall \overrightarrow{e_s} \in \mathsf{E_S}$

Objective Function:

Embedding Cost	$\min \sum_{r \in \mathsf{R}, v \in \mathsf{E}^r_\mathsf{V}, s \in \mathsf{E_S}} y^r_{v,s}$

Figure 3.3: *FlexMIP:* Embedding constants, variables, constraints and the objective function. Explanations are given in Section 3.1.3.

Setup

In our evaluation, if not stated otherwise, we will focus on the following default scenario. We consider two different properties $P = \{p_1, p_2\}$ with four different types each ($T(p_1) = \{t_1^1, t_2^1, t_3^1, t_4^1\}$, $T(p_2) = \{t_1^2, t_2^2, t_3^2, t_4^2\}$). By default, we do not allow to migrate already embedded VNets. The substrate node types are chosen independently at random from the base types such that each type occurs equally often (up to rounding), and the virtual node types are chosen independently uniformly at random according to the specificity level.

Concretely, we study five different degrees of specificity (in increasing order of specificity): (1) *all* types are allowed (no restrictions, i.e., specificity $\varsigma = 0$); (2) only two types (either $\{t_1^1, t_2^1\}$ or $\{t_3^1, t_4^1\}$) are allowed for $T(p_1)$, but all types of $T(p_2)$ (specificity $\varsigma \approx 0.533$); (3) only two types (either $\{t_1^1, t_2^1\}$ or $\{t_3^1, t_4^1\}$) are allowed for $T(p_1)$ and only either $\{t_1^2, t_2^2\}$ or $\{t_3^2, t_4^2\}$ for $T(p_2)$ (specificity $\varsigma = 0.8$); (4) only one type is allowed for $T(p_1)$ and only two types (either $\{t_1^2, t_2^2\}$ or $\{t_3^2, t_4^2\}$) for $T(p_2)$ (specificity $\varsigma \approx 0.933$); (5) only one type is allowed for each property $T(p_1)$ and $T(p_2)$ (i.e., $\varsigma(v_V) = \{t_1^1, t_1^2\}$ for all nodes $v_V \in V_V$, specificity $\varsigma = 1$). Note that for the upcoming plots we included a linear connection between the data points for better visibility.

Furthermore, we assume that the nodes in the substrate all have a capacity of one unit, and that the links have an infinite capacity. The virtual nodes and links of the VNet have a demand of one unit (no collocation). Finally, we allow the embedding algorithm to split a virtual link into multiple paths.

Our substrate network is generated using the *Igen topology generator* [92]. Our default model uses one hundred nodes. Nodes are generated randomly and we use the clustering method `k-medoids:5` with five clusters (PoPs) based on distance. The nodes in these PoPs are access nodes which are all connected to the PoPs two backbone nodes. These backbone nodes are picked geographically as the most central ones among the access nodes within a cluster. The backbone topology is built by using a Delaunay triangulation connecting a backbone node with other backbone nodes next to it. Thereby the connectivity is preserved since the triangulation includes the minimal spanning tree and alternative paths are created to guarantee redundancy [62].

As for the VNets we will focus on master-slave (i.e., star) topologies. In the following, we will refer to a star with one center node and $x - 1$ leaves as an *x-star*. In most cases we study 4- or 5-stars.

Impact of Substrate Size and Load

We first study the impact of the substrate size and load and consider two different scenarios: (1) an empty substrate network, and (2) a scenario where the substrate nodes already host some virtual nodes. In both scenarios the arriving VNet is a 5-star.

Figure 3.4 (*left*) plots the PoS as a function of substrate sizes for the empty substrate scenario. As expected, since a larger network offers more embedding options, it is more likely that a low-cost embedding can be found, and the PoS is lower on larger substrates. At around one hundred nodes, the embedding is almost perfect for a VNet with specificity $\varsigma = 0.8$ resulting in a PoS of nearly one whereas the PoS for the 20-nodes substrate is almost 1.2. At a specificity $\varsigma = 1$ the PoS is nearly two in the 20 nodes substrate scenario implying that we need roughly twice as much link resources than actually stated in the VNet requirements. As to be expected, the larger substrates have a smaller PoS and the absolute difference is increasing with the specificity.

Let us now consider a scenario where there is already some load on the substrate network. We study a simplified model where x substrate nodes chosen uniformly at random are set to full load, i.e., no virtual nodes can be embedded. We compare the scenario where x out of 100 nodes are already in use to a scenario where the substrate consists of $100 - x$ nodes. The results shown in Figure 3.4 (*right*) look similar to those of the substrate size scenario. For the $\varsigma = 1$ case the more loaded substrates (40-80 nodes in use) have a higher PoS than their representatives in the substrate size scenario. Given the larger substrate with many nodes in use simultaneously, the distances between the free nodes increase. This yields longer paths allocated for embeddings, and therefore a higher PoS. For lower specificities, this is negligible due to the smaller node type variance.

In order to get a better understanding of the impact of load on the PoS we studied a scenario where there are 15 VNets arriving over time. Each of them is embedded by *FlexMIP* without using migration, leading to more

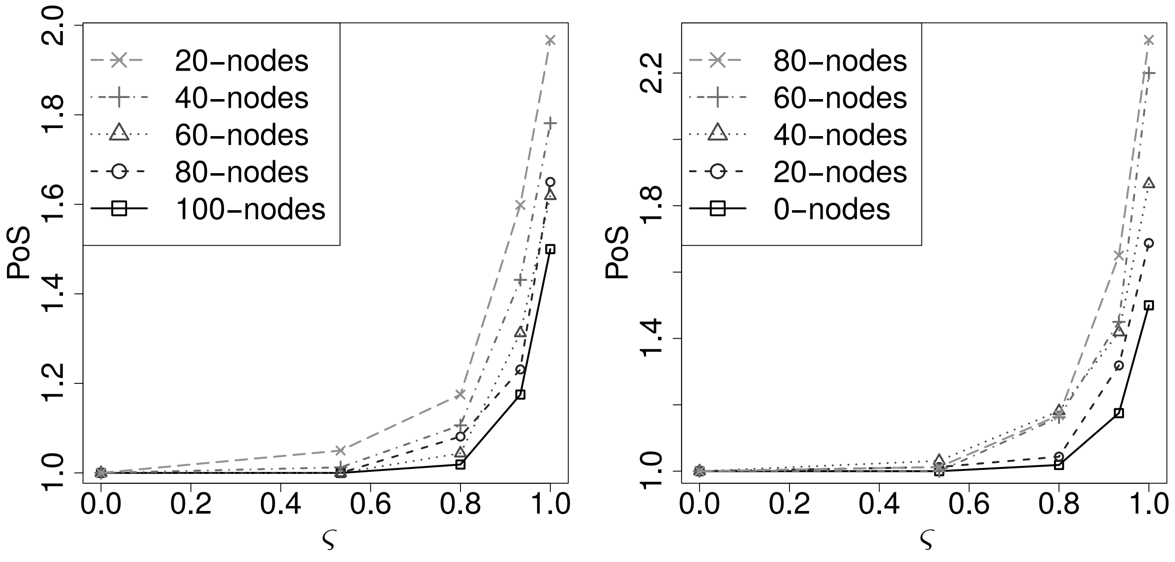

Figure 3.4: *Left:* Impact of substrate size on the PoS. Each substrate was created with the Igen topology generator having five PoPs and two backbone nodes per PoP. *Right:* Impact of load on the PoS. Each scenario is based on the 100 nodes Igen substrate with different numbers of fully utilized nodes. The nodes are chosen uniformly at random.

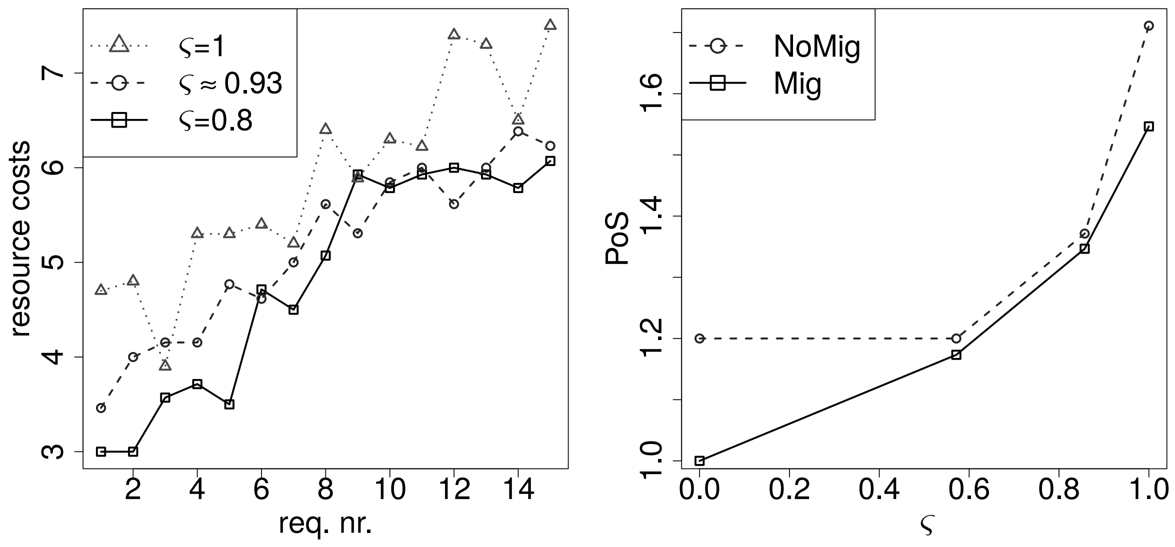

Figure 3.5: *Left:* Amount of link resources needed per embedding as a function of request order. There are 15 incoming 4-star VNets with different ς on a 100-nodes Igen substrate with a substrate link capacity that allows the embedding of two VNet links. For this experiment, we disabled migration. *Right:* Impact of migration on the PoS. There are five 4-star VNets arriving over time on a 40-nodes Igen substrate with eight different substrate node types.

load over time. Figure 3.5 (*left*) shows the amount of link resources needed per embedding depending upon VNet arrival, and the substrate load respectively. While the first incoming VNet can always be embedded perfectly in the $\varsigma = 0.8$ scenario, a higher specificity leads to 3.5 and 4.7 links on average. The link resource costs are increasing with the load and we notice the tendency of lower specificity impacts for the $\varsigma = 0.8$ and $\varsigma \approx 0.93$ scenarios: the curves of the resource costs are converging. Fully specified VNets are still causing more resource costs for each VNet. The impact of the load is shown in the resource costs for the 13th VNet or higher which nearly takes twice as much resources as the first VNet. This especially occurs when the substrate is used close to its capacity. Since a substrate provider will typically try to fully utilize its infrastructure as well as trying to avoid costly embeddings, the PoS has to be understood in relation to the substrate load.

Impact of Migration

The load scenario from Section 3.1.5 was static in the sense that load was modeled on fixed nodes only. However, the possibility to migrate already embedded VNets to more suitable locations is one of the key advantages of the network virtualization paradigm, and hence we now attend to the use of such migrations. Migration can have very positive effects on the PoS: For instance, when a scarce type may have been blocked earlier in time by a VNet of low specificity, a migration may reduce the resource costs significantly. A better location to migrate to may also become available due to the expiration of a VNet.

We study a scenario where five 4-star VNets arrive over time on a 40 node Igen substrate with eight different substrate types. We only study runs where all five VNets have been embedded, resulting in a load of 50% on the substrate. This avoids heavily loaded substrate scenarios as well as scenarios of abundant capacity. Both scenarios naturally lead to no or only small effects of migration due to nearly optimal embeddings. Figure 3.5 (*right*) shows the aggregated PoS over all five VNets. We show averaged values as the embedding costs for an already embedded VNet can change over time in the migration scenario. Interestingly, migration is already effective even without specificity on the VNets (compare PoS Mig:1 - NoMig:1.2). This is due to embeddings which are initially optimal regarding resource costs but use resources that might be more effective in later embeddings, i.e., nodes with a higher degree. While the impact of migration is rather low for the following specificities, it is again recognizable for fully specified VNets.

Generally migration lowers the resource costs and hence the PoS in all our scenarios.

Impact of Type Distribution

The diversity of resources and especially the distribution of requested and supplied types is crucial for the PoS. We expect that in a scenario where the requested types follow the same distribution as the available substrate types, the embedding cost and hence the PoS is lower. In the following, we therefore study different probability distributions for the node types in the substrate as well as the VNets. In addition to the uniform distribution studied so far, we consider a heavy-tailed distribution: a Zipf distribution with exponent 1.2.[1]

Figure 3.6 (*left*) studies five different scenarios: the standard scenario where both (substrate and VNet) types are uniformly distributed, a scenario where both are heavy-tailed distributed and the mixed cases. Additionally we study a scenario where the substrate types are Zipf distributed and the VNet types are *inversely* Zipf distributed in the sense that the least frequent type is the most frequent one in the other distribution. As to be expected, we see that the highest PoS is obtained in the scenario with contrary Zipf distributions. While this scenario is very different from the one with next highest PoS (S-zipf V-uni), the case where both types follow the same Zipf distribution only marginally differs from the scenario where both types are uniformly distributed. Another interesting observation is that the PoS which is obtained in the scenario where the substrate node types are Zipf distributed and the VNet types are uniformly distributed is higher than the PoS of the opposite combination. This is due to the fact that in a Zipf distributed substrate, there are many node types from which only three or less nodes exist whereas the types are requested at equal proportions. Therefore, the possibility that a scarce and hence relatively far away type is requested is high. The opposite distribution has a lower PoS because even the possibility to have at least two nodes from the most common node type in the request is below 50% and around 20% for three or more nodes. Since there are approximately six nodes per node type in the uniformly distributed substrate, the distances are not that large, resulting in a lower PoS for

[1]E.g., the type distribution in a 100-node substrate with 16 node types is $[28, 12, 12, 8, 8, 5, 5, 5, 3, 3, 2, 2, 2, 2, 2, 1]$.

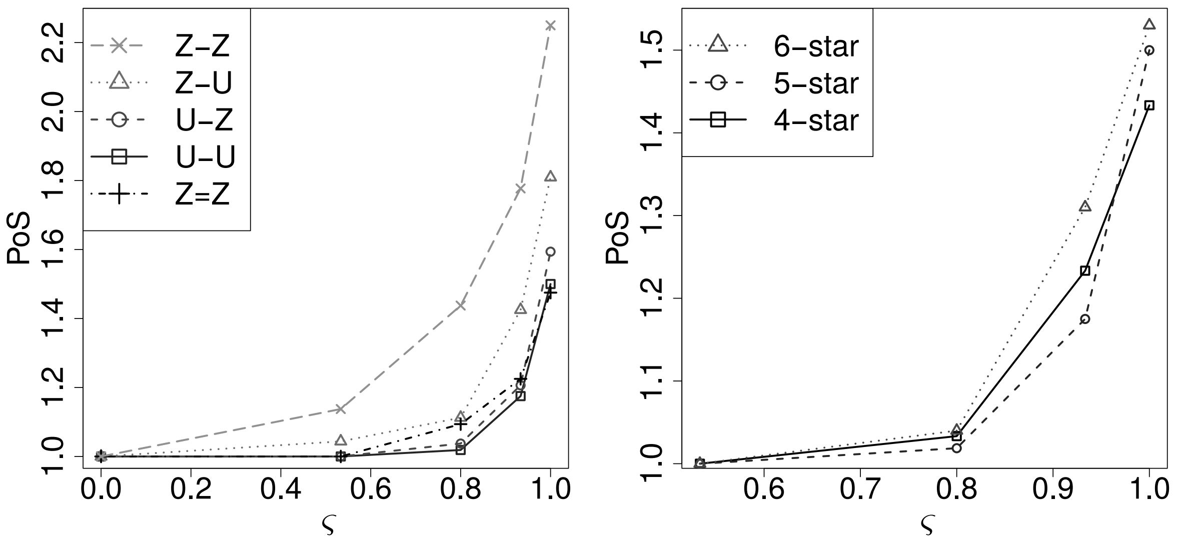

Figure 3.6: *Left:* Impact of different type distributions on the PoS. We compare all combinations between uniformly and heavy-tailed distributed types as well as a scenario where two heavy-tailed distributions have inverse type frequencies (i.e., the most frequent type becomes the least frequent type). As a heavy-tailed distribution Zipf was chosen with exponent 1.2. (legend: Z=Zipf, U=uniform, Substrate-VNet)*Right:* Impact of different VNet sizes on the PoS via 4-, 5- and 6-star VNets embedded on the 100-node Igen substrate.

this scenario. The same argument holds for the only marginal difference between the scenarios where both distributions are Zipf and where both distributions are uniform.

One takeaway from these results is that a specialization of the substrate entails the risk of a high PoS if the demand does not perfectly fit.

Impact of VNet Topology

Section 3.1.5 has shown how the size of the substrate impacts the PoS, and we expect a similar impact of changing the VNet size. Figure 3.6 (*right*) shows the PoS for different sizes of the VNets regarding the number of nodes. For small specificities ($\varsigma \leq 0.8$) the PoS is almost constant. This is due to the relatively (regarding the amount of types and the size of the VNets) large substrate which is robust against these small limitations on the embeddings. The flexibility can still be maintained and an optimal allocation can be found. For higher specificities, we see similar PoSs for the different star sizes. There is a tendency that for increasing specificity the larger stars have a higher PoS than the smaller ones but those differences are rather small. Therefore we investigate also the effect of changing the topology of a 4-star by adding links.

Figure 3.7 (*left*) shows a scenario where additional leaves of the 4-star are connected one by one. The figure reveals the interesting part of the higher specified VNets only. All scenarios with additional links have a higher or at least equal PoS than the standard star scenario, since the complexity to embed these topologies is increasing. Nevertheless the difference between the most complex topology (4-star+3) and the 4-star is relatively low, with an absolute difference always around 0.2.

We find that the modification of the VNet topologies by adding nodes or links does not have a big impact on the PoS. However, note that this conclusion might be different if the load on the substrate is increased.

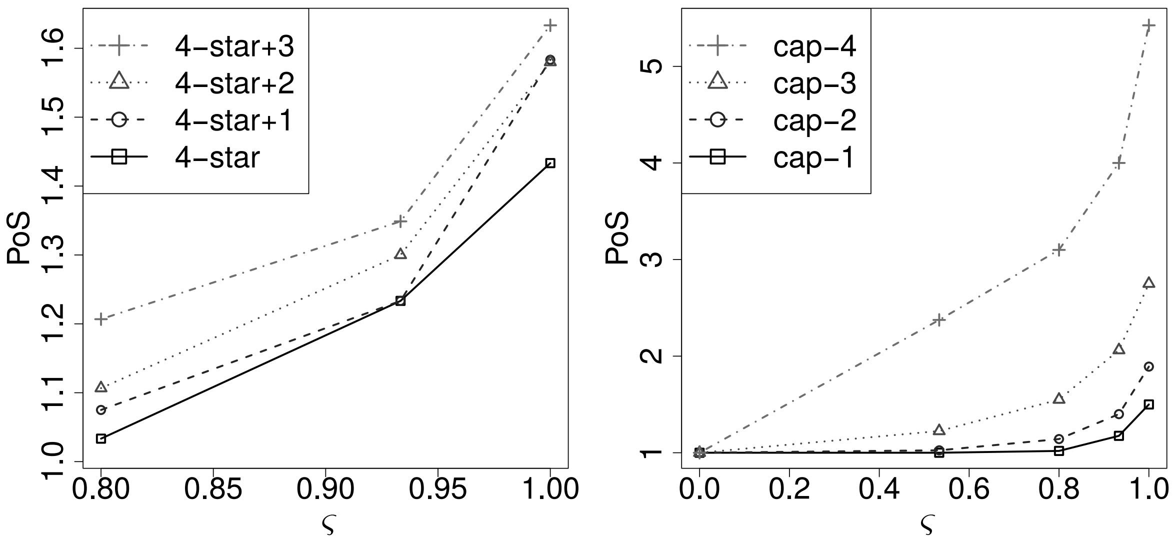

Figure 3.7: *Left:* Impact of additional links on the PoS: we connect the leaf nodes with each other, and 4-star+1 stands for the scenario with a 4-star VNet and two of its leaves connected. *Right:* Impact of substrate node capacities on the PoS. The figure shows the result from embedding a 5-star VNet while changing the node capacity of the underlying substrate from one VNet node per substrate node up to four nodes per substrate node.

Impact of Capacity

Network virtualization allows the substrate provider to embed VNets according to its needs, e.g., to minimize link resource costs in the case of our algorithm *FlexMIP*. The capacities in the infrastructure will not fit exactly the specifications of the VNets in general. Therefore the embedding of several VNet nodes on one substrate node promises lower link resource costs while the restriction on link capacities may yield higher costs.

Figure 3.7 (*right*) shows a scenario where the node capacities are increased up to four units. The curves show a similar trend and there is a higher PoS noticeable with increasing node capacity. This is a consequence of using the substrate nodes at full capacity without VNet node specificities whereas a higher specificity prevents this due to the different VNet node types. A PoS is also observed on specificities where there was no additional costs before. At a specificity of $\varsigma \approx 0.533$, the PoS for the scenario with a node capacity of four is already higher than the PoS of a fully specified VNet in a scenario with node capacity one.

An Out-Sourcing Scenario

While our previous simulations may also describe geographical types (as a property), in this subsection we examine a concrete geographic use-case more closely. Generally we now change the experiments in a way that some VNet nodes assume a fixed position in the substrate. This use-case can for example represent a company which owns certain servers and runs its own network, but out-sources computation or storage to the cloud. While the locations of the infrastructure in the corporate network is given, the out-sourced resources may be specified at different granularity.

Figure 3.8 (*left*) shows the impact of the amount of fixed corporate nodes on the PoS. The 5-star is the incoming VNet basis while we vary the amount of fixed nodes from one to four. In all cases the PoS is calculated regarding the 0-fixed scenario while ς is only referring to the specificity of the remaining free nodes. Noticeable on the first sight is that even for a specificity of $\varsigma = 0$ for the remaining nodes, the PoS is in all cases higher than one. Especially with three or four nodes fixed (meaning only two and one additional nodes

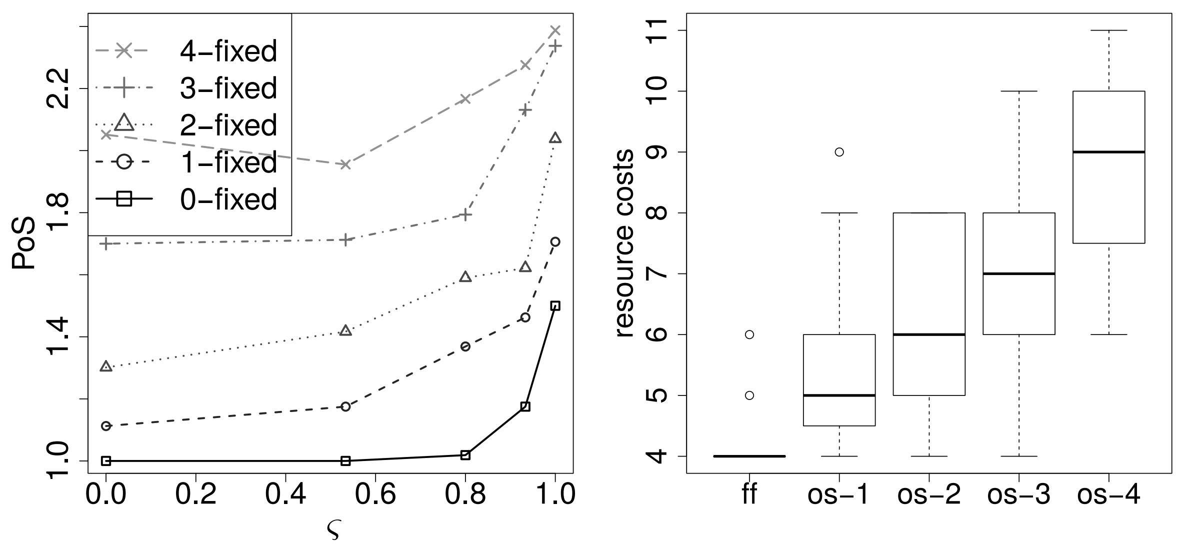

Figure 3.8: *Left:* Impact of different degree of out-sourcing on the PoS. Embedding of a 5-star VNet while changing the number of fixed corporate nodes in it, e.g. three fixed nodes lead to only two free nodes affected by the specificity. Note that the specificity in those scenarios is referred only to the free nodes while the PoS is calculated regarding the 0-fixed scenario without specificity (causing higher PoSs than 1 even for $\varsigma = 0$). *Right:* Boxplot showing the variation for the fully flexible scenario (ff - zero nodes fixed) and four different out-sourcing scenarios (os1 - one fixed node, etc.) for a specificity of $\varsigma \approx 0.53$.

to be embedded) the PoS reaches ~ 1.7 and ~ 2.05 respectively. With an increasing specificity the PoSs of the different scenarios are converging. This is due to the declining effect of the positioning of the outsourced nodes and the increasing effect of the specificity. Keep in mind that this effect might be different in different use cases, e.g., if the fixed nodes are positioned next to each other and not picked uniformly at random of the substrate.

The impact of the fixed node locations can also be understood as a function of the link resource costs of the out-sourcing scenarios. Figure 3.8 (*right*) shows a boxplot comparing these costs with each other for a specificity of $\varsigma \approx 0.53$. In the fully flexible scenario the VNet is almost always embedded with the smallest possible amount of link resources while an increasing number of fixed nodes also increases the variance as well as the used link resources. Nonetheless, an optimal embedding regarding the link resource costs can be achieved even in the out-sourcing scenarios provided an appropriate random placement of the fixed nodes, which can be observed in the scenarios with one to three fixed nodes.

3.1.6 Excursion: Use of Migration

An intriguing question regards the impact of migration on the Price of Specificity. At first sight it may seem that migration can only be beneficial (see also our simulation results in Section 3.1.5). However, we will show that this is only true for scenarios where links have unbounded capacities. Otherwise, there exist situations where migration can be harmful. We will refer to this phenomenon as the *Migration Paradox*.

In order to study the impact of migration on the PoS we extend the Definition 3 towards several embedded VNets.

Definition 5 (Price of Specificity ρ for multiple VNets)**.** *Given a sequence of VNets* $\mathcal{G}_k = (G_V^1, ..., G_V^k)$*, the PoS* $\rho(\mathcal{G}_k)$ *is defined as the average PoS over all VNets* $\rho(\mathcal{G}_k) = \sum_{i=1}^{k} \rho(G_V^i)/k$.

Given the PoS definition for a sequence of VNets, we will make the following assumptions: (1) We focus on embedding algorithms $\textsc{Alg}$ which greedily accept all incoming VNets if possible, while trying to minimize the corresponding embedding costs. (2) Migration itself is only causing no/negligible costs regarding the PoS. (3) The substrate links have unbounded link capacities.

Certainly, all scenarios satisfying these assumptions can only benefit from migration.

Theorem 1. *In scenarios satisfying Assumptions (1)-(3), migration can only decrease the overall PoS of the embedded VNets, meaning $\rho_1(\mathcal{G}_k) \leq \rho_0(\mathcal{G}_k)$ for any sequence of VNets $\mathcal{G}_k$, with ρ_1 representing the PoS for the migration scenario and ρ_0 representing the one without.*

Proof. Let $\mathcal{G}_k$ be a sequence of VNets (with $\mathcal{G}_k = (G_V^1, ..., G_V^k), k \in \mathbb{N}$) requested at the substrate one after another. Note that due to Assumption 1 and 3, an identical order of incoming VNet requests results in exactly the same VNets that will be accepted in both scenarios, meaning that there are valid embeddings for the same requests (cf Definition 2). Since only unavailable node types can cause a VNet to be rejected, it is sufficient to show that for an arbitrary sequence of embedded VNets the link resource usage is not exceeding those of the scenario without migration. The notation $\mathcal{G}_{k'},\ k' \leq k$ describes such a k'-tuple of embedded VNets from an incoming sequence $\mathcal{G}_k$.

The proof is by induction over the number of requests. Take an arbitrary sequence $\mathcal{G}_{k'} = (G_V^1, ..., G_V^{k'})$ of embeddable VNets. We will show that $\rho_1(\mathcal{G}_{k'}) \leq \rho_0(\mathcal{G}_{k'})$ for each $k \in \mathbb{N}$. For $k = 1$ the claim holds: The only incoming VNet G_V^1 of $\mathcal{G}_1$ will always be embedded equally in both scenarios. There are no other VNets that can be migrated, hence $\textsc{Alg}$ embeds this VNet in both scenarios identically. Therefore $\rho_1(\mathcal{G}_1) \leq \rho_0(\mathcal{G}_1)$ is satisfied.

For the *induction step ($k > 1, k \in \mathbb{N}$)*, assume that $\rho_1(\mathcal{G}_k) \leq \rho_0(\mathcal{G}_k)$ holds for an arbitrary $k \in \mathbb{N}$. We show that the embedding of an arbitrary additional VNet G_V^{k+1} still satisfies the claim $\rho_1(\mathcal{G}_{k+1}) \leq \rho_0(\mathcal{G}_{k+1})$. We know that $\rho_1(\mathcal{G}_k) \leq \rho_0(\mathcal{G}_k)$, and an additional VNet request cannot yield higher costs without migration, as the embedding configuration can always be migrated to any possible cheaper configuration. The migration will not increase the PoS due to Assumption 2 and therefore a lower PoS may be achieved. ■

However, note that if link capacities are limited, Theorem 1 no longer holds. Figure 3.9 shows an example where migration can lead to larger resource costs. It shows a simple substrate with a line-topology and link capacities of one. There are only two different types of nodes in the substrate, namely *A* and *B* and a small VNet with two connected nodes of type *B* embedded. This VNet has to be migrated in order to embed a VNet with two connected nodes of type *A*. Therefore the new embedding is using all of the remaining link capacity and prevents the embedding of additional VNets.

Thus, we have the following result.

Theorem 2. *Generally, there are scenarios where migration can* increase *the PoS.*

Therefore it is necessary to have a proper access control even though the provider is able to migrate certain VNets.

3.1.7 Economical Aspects of PoS

This section has mainly studied the price of specificity from a resource cost perspective. Of course, the real cost of a VNet (as paid by a customer) may depend on many additional factors. We round off this section by a short discussion of some other aspects we deem relevant in this context.

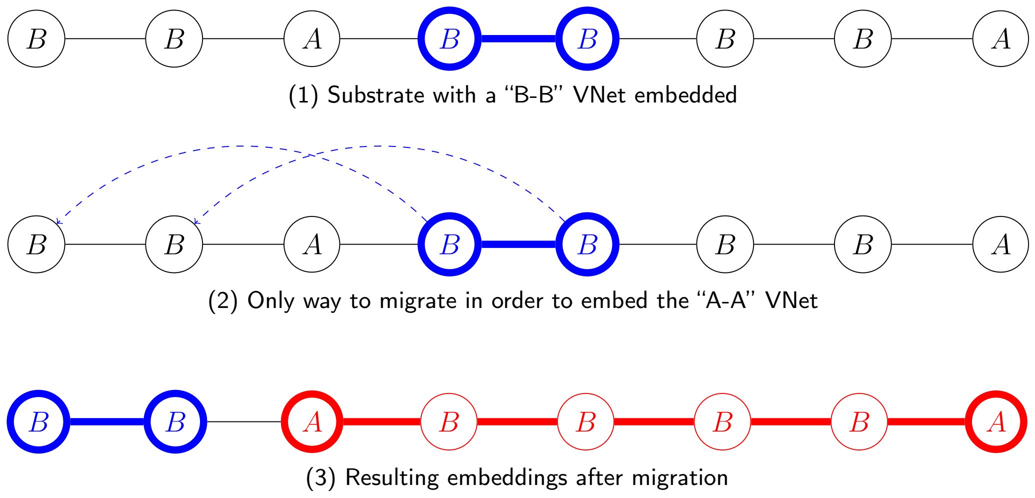
B B A B B B B A
(1) Substrate with a "B-B" VNet embedded
B B A B B B B A
(2) Only way to migrate in order to embed the "A-A" VNet
B B A B B B B A
(3) Resulting embeddings after migration

Figure 3.9: Blocked resources due to enabled migration. After migrating the "B-B" VNet and embedding the "A-A" VNet no link capacities are available for further VNet embeddings.

VNet Templates

In a market for dynamic virtual networks which can be requested at short notice and for limited time periods, it can make sense to define *VNet templates*: virtual networks of a certain (e.g., popular) type. For example, a substrate network provider may offer (and even publicly advertise) fully connected VNets for communication intensive applications or tree-like multicast networks for content distribution as a standard solution. This has the benefit that it simplifies or even short-cuts the negotiation phase between customer and provider, which makes the networks cheaper. Given the corresponding demand, a provider can pre-compute the embeddings of such standardized networks, and also the management may be simplified.

Thus, such templates can be seen as a means of the provider to influence the demand, and hence also the Price of Specificity. Moreover, once a certain template becomes popular, the infrastructure provider may even start to optimize its network topology accordingly, see also our insights in Section 3.1.5.

VNet Pricing

How does the Price of Specificity relate to the actual price of a VNet? Intuitively, the resource costs of a VNet of a given specificity represent a measure of how much VNets, on average, can be embedded of the corresponding VNet type. Thus, a natural but simplistic strategy of a physical infrastructure provider could be to charge the different requests according to the Price of Specificity factors.

Of course, this is an overly simplistic approach. The real price of a VNet depends on the current load on the substrate network which effects the required amount of resources to embed this VNet. Moreover, even if the resource cost (including the opportunistic cost of not being able to embed future requests) can be estimated well, the real price of a virtual network will depend on the *market situation*: the demand for VNets, the current supply of resources, and the level of competition between different providers. While the VNet prices may be close to the provider's actual costs in fully competitive markets, the prices will be higher in monopolistic situations. In this sense, due to the focus on resources, we expect the Price of Specificity to be of greater influence on the VNet price in competitive markets and hence, study such a situation for a distributed datacenter setting in Section 3.2. In Section 4.1 we will then propose an actual pricing scheme for VCs in datacenters.

3.1.8 Related Work

The important problem of finding good VNet embeddings [87] has been studied intensively, both from offline [78] and online perspectives [19, 42], by focusing on bandwidth constraints only [43], by pursuing heuristic approaches without admission control [113], or by employing simulated annealing techniques [96]. The survey by Belbekkouche [20] provides a nice overview of allocation and embedding algorithms. Lischka and Karl [76] present an embedding heuristic that uses backtracking and aims at embedding nodes and links concurrently for improved resource utilization. A mixed integer program for the embedding of certain types of VNets is formulated in [33]; the formulation has been extended to support migration and reconfiguration in [101]. Bienkowski et. al [23] study migration in a mobile network virtualization setting from a competitive analysis perspective. Since the general embedding problem is computationally hard, most of the literature is on heuristical or approximative algorithms. An interesting perspective is taken by Yu et al. [80] who advocate to rethink the design of the substrate network to simplify the embedding which makes it computationally tractable; for instance, they allow to split a virtual link over multiple paths and perform periodic path migrations. The general embedding problem is also related to network design [108], virtual circuit planning [16], or minimal linear arrangement [29].

Our contribution is orthogonal to this line of research. In fact, the Price of Specificity could be studied for each embedding algorithm reviewed above. In order to focus on the main properties of the Price of Specificity, we use a an optimal embedding approach for our evaluation, and ask the question how the VNet specification effects the cost.

There are many networking domains where economical aspects play a central role, for instance in the Internet backbone, in the cloud, in wireless spectrum allocation, or in the grid where storage and computational resources come at a certain price (e.g., [40, 58, 91, 106]). Especially federation (e.g., collaboration of different ISPs or cloud providers) and fairness issues (over multiple resource types) have gained much attention recently (e.g. [49, 105]). For a good introduction to some classic tradeoffs, we refer to the V-Mart [112] paper (and the references therein) which attends to the inter-domain embedding problem and uses an auction-based model, and in the context of computing resources, to the GridEcon a market place proposed in [12].

In the context of the relatively new concept of network virtualization itself, not much work on economical aspects exists yet. Courcoubetis et al. [35] identify incentive issues arising in the management of virtual infrastructures and show that well-designed policies are mandatory to prevent agents from contributing less resources than is desirable. PolyViNE [34] is a decentralized, policy-based inter-domain embedding protocol ensuring competitive prices for service providers. In more general context, Antoniadis et al. [14] employ coalitional game theory to study how participants should share the value of federation in virtualized infrastructures (in the context of ISP interconnections, peer-to-peer systems, the Grid, or cloud computing).

3.1.9 Summary

While today, we have a fairly good understanding of how to realize the vision of virtual networks (cf, e.g. [37]), surprisingly little is known about economical implications. We consider this section as a first step to shed light on the impact of virtual network specification flexibility on the embedding cost. We define the *specificity* of a VNet, which characterizes the amount of choices a provider has in embedding this VNet. Subsequently, the impact of the specificity on the embedding cost is defined by the *Price of Specificity (PoS)*. Based on optimal embeddings, we find that the PoS increases especially for highly specified VNets. Collocation and scenarios with scarce resources yield further areas where the specificity has a large impact on the embedding cost.

We hope that our approach provides a means to reason about policies for VNet pricing and embedding, especially in competitive markets where infrastructure providers operate within small budget margins and study such a competitive market in the datacenter context more closely in Section 3.2.

3.2 Flexibility Beneficiaries in Distributed Cloud Markets

Our analysis in Section 3.1 provides insights on the impact of specification flexibilities on the resource costs. These impacts are studied independently of competing providers. Since a VNet market might very well be a competitive environment, we are now interested in the question: "Who reaps the benefit of the different levels of flexibilities?". To answer this question, we turn to a distributed cloud market in datacenters, in which we study two different market models.

Today's datacenters concurrently host a wide range of applications of different customers with different requirements [60]: while some applications are network-hungry [64] or latency sensitive [110] (e.g., a web service), and may have deadlines [109] and require strict QoS networking guarantees, other applications (e.g., batch processing jobs) are delay-tolerant. For example, a customer may expect that being more delay-tolerant and flexible in terms of resource rates, pays off, i.e., render the service cheaper [10]. Indeed, as the cloud provider may exploit specification flexibilities to schedule applications more flexibly and hence make a better use of its resources, it may share the gains with its customers.

Contribution

This sections investigates on the beneficiaries of specification flexibilities in terms of deadline dependent VNets in a distributed cloud market. We present two simple market models, where customers buy resources either directly at the provider or indirectly via a broker. We find that increased flexibility on the customer side especially benefits the cheaper providers, but not necessarily the cheapest provider. While the social welfare of both customers and providers is increased under heterogeneous flexibilities, inflexible customers might even be worse off.

We provide two basic resource buying strategies for a broker and evaluate the impact of a broker on the market. A broker gets discounts for contracts with a larger resource amount and duration. Out results show that the broker disproportionally benefits from customer flexibilities. Given a higher variance however, the benefits are shifted towards the provider, as resources cannot be bundled efficiently on the broker side.

3.2.1 Model: Horizontal & Vertical Market

It has been shown that cloud applications suffer from resource interference on the network, in the sense that application performance can become unpredictable. Longer job execution times also entail higher costs for the customers who are charged on a *per-VM-hour basis*. [86]

To overcome these problems, more powerful resource reservation models, e.g., Virtual Clusters (VCs) have been proposed [18]. Compared to Section 3.1 where the VNets could be completely specified we consider customers requesting VNets in form of VCs. Each VNet specifies (1) *resource rates* for virtual nodes and links, i.e., a fixed amount of CPU per time or bandwidth (using the *VC* abstraction presented in Section 2.2), (2) a *duration* for which the VNet must be embedded (at the specified resource rate), and (3) possibly a *deadline* by which the VNet must have been embedded for the entire duration (and rate). We propose the following two simplified VNet market models (cf Figure 3.10).

In the *horizontal market model* (Figure 3.10 *left*), the customers (or "tenants") directly request virtual networks (VNets) on demand from different cloud providers. We assume that a customer first issues the VNet request (annotated with the required resources) to all cloud providers, in order to obtain an offer on (1) *when* the VNet request can be scheduled (time t), and (2) at which *price* p. We assume that the providers have fixed but different prices per resource unit, and will greedily schedule a VNet at the earliest possible point in time. Given the time-price tuples (t, p), the customer will choose the best provider offer according a utility function $u(t, p)$. Depending on the application, the customer may be relatively flexible in the execution time as long as the best price is obtained; or, conversely, he or she may be relatively flexible in the price as long as the job is processed as soon as possible.

In the *vertical market model* (broker market), customer requests are handled by a broker (cf Figure 3.10 *right*) that is responsible of embedding the VNets on its virtual resources (similar to the role for the cloud provider in the basic model). The broker role benefits from being able to buy larger chunks of resources as it obtains a discount from the cloud provider. Concretely, we will assume that the broker can buy different *resource contracts* from the cloud provider: a resource contract consists of a resource volume (i.e., an overall resource rate R) and a duration D. The larger the product $R \times D$ of resource volume and duration of the contract (henceforth also referred to as the contract *area*), the higher the discount. Resource discounts are common (e.g., in Amazon's EC2 reserved instances) and yield a tradeoff for the broker: buying too large contracts may be wasteful as the actual resources cannot be resold, and buying too small contracts may yield small discounts. In order to study how the costs of the broker and the income of the cloud provider depend on how flexible the customers are with respect to the *VNet deadline*, we consider different broker strategies.

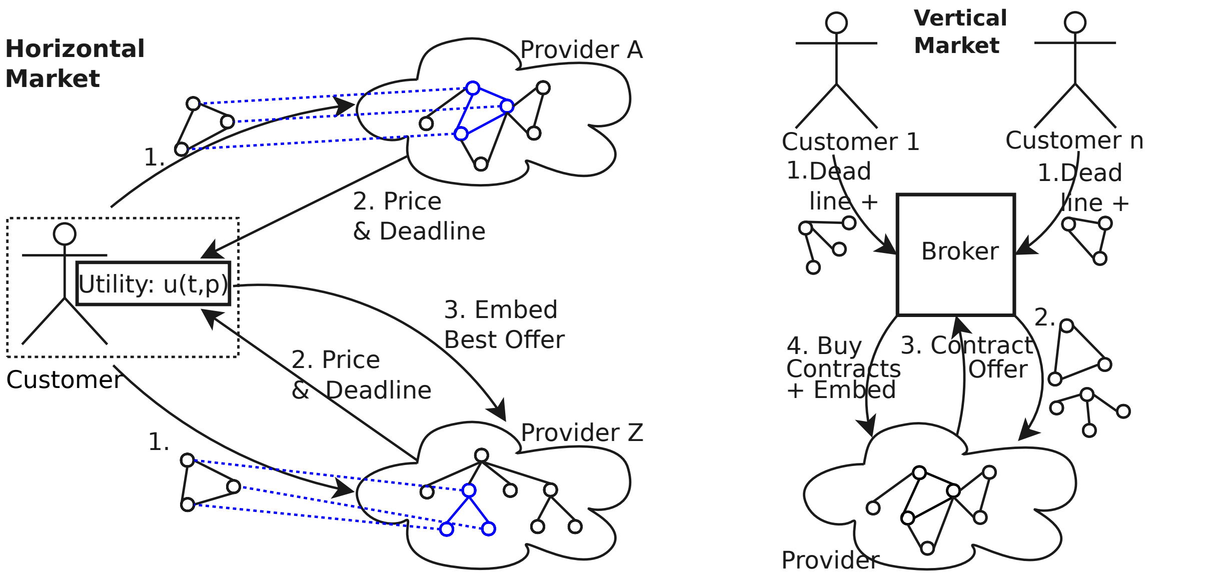

Figure 3.10: *Left:* Horizontal market: A customer requests an offer for a VNet embedding from each provider. Depending on the price-deadline tuples returned by the providers, the best option is chosen according to a given utility function. *Right:* Vertical market: Customer VNet requests (with deadlines) are directed towards the broker which is buying resource contracts (subject to discounts for larger contracts) from the cloud providers.

3.2.2 Benefits in Horizontal Market

We first study how VNet flexibilities influence the income distribution of the different cloud providers. We make the natural assumption that cheaper and faster executions are always preferred over more expensive and longer alternatives. Concretely, we consider the following exemplary utility functions: (1) customers are relatively flexible in time as long as the price is low: $u_f(t,p) = -t - 10 \cdot p$; (2) customers are inflexible regarding time even if this turns out to be more expensive: $u_i(t,p) = -10 \cdot t - p$; and (3) customers are not specifically flexible or inflexible $u_e(t,p) = -t - p$. Finally, (4) we also investigate scenarios where VNets have *strict* deadlines d; i.e., the customer will not accept offers violating this deadline. We will assume that providers have fixed resource prices and schedule a VNet request at the next possible occasion. A VNet request will require a constant resource rate, and cannot be stretched or shortened in time once it is started. Therefore, we use a simple greedy algorithm which computes earliest embeddings on the providers, together with the corresponding prices.

In our experiments, we have three different providers whose prices differ by a certain percentage. VNet requests arrive over time according to a Poisson distribution with exponentially distributed inter-arrival times (parameter $\lambda = 1$). The request durations follow a heavy tail distribution (*Pareto* distribution with exponent $\alpha = 3$, minimum 1, and scaling factor 100) and are chosen independently. The arrival and duration process yields a dynamic demand [22, 31]. For simplicity, the resource requirements of all requested VNets are identical and we will assume that customers request one unit of volume. We assume an underlying fat-tree topology [11] as it is the most common datacenter topology in the literature. As embedding algorithms are not in the focus of the thesis, we assume no oversubscription within the network, i.e., each layer provides the aggregated bandwidth of the layer below them and thus, full bisection bandwidth. Providers have a capacity of 70 units.

Provider Perspective

We first study the impact of different customer flexibilities on the providers. In our model, the revenue of a provider depends on the number of customer requests it will eventually serve. We compare four different scenarios; three *homogeneous* ones where all customers have the same utility function (either *flexible* u_f, *equal* u_e or *inflexible* u_i), and a *heterogeneous* scenario where customers with different utilities (one half uses u_f and one half u_i) compete for provider resources.

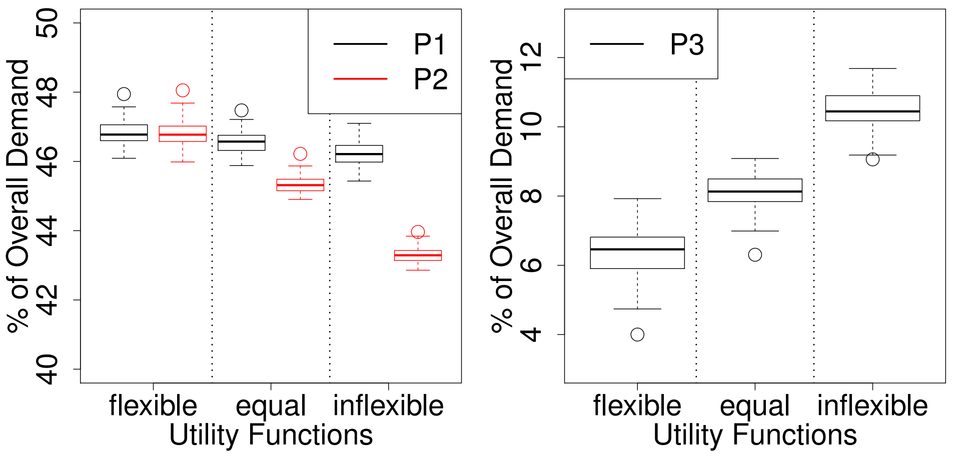

Figure 3.11: Boxplots (*left:* $P1$ and $P2$, *right:* $P3$) of the overall workload, in percentages per provider and under a stable pricing scheme $(100, 110, 120)$. The data is collected over 100 runs with 20k requests each (excluding the first 1k to to remove the "bootstrap phase"). All providers have a capacity of 70 units, the demand is subject to *Poisson arrival* ($\lambda = 1$) and durations are *Pareto* (α=3, min=1, scaling factor=100).

Figure 3.11 shows the percentages of the demand assigned to the three providers: $P1$, $P2$, $P3$ in the three different homogeneous scenarios. In this experiment, the unit price of provider $P2$ is 10% higher than $P1$, and $P3$ is 20% higher than $P1$. Provider $P3$ has a lower workload, as the aggregate demand does not always fill all provider resources. The share of demand on $P1$ changes only slightly over the scenarios. Most of the demand that $P3$ gains (approx. 2% of the overall demand) while increasing the importance of the time dimension, is stolen from the $P2$'s share. That is because customers prefer $P3$ at demand peak rates, rather than having to wait for $P2$. The heterogeneous scenario shows a behavior similar to the equal scenario and is hence omitted here.

Given a certain degree of flexibility (e.g., customers with time uncritical applications such as bulk data transfers or batch jobs), the variance in demand can be exploited to shift load in time. While $P1$'s capacity is completely used $> 99\%$ of the time, the workloads of $P2$ and $P3$ vary. An increase of the *Poisson* arrival rate to $\lambda = 1.2$ leads to a more frequent demand excess, and $P2$ hardly has available capacities over longer time periods. Also $P3$ obtains approx. a quarter of the overall demand. If not stated differently, in this section, we will focus on demands similar to the top scenario where there are sufficient capacities even in high-demand periods.

With increasing demand, having *strict* deadlines becomes more critical: later, a result may no longer be useful. Table 3.1 shows the percentage of requests that cannot be embedded within their deadlines. In this experiment, the deadlines are chosen as a function of the corresponding request duration: we have three different flexibility levels, one adding 1% to the duration, one 10%, and the most flexible one adds 20%. With a Poisson arrival parameter $\lambda = 1.5$, the percentages are nearly equal, and independent of the deadline flexibility. This is due to

		deadline		
λ	median	1%	10%	20%
1.2	182	0.20	0.14	0.01
1.3	193	1.54	0.84	0.1
1.4	209	5.34	4.09	1.89
1.5	224	11.29	10.89	9.82

Table 3.1: Percentage of requests that cannot be served within their deadline. The deadline occurs after the duration plus a percentage (1%,10% or 20%). The arrival process is *Poisson* with parameter λ. We show the median demand (total capacity is 210).

the demand excess for nearly all time periods. In general strict deadlines are beneficial for $P3$, as it increases the fraction of customers that cannot wait.

Customer Perspective

Next we examine the customer perspective. Clearly, the pricing scheme on the market combined with the current resource demand and also the flexibilities of the other customers, will influence the obtained customer utilities.

Time is money. In order to compare different pricing schemes, we examine customers with no specific preference on time or price (utility function u_e). The pricing schemes on the competitive provider market lead to different decisions on how long a request waits for being embedded and what to pay. We investigate the customer utilities on three different pricing scenarios with $1\%, 10\%$ and 25% difference between two providers. Figure 3.12 shows also the respective waiting times. Comparing these plots one must be aware that an increase of the prices also reduces the average utilities. Surprisingly, despite the fact that the cheapest price stays constant over all scenarios, the best utility (no waiting time on first provider) is only reached in the 1% scenario. This is because of the small price difference for customers who do not wait and rather pay the small overhead for embeddings on $P2$ and $P3$. With higher differences, the customers are more likely to wait for free capacities on cheaper providers, which leads to the longer waiting times and queues on those providers, and eventually prevents an immediate embedding there. This also explains why the utilities in the first plot can be classified into three groups with nearly identical utilities. The increasing price variance renders the differences larger and leads to even smaller utilities (up to -50% from the maximum) at peak times, and a wider overall distribution. Unsurprisingly, the waiting time plots show a change of the slope at the points where the time matches its utility-wise analog price value (1, 10, 25). The slope is steeper before these points since there are many customers who prefer to wait for the next cheaper provider.

Dependency on other customers. The utilities obtained by the customers are inter-dependent. To study these dependencies, we extend our setting to a heterogeneous one where the flexibilities of the customers are mixed. The comparison of these utilities (cf Figure 3.13) shows that on the one hand mixed utility functions are beneficial for flexible customers and increase their utility by ~ 10%. On the other hand, mixed utility functions are reducing the utilities for inflexible customers. While in a homogeneous scenario the customers tend to choose a more expensive provider early, which keeps the waiting times low, the flexible customers in the heterogeneous scenario are willing to wait longer for capacities on the cheapest provider. This leads to a situation where the flexible customers reserve the capacities on the cheapest provider while the inflexible customers are stuck with the expensive providers. Since this impacts only approximately 50% of the inflexible customers whose utilities decrease by ~ 10%, the overall utility under heterogeneous demands increases.

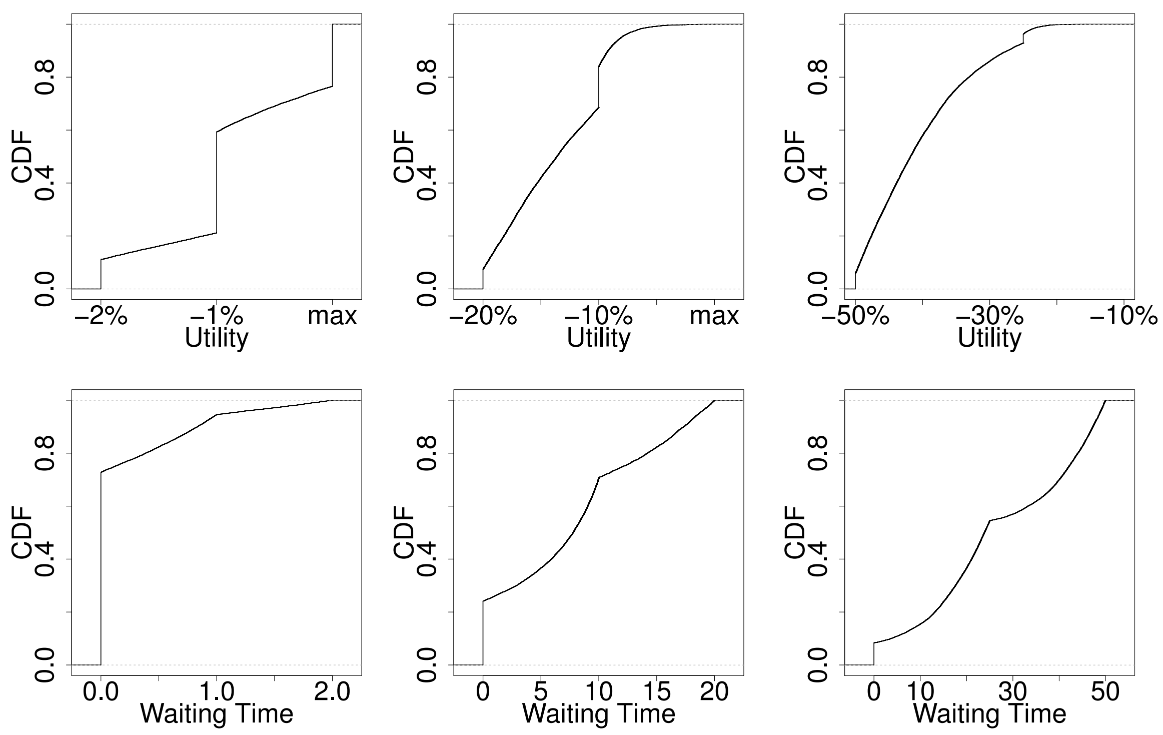

Figure 3.12: CDFs of customer utilities and their respective waiting times given three different pricing scenarios ($1\%, 10\%, 25\%$ price differences between the providers). The utilities are given in a relative percentage to the theoretically highest utility. The customers are not specifically flexible or inflexible.

3.2.3 Benefits in Vertical Markets

Let us study the effects of flexibility in the vertical model: we assume the customers send their VNet requests to a broker who resells resources from the cloud provider. The business model of the broker is to buy large resource contracts from the cloud provider, and uses these resources to satisfy multiple VNet requests. Concretely, we assume that the cloud provider offers a single kind of resource, and that contracts are given in terms of a resource rate R and a duration D. The higher the product of resource rate and duration, henceforth called the *area* $A = R \times D$ of the contract, the lower the unit price. We define the *discount* δ as the factor by which a contract of twice the area is more expensive: $\delta = 1.5$ means that for a twice as large contract, the price is 50% higher; $\delta = 2$ implies no discount is given and $\delta = 1$ means infinite discount.

We assume that each VNet request $vnet$, arriving online at time $vnet$.arrival(), specifies a constant resource rate $vnet$.rate(), a duration $vnet$.dur(), and a deadline $vnet$.dead() by which it must be completed. We focus on a simple deadline utility function in the sense that the customer only cares whether the deadline is met or not, and accordingly pays a fixed price or nothing. Thus, to maximize revenues, the broker should embed as many VNets as possible which meet the deadline. We define the flexibility F of a $vnet$ request as $F = (vnet.\text{dead}() - vnet.\text{arrival}())/vnet.\text{dur}()$, the factor by which the feasible embedding time period exceeds the duration. (We will assume that $F \geq 1$.)

The broker can benefit from delaying VNet requests if the time period until their deadline is relatively large compared to the VNet duration: Then, a larger resource contract can be bought at a better discount. It seems that the broker can reap all benefits from more flexible deadlines relative to the the VNet durations. Moreover, the flexibility gains accruing at the broker translate into a corresponding income loss at the cloud provider, as contracts become cheaper: a zero-sum game. However, as we will see, the situation is slightly more complicated and the benefits depend on the variance in demand. Moreover, the distribution of benefits

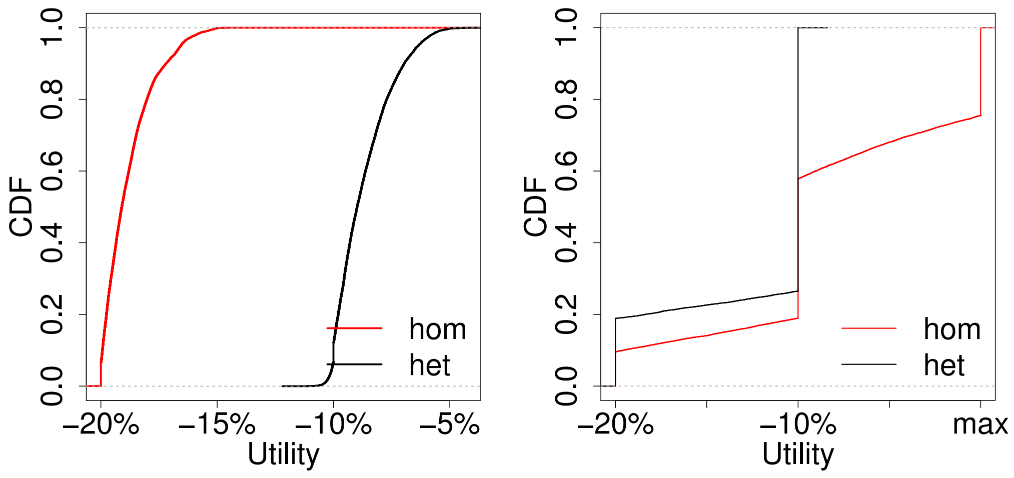

Figure 3.13: CDFs of customers utilities in scenarios with heterogeneous and homogeneous demands. The utilities are compared based on the customer flexibilities (*left:* flexible only, *right:* inflexible only). The utilities are given as a percentage of the theoretical maximum utility.

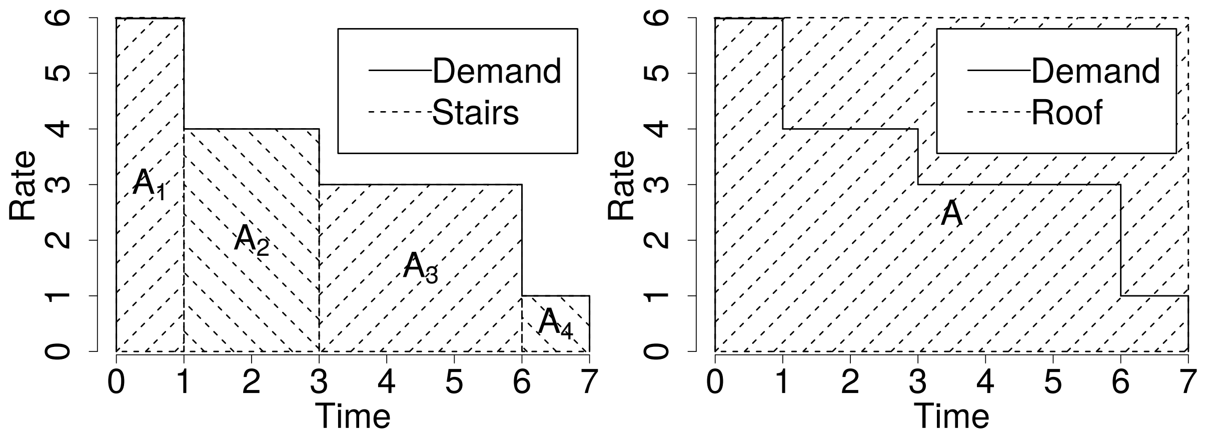

Figure 3.14: Contracts bought by STAIRS and ROOF for a given VNet buffer B.

of course also depends on the strategy by which the broker delays requests and buys resources. We compare three natural broker strategies: A greedy strategy called INSTANT where the broker immediately buys a new contract specifically for each incoming VNet request, and two strategies ROOF and STAIRS where the broker uses a VNet buffer B to delay requests and buy larger contracts.

STAIRS and ROOF differ in when and how the buffer B is filled and Figure 3.14 illustrates how STAIRS and ROOF partition the VNet buffer B into contract. Whenever a VNet $vnet$ arrives, STAIRS includes it in the buffer B. For each time step Δt, STAIRS checks if the VNets in B can be further delayed. If one of the VNet requests $vnet \in B$ can no longer be delayed and must be embedded the latest at the current time t, STAIRS groups the VNets in the buffer into *intervals*, all starting at t, and buys contracts for all these networks. In contrast, ROOF first checks if there are spare capacities available from previously bought contracts. If this is the case, these capacities are used for the embedding of $vnet$. Otherwise, the broker delays the embedding of $vnet$ and adds it to the buffer B. If one of the VNets in B cannot be delayed any longer, ROOF buys a single large contract (i.e., it empties the entire buffer).

Figure 3.15 (*left*) plots the total contract price paid by ROOF and STAIRS, as a function of the contract discount and relative to the INSTANT price which serves as a baseline. We assume the discount model for the product of contract duration D and resource rate R, i.e., for the area $A = R \times D$: for a twice as large area, the price is between one and two times higher (the former implies free resources, the latter no discount). As expected, ROOF performs bad without discounts and benefits from buying large contracts if discounts are high. STAIRS is always at least as good as INSTANT (equal in the no discount scenario). However, under high discounts, STAIRS pays relatively more again. This can be explained by the overall decreased cost of INSTANT.

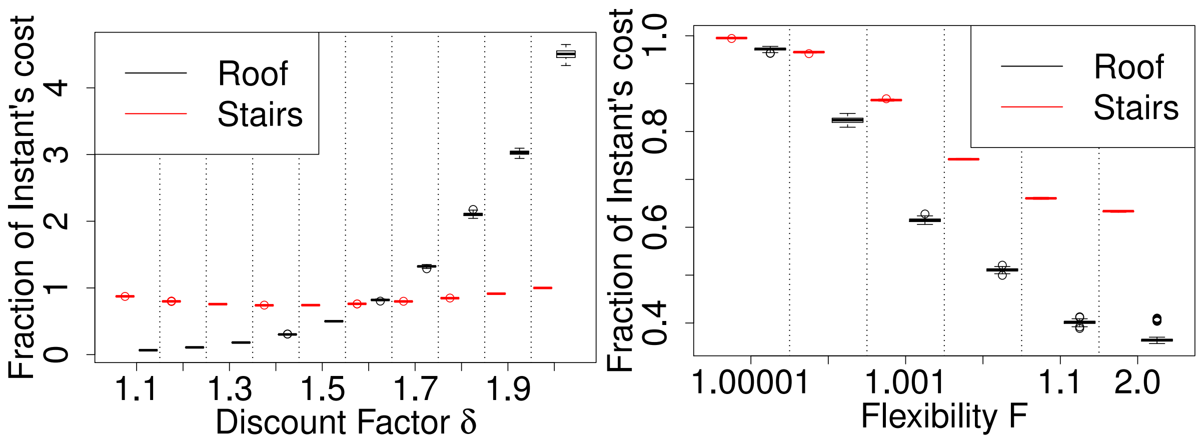

Figure 3.15: *Left:* Price paid by STAIRS and ROOF relative to INSTANT's price as a function of the discount factor for twice as large contracts. The arrival times are generated using a $\lambda = 1.2$ *Poisson* distribution and the durations are generated according to a *Pareto* distribution with $\alpha = 2$. The flexibility factor is $F = 1.01$. *Right:* Price paid by STAIRS and ROOF relative to INSTANT's price as a function of F. (Arrival times $\lambda = 1.2$, durations $\alpha = 2$, discount factor $\delta = 1.5$)

Broker Perspective. Figure 3.15 (*right*) shows the costs of ROOF and STAIRS relative to INSTANT's costs. Note that since INSTANT does not benefit from flexibilities (it does not delay any requests), INSTANT's costs can be used as a baseline and the fraction of its costs can be regarded as the *benefit of flexibility*. This benefit is plotted as a function of the flexibility ratio (overall feasible time period divided by request duration). We see that for very low flexibilities, the strategies do not differ much: on average, only roughly 5% of the VNet requests are not immediately embedded. The fraction of delayed VNets increases with higher flexibilities F, e.g., to 75% (for $F = 1.001$) resp. to 95% (for $F = 1.01$). Using the ROOF strategy, the broker will benefit more since the already bought resources can be used to a greater extent.

Provider Perspective. The provider perspective is similar: the flexibility benefits that could be exploited by the broker automatically translate into an additional revenue for the cloud provider; we have a zero-sum game. Interestingly, these benefits also depend on the variance of the demand. Table 3.2 shows the price paid by the broker (and thus the income of the cloud provider), under different variances of the VNet duration, i.e., different α values in the Pareto distribution (note that larger α also increases the demand, cf Table 3.2 INSTANT). In general, we observe that a higher variance benefits the cloud provider: resources cannot be delayed efficiently. Interestingly, a higher variance does not necessarily lead to a higher price for the broker using the ROOF strategy however. This can be explained by the large discount in these cases.

3.2.4 Summary

This section studies the benefits and beneficiaries of specification flexibilities from Section 3.1 and introduces two most simple market models. In horizontal markets, we find that customers can often benefit from being

Pareto α	1.5	2	2.5	3
INSTANT	12109	11503	11183	10951
STAIRS	10264	9143	8493	7943
ROOF	1105	1291	1248	1190

Table 3.2: Mean provider income given VNets with different durations (a smaller *Pareto* α means a higher variance). Inter-arrival times $\lambda = 1.2$, discount $\delta = 1.2$, flexibility $F = 1.01$.

more flexible in some dimension (e.g., execution time) in the sense that the service is improved in the other dimension. Moreover, more flexibility on the customer side disproportionally benefits the cloud providers offering cheaper prices. Interestingly however, it is often not the cheapest provider that benefits the most from customer flexibilities. Finally, we also find that while the social welfare of both customers and providers is increased under heterogeneous requirements and flexibilities, not all customers can benefit in this setting; inflexible customers may even be worse off. Overall, more resources are needed in case of inflexible customers, opening new business opportunities for providers.

In vertical markets, we observe that the benefits from customer flexibilities typically accrue at the broker, which can bundle requests and exploit potential discounts on the provider side. However, depending on the broker strategy, a higher variance in the request demands helps the cloud providers to increase their revenues.

4 VNet Pricing and Buying Strategies

Now that we better understand the impact of specifications on the resource cost (Section 3.1) and on competitive cloud markets (Section 3.2), we turn to actual pricing and buying strategies for VNets. The objective of a provider offering these VNets is to maximize its revenue. As the degree of specification influences the resource cost it is likely that it also influences the pricing scheme. Thus, we study an actual pricing scheme for VNets in datacenters in Section 4.1, which includes pricing for bandwidth resources and is based on the principle that customers pay only for what they requested. We find that such pricing schemes can yield discounts and evaluate different strategies to exploit these discounts in Section 4.2. We assume the perspective of a broker, as he benefits the most from discounts on large and long running resource contracts.

4.1 VNet Pricing

Treating networking only as a second class citizen without resource and performance isolation guarantees and charging customer on a *per-VM-hour basis* is unfair to certain customers. Afterall, the traffic generated by cloud applications such as MapReduce and distributed databases is not negligible. Indeed, it has been shown that cloud applications suffer from resource interference on the network, in the sense that application performance can become unpredictable. Longer job execution times hence entail higher cost for the customers who are charged on a per-VM-hour basis [86]. This section presents a pricing scheme which is not only based on VMs but also takes networking resources into account. We also find indications that current embedding algorithms are not suited to cope with VNets with heterogeneous specifications and show the benefits of a simple embedding modification.

We stick to the Virtual Cluster abstraction introduced in Section 2.2, where a virtual cluster $VC(n, b)$ connects n virtual machines to a virtual switch at bandwidth b; essentially a hose model [86]. Datacenter networks are usually oversubscribed [52], meaning that there does not exist full bisection bandwidth within the network. While we neglected this fact in Section 3.2, it is now important to consider them in order to better understand the pricing strategies. This leads to two fundamental questions for this section: (1) How to embed virtual clusters on a given (oversubscribed) physical network? In order to make an optimal use of its resources and hence maximize the profit, a provider may want to multiplex as many virtual clusters as possible over the physical infrastructure (while fulfilling the requested virtual cluster specification). (2) How to efficiently price virtual clusters? While today's cloud pricing models typically focus on VM hours only, the pricing of virtual clusters becomes a 2-dimensional problem: a customer requesting more bandwidth should be priced accordingly. This section addresses these two questions.

Contribution

We present the first pricing scheme for virtual clusters, called DSP (*Demand-Specific Pricing*), which explicitly takes into account the different computation and bandwidth requirements. That is, unlike the dominant-resource pricing scheme *DRP* presented in the literature before [17], DSP is designed according to a specification-dependent, *pay-only-for-what-you-request policy*: While in *DRP*, the size n and the bandwidth b of a virtual cluster are strictly coupled, DSP allows customers to request virtual clusters $VC(n, b)$ of arbitrary and independent size n and bandwidth b, and be priced accordingly and in a fair manner. Moreover, DSP ensures desirable properties such as location independent pricing.

Together with this pricing scheme we present a new embedding algorithm called TETRIS[1] which is also specification-dependent in the sense that TETRIS accounts for differences in the node and link requirements of virtual clusters. Concretely, TETRIS is tailored to an online scenario where different virtual clusters $VC_1(n_1, b_1), VC_2(n_2, b_2), \ldots$ are requested over time, and collocates "opposites": computation-intensive but communication-extensive virtual clusters are mapped together with computation-extensive but communication-intensive virtual clusters, to maximize the number of simultaneously hosted virtual clusters over time. Given the online nature of the problem, this is a non-trivial task. We show that our algorithm outperforms previous algorithms, in the sense that a provider can host more virtual clusters, and hence increase its profit.

4.1.1 Background & Model

Most cloud providers today still offer virtual machine services only, charging their customers on a per-hour basis. However, we witness a trend towards more network oriented specifications (see also, e.g., *Amazon Placement Groups*, *Amazon EBS-Optimized Instances* or *Microsoft Azure ExpressRoute*), and especially the virtual cluster abstraction is becoming a popular model for datacenter applications. [18]

In [18], a first algorithm (henceforth called OKTOPUS) was proposed to embed virtual clusters in fat-tree datacenter topologies, and Ballani et al. [17] proposed a first pricing scheme for virtual clusters which give minimal bandwidth guarantees. Essentially, their scheme is based on *Dominant Resource Pricing*, short *DRP*. *DRP* provides different templates for the customers, with predefined sizes n and an associated amount of minimal guaranteed bandwidth b. While this model is attractive for its simplicity, also in the sense that the interface between the customer and provider may not have to be changed, the minimal bandwidth b is a function of n and cannot be chosen by the customer. As such, *DRP* is still 1-dimensional and does not leverage the full flexibility of the virtual cluster model, which is described by two *independent* parameters n and b.

Indeed, virtual cluster specifications are likely to come with different requirements [60] and can be heterogeneous [86]: a latency sensitive webservice can be very different in nature than, say, a delay-tolerant batch processing job or a network-hungry database synchronization application. One implication of the *DRP* scheme is that customers who know their virtual cluster demands might suffer from the inherent 1-dimensionality: in order to meet their application requirements, customers may be forced to upgrade to the next larger template, increasing both resources.

This section seeks to overcome this problem by allowing customers to specify their computation and communication requirements separately. Concretely, we allow the customer to specify three parameters independently: the number of VMs n, the computational type c of the virtual machines (e.g., small, medium, or large instances), as well as the bandwidth b. That is, we use a virtual cluster abstraction $VC(n, c, b)$, where all virtual machines are of the same computational type c, and are connected to a virtual switch at bandwidth b.

We use the following conventions in our notation. The computational type c is normalized in the sense that c describes the fraction of the capacity of a physical server. Similarly, we will normalize b to denote the requested fraction of the overall link capacity of a physical server. A central concept for our algorithm TETRIS (improving upon OKTOPUS) and pricing scheme DSP (improving upon *DRP*) is the *resource ratio* between the requested node and link resources, henceforth denoted by $\rho = c/b$.

[1]The name of the algorithm is due to its strategy to balance different resources equally, see also Figure 4.2 for an illustration.

In general, we will assume that requests arriving in an online fashion have to be immediately embedded or rejected by the provider. In order to successfully embed a virtual cluster, the provider has to fulfill all its specifications.

4.1.2 Pricing Scheme

The proposed specification-dependent pricing scheme DSP is based on a unit price for computation, henceforth denoted by π_c, as well as a unit price for communication (i.e., bandwidth), henceforth denoted by π_b. Ideally, for a virtual cluster request with a per-VM computational demand c and a per-VM bandwidth demand b, a customer should be charged according to the resource proportions, e.g.

$$\Pi_{\mathit{ideal}} = n \cdot (c \cdot \pi_c + b \cdot \pi_b)$$

However, compared to a dominant resource pricing scheme, this solution can result in a lower income at the provider, especially if requests are highly heterogeneous leading to a higher fragmentation. While this income loss could be compensated by increasing the unit prices π_c, π_b accordingly, one has to be careful not to punish customers with an ideal resource ratio $\rho = 1$, who would prefer providers offering *DRP* in this case. To solve this problem, in the following, we propose to add a small charge for customers with a resource ratio $\rho \neq 1$.

But let us first revisit the *DRP* scheme given our notation. In *DRP*, a customer who requests a virtual cluster $VC(n, c, b)$, with relatively lower resp. higher computation requirements compared to the communication requirement, is forced to upgrade the request to the next larger template for both resources. The corresponding formula for the *DRP* scheme is

$$\Pi_{\mathit{DRP}} = n \cdot [\max\{c, b\} \cdot (\pi_c + \pi_b)]$$

In order to bridge the difference between Π_{ideal} and Π_{DRP}, we propose the following *demand specific pricing* scheme DSP which introduces an extra fee for requests with an unbalanced resource ratio ρ. In this section, we will assume a linear dependency between the extra cost and ρ, although other dependencies (e.g., quadratic) could also be expressed in our model. This decision is based on the assumption that more skewed requests are more likely to generate fragmentation. In summary, DSP computes the virtual cluster price as follows:

$$\Pi_{\mathit{DSP}} = n \cdot (b \cdot \pi_b + c \cdot \pi_c) + \begin{cases} (c - b) \cdot \pi_b \cdot \lambda_b, & \text{if } c \geq b \\ (b - c) \cdot \pi_c \cdot \lambda_c, & \text{else} \end{cases}$$

where $\lambda_c, \lambda_b \geq 0$ are weighing factors. Note that this scheme can be seen as a generalization of the dominant resource pricing strategy: $\lambda_c, \lambda_b = 1$ implies that $\Pi_{\mathit{DSP}} = \Pi_{\mathit{DRP}}$. Lower weights result in savings for the customers, and $\lambda_c, \lambda_b = 0$ implies $\Pi_{\mathit{DSP}} = \Pi_{\mathit{ideal}}$. Given that the customers have a good understanding of their specific requirements in terms of computation and communication—a reasonable assumption as we argue—this pricing scheme leads to a higher provider profit, and in a competitive market, the extra income compared to *DRP* is shared with the customers.

Let us elaborate on the weighing factors λ_c and λ_b. In general, the factors should depend on the amount of embedded virtual requests as well as the current resource demand and supply. If the provider has a good estimation of the virtual clusters that will be requested, the values can be computed ahead of time; otherwise, the factors may be estimated over time, see also Section 4.1.4 for a discussion. Given a difference of Δ between the provider profit under *DRP* for upgraded and non-upgraded requests, the extra income generated by the lower resource consumption of the non-upgraded requests could be evenly distributed over requests requiring more of either one of the two resources. (Recall that the price of requests with $\rho = 1$ will not change.) The calculation for λ_b in a scenario with N virtual machines for which $c > b$ and with expected requirements $E(c)$ and $E(b)$, is given by

$$N \cdot (E[c] - E[b]) \cdot \pi_b \cdot (1 - \lambda_b) = \Delta/2$$

The factor λ_c can be computed similarly. In both cases, $c > b$ and $b > c$, a similar difference for $E[c]$ and $E[b]$ also leads to similar fees. This is fair, as one type of request only generates more profit because of the other one.

Also note that DSP keeps the desirable location independence of *DRP* [17]. However, unlike *DRP*, DSP is specification-dependent, i.e., a customer only has to pay for what he or she specified.

4.1.3 Embedding Algorithm

In order to fully exploit differences in the virtual cluster specification and to maximize the resource utilization (and hence provider profit), a new embedding algorithm has to be devised: the state-of-the-art virtual cluster embedding algorithm, Oktopus [18] (as well as its variants [111]), are based on an aggressive collocation strategy, which turns out to be suboptimal in settings where requests can have resource ratios $\rho \neq 1$.

In the following, we propose TETRIS, a virtual cluster embedding algorithm which leverages the virtual cluster specification details. The algorithm is tailored toward fat-tree datacenter topologies (cf Figure 4.1), the standard architecture today. In a nutshell, hosts (or equivalently: servers) which are connected to the same top of rack (ToR) switch, constitute a rack. Racks connected to the same aggregation switch form a pod. The fat-tree depicted in Figure 4.1 consists of two pods, containing three racks each; there are two hosts per rack.

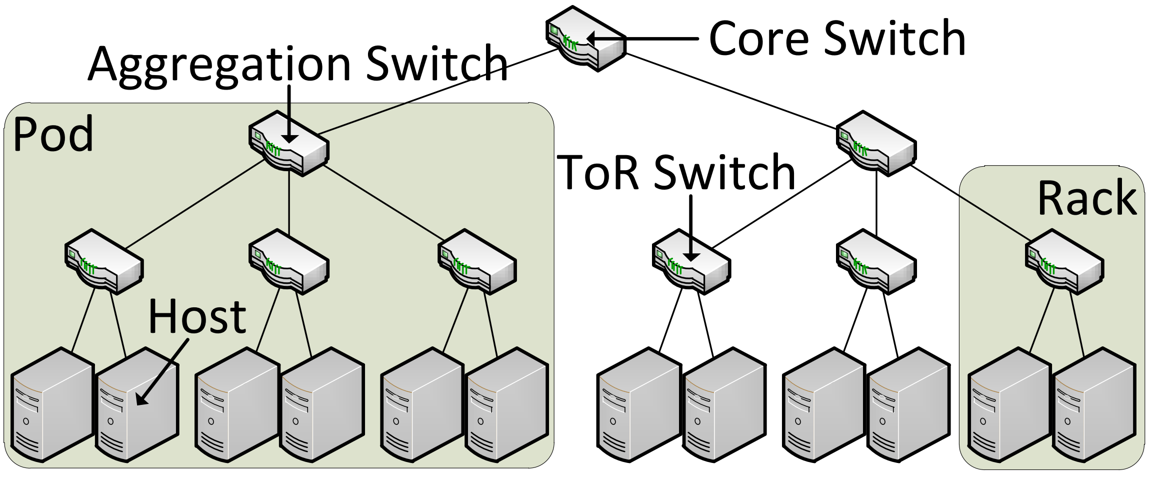

Figure 4.1: Fat-tree datacenter topology.

We will compare our embedding algorithm TETRIS to the OKTOPUS algorithm [18]. OKTOPUS is designed to embed arbitrary virtual clusters efficiently and is not limited to any templates. Hence it can also directly be used as an embedding algorithm for DSP, without any modifications. To find an embedding, OKTOPUS traverses the different hierarchical levels of the fat-tree. It tries to embed the virtual cluster on single hosts first, and if no solution is found, it continues on the rack level. This process is repeated until a solution is found or OKTOPUS failed to find a solution over the entire substrate. As a result of this approach, the resulting embeddings are dense and use low amount of bandwidth. The problem of such dense embeddings is that requests with a resource ratio $\rho \neq 1$ are collocated which wastes physical resources. Figure 4.2 (*left*) illustrates this point. For OKTOPUS, VC_1 is embedded on the right three hosts. The residual capacity in terms of VM slots on each host is $1/2$ of its total capacity, however, the bandwidth on the links is used up, rendering it unlikely that the remaining VM slots can be used in the future. On the other hand, the left three hosts, which host VC_2, only utilize 50% of their bandwidth, but have no remaining VM slots.

The core idea of our algorithm TETRIS is to distribute skewed VCs over multiple hosts, without increasing the bandwidth costs compared to OKTOPUS. Similar to OKTOPUS, TETRIS also traverses the hierarchical levels (short: ℓ) of the fat-tree, but instead of collocating as many *VMs* on a single host as possible, TETRIS distributes the *VMs* over physical machines (short: p) depending on the ratio of the residual resources per host. This is described in Algorithm 1.

In our example in Figure 4.2, using TETRIS results in a distributed embedding of both VCs. While all resources on the left three hosts are utilized, the right three hosts have spare capacities, both in terms of bandwidth and *VM* slots. Hence subsequent requests can more likely be accepted.

Note that the current design of TETRIS only considers the bandwidth on the access level. Hence, it can fail to find a feasible solution if the other layers of the fat-tree are oversubscribed. Therefore our current

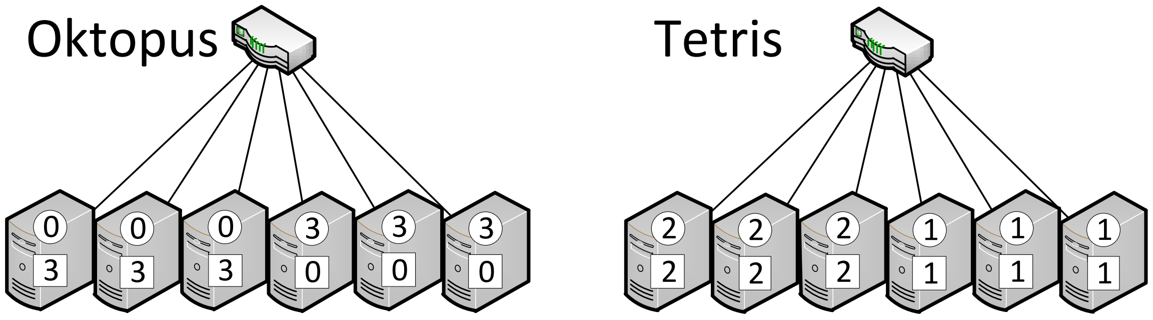

Figure 4.2: Embedding behavior of Oktopus and Tetris. Six hosts are connected to a switch. Two VCs are requested: $VC_1(9, 1/6, 2/6)$ and $VC_2(9, 2/6, 1/6)$. The numbers in the circles represent *VMs* of VC_1 which are mapped to the corresponding hosts, the numbers in the squares represent *VMs* of VC_2.

Algorithm 1: Tetris

```
Require: Fat-tree $T$, virtual cluster $VC$
1: for $\ell \in$ {host($T$), rack($T$), pod($T$), root($T$)} do
2:     for $v \in VC$ do
3:         find $p \in \ell$ with highest ratio of residual resources after embedding of $v$
4:     if $\forall\ v$ embedding found then
5:         return embedding
6: return $\perp$
```

implementation treats Tetris as an extension to Oktopus and falls back to regular Oktopus behavior if no solution was found.

4.1.4 Simulations

In order to study the benefits and limitations of DSP and Tetris in different settings, we implemented a discrete event simulator. As the pricing results also depend on the embedding algorithm, we study three combinations: we integrated *DRP* with Oktopus, DSP with Oktopus, and DSP with Tetris. To ensure a fair comparison, we use the same parameters and methodology as in [18] and [17].

Requests. The virtual cluster requests arrive according to a Poisson process with exponentially distributed durations, chosen to induce a system load of around 80%. By default, the mean size of a VC is $n = 49$, c and b are chosen uniformly from $\{1/8, 1/4, 1/2\}$. The templates (c, b) of *DRP* are $(1/8, 1/8), (1/4, 1/4), (1/2, 1/2)$ and the customer is bound to select the next larger template for requests with $\rho \neq 1$. For each parameter set, we request $80k$ virtual clusters. To avoid artifacts related to the initially empty datacenter, we start evaluating our metrics after $10k$ requests. The remaining values are omitted from the dataset.

Physical Setup. We model our datacenter as a three-layer fat-tree (Figure 4.1 illustrates a small fat-tree). 40 hosts form a rack, 40 racks form a pod. In total we have 10 pods. Given that each physical element has a capacity of 8 *VM* units, this leads to a total capacity of $128k$ small VMs. By default we assume that the links between the ToR switches and the aggregation switches are oversubscribed by a factor of 4, while the links between the aggregation switches and the core are not oversubscribed.

Metrics. Various works have used the acceptance ratio of an embedding algorithm in order to measure its performance. This metric however, is biased to prefer algorithms which accept a large number of small requests instead of fewer bigger ones. Therefore, we decided to evaluate the sum of the embedded virtual resources in both dimensions: bandwidth and VM slots. A request $VC(10, 1/4, 1/8)$ will have a *resource sum* of 10 for bandwidth and 20 for *VM* slots. Note that even though we will embed a $VC(10, 1/4, 1/4)$ for *DRP*, the bandwidth sum remains 10, as the customer does not benefit from the over-provisioning.

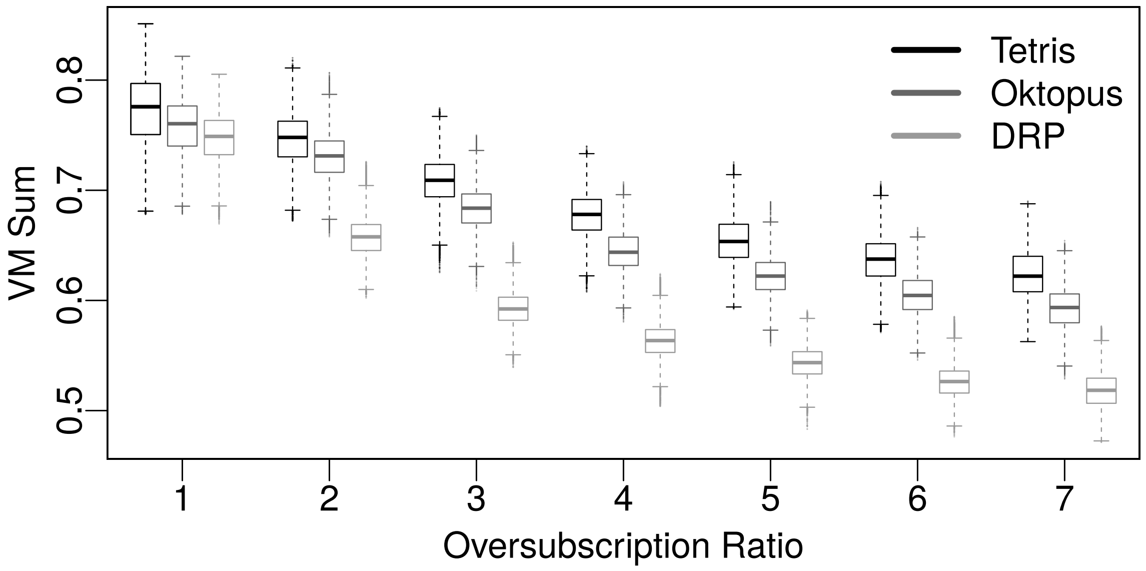

Figure 4.3: Embedded *VM* slots for TETRIS with DSP (*left*), OKTOPUS with DSP (*middle*) and OKTOPUS with *DRP* (*right*) as a function of the oversubscription.

Figure 4.3 shows the impact of the oversubscription factor on the embedded number of virtual machines. For all oversubscription ratios, we can observe that TETRIS with DSP outperforms the other combinations, while DSP is superior to *DRP* in combination with OKTOPUS. While the differences are small for an oversubscription factor of 1, we see an increase of the difference with the oversubscription factor; it diminishes after an oversubscription factor of 5, where the results become stable.

Naturally, we can only reap the full benefits of DSP and TETRIS in highly utilized datacenters, as shown in Figure 4.4: While first marginal effects can be observed at a load of 0.4, the benefits of DSP are visible starting at 0.5 and the benefits of TETRIS at 0.6. The effects increase until a load of 1. Given that highly utilized datacenters are a reality today and the key to provider benefits, these results are encouraging.

To analyze the benefit of the pricing model, we consider our default scenario: The mean resource sum for DSP with OKTOPUS is 15% higher than for *DRP* with OKTOPUS. This means that the amount of concurrent active guarantees is 15% larger. Using TETRIS with DSP yields another 5% improvement, resulting in a total benefit of 20%. Similar numbers apply for the bandwidth resource sum.

Assuming $\pi_b = \pi_c$ and a uniform distribution of accepted requests (i.e., an equal amount of embedded *VM*s of each (c, b) tuple), inserting the 20% as a Δ in Section 4.1.2 leads to savings of 27% for customers with skewed requests. The corresponding values of λ_c and λ_b are $1/6$, i.e., customers have to be charged for about 17% of the resource difference to *DRP*s templates in order to keep the revenue for the provider constant.

4.1.5 Summary

This section presents the first specification-dependent pricing scheme for virtual clusters, the standard abstraction of cloud applications today. Together with the pricing scheme, we also develop an extension TETRIS for the virtual cluster embedding algorithm OKTOPUS, which shows how much standard embedding algorithms can be enhanced given heterogeneous VNet requests. Our approach can easily be combined with interfaces, which translate high-level customer goals into virtual cluster requirements and compute resource combinations, such as [64].

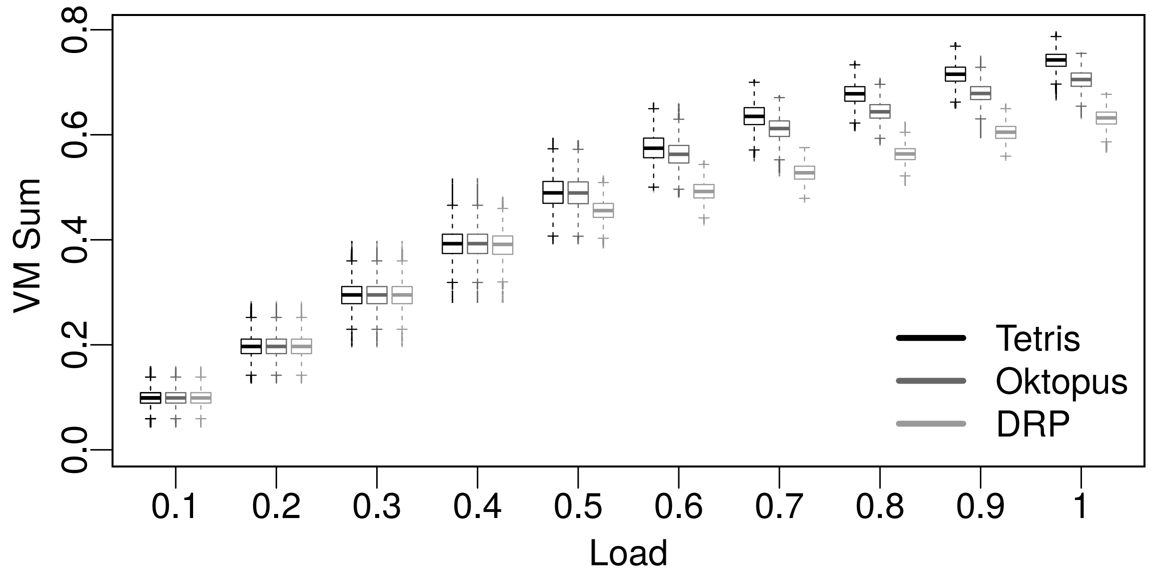

Figure 4.4: Embedded *VM* slots for TETRIS with DSP (*left*), OKTOPUS with DSP (*middle*) and OKTOPUS with *DRP* (*right*) as a function of datacenter load.

We find that the proposed specification-dependent pricing can increase the social welfare, leading to savings of up to 25% compared to *DRP*. In fact DSP enables customer to have guaranteed application performance while they only need to pay for the resources they use, including a small extra fee. Moreover, we find that TETRIS distributes virtual clusters with skewed resource requirements over several hosts and therefore embeds 5% more virtual resources than OKTOPUS.

4.2 VNet Buying

Section 4.1 shows that it is plausible, that a pricing scheme for VNets introduces discounts on resources. These discounts might even be larger if the customer chooses to buy more resources for a longer time period. This situation especially occurs if a broker is involved, who rents resources in larger scale.

This section studies the problem of a (cloud) *broker* who rents resource bundles from a (cloud) *provider*, to offer a certain service to its users (or to resell the resources). The broker is faced with the challenge that its resource demand is not known in advance (e.g., it depends on the popularity of its website). In order to ensure that its resource demand is satisfied at any time, and to avoid a costly over-provisioning of the service, additional resources must be bought in an *online* manner. The online resource allocation problem may further be complicated by the fact that the provider offers *discounts* for larger and longer resource contracts.

The goal of the broker is to come up with a smart resource renting strategy to satisfy its dynamic and unpredictable resource demand, while minimizing the overall costs of the bought resource bundles.

Contribution

This section shows that at the heart of efficient cloud resource allocation lies a fundamental algorithmic problem, and makes the following contributions. We first observe that the problem of renting a single resource over time can be seen as a *2-dimensional* variant of the well-known *online Parking Permit Problem* (PPP). While in the classic parking permit problem, only the *contract durations* need to be chosen, in the 2-dimensional variant PPP^2 introduced in this section, also the *resource rates* are subject to optimization.

We present the deterministic online algorithm $\mathrm{ON2D}$ whose performance is close to the one of a clairvoyant optimal offline algorithm which knows the entire resource demand in advance: given some simplifying assumptions (stated in Section 4.2.1), $\mathrm{ON2D}$ provably achieves a competitive ratio of $O(k)$, where k is the total number of available resource contracts; this is asymptotically optimal in the sense that there cannot exist any deterministic online algorithm with competitive ratio $o(k)$.

We also give a constructive proof that the offline variant of the PPP^2 problem can be solved in polynomial time, by presenting a dynamic programming algorithm $\mathrm{OFF2D}$ accordingly. To the best of our knowledge, $\mathrm{OFF2D}$ is also the first offline algorithm to efficiently solve PPP and PPP^2 for long enough request sequences. $\mathrm{OFF2D}$ is used as a subroutine in $\mathrm{ON2D}$.

Finally, we show that our algorithms and results also generalize to multi-resource scenarios, i.e., to higher-dimensional parking permit problems.

4.2.1 Model

We attend to the following setting (for an illustration, see Figure 4.5). We consider a broker with a dynamically changing resource demand. We model this resource demand as a sequence $\sigma = (\sigma_t)_t$, where σ_t refers to the resource demand at time t. We use $\widehat{\sigma}$ to denote $\max_t \sigma_t$. The broker is faced with the challenge that its future resource requirements are hard to predict, and may change in a worst-case manner over time: We are in the realm of *online algorithms and competitive analysis*.

In order to cover its resource demand, the broker buys resource contracts from a (cloud) provider. For ease of presentation, we will focus on a single resource scenario for most of this section; however, we will also show that our results can be extended to multi-resource scenarios. Concretely, we assume that the provider offers different resource contracts $C(r,d)$ of *resource rate* r and *duration* d. We will refer to the set of available contracts by $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$. Each contract type has a *price* $\pi(C) = \pi(r,d)$, which depends on its rate $r(C) = r$ and its duration $d(C) = d$. We will assume that resource contracts have a monotonically increasing rate-duration product $r \times d$, and will denote by C_i the i^{th} largest contract.

A specific contract instance of type C_i will be denoted by $C_i^{(j)}$, for some index j. Each instance $C_i^{(j)}$ of contract type $C_i(r,d)$ has a specific start time $t_i^{(j)}$, and we will sometimes refer to a contract instance by

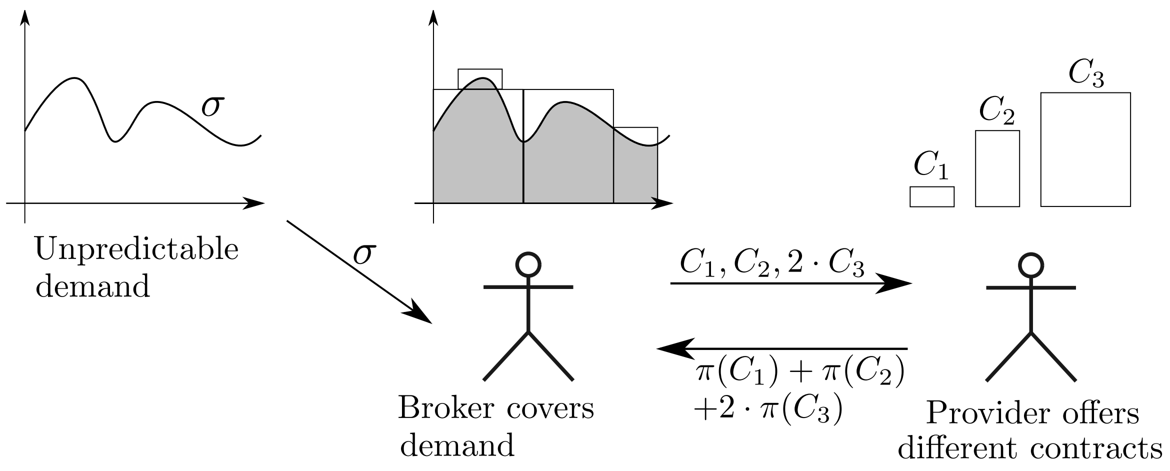

Figure 4.5: Overview of the model: A broker has to cover its resource demand σ by buying contracts $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ from the provider. Larger contracts C_i come with a price discount (price $\pi(C_i)$).

$C_i^{(j)}(t_i^{(j)}, r_i, d_i)$. The identifiers are needed since multiple contracts of the same type can be "stacked" in our model, but will be omitted if the contract is clear from the context.

We will make three simplifying assumptions:

A1 *Monotonic Prices:* Prices are monotonically increasing, i.e., larger contracts are more expensive than shorter contracts: $\pi(C_{i-1}) < \pi(C_i)$ since $r_{i-1} \times d_{i-1} < r_i \times d_i$ for any i.

A2 *Multiplicity:* The duration and resource rate of a contract of type $C_i(r_i, d_i)$ are perfect multiples of the duration and rate, respectively, of contract $C_{i-1}(r_{i-1}, d_{i-1})$. That is, we assume that $d_i = x \cdot d_{i-1}$ and $r_i = y \cdot r_{i-1}$ for fixed bases $x, y \geq 2$. Moreover, without loss of generality (*w.l.o.g.*), we will assume that the smallest contract has $d_1 = 1$ and $r_1 = 1$.

A3 *Intervals:* We assume that a contract of duration d can only be bought at time $t_0 + i \cdot d$, where $t_0 = 0$ is the start time.

Assumption A1 is natural: contracts which are dominated by larger, cheaper contracts may simply be ignored. Assumption A2 restricts the variety of available contracts the broker can choose from, and constitutes the main simplification made in this section. Assumption A3 mainly serves the ease of presentation: as we will prove in this section (and as it has already been shown for the classic Parking Permit Problem [85]), an offline or online algorithm limited by the interval model is at most a factor of two off the respective optimal solution in the general model.

Now, let $\textsc{On}$ be some online algorithm, let $\mathcal{C}_t(\textsc{On})$ denote the set of contracts bought according to $\textsc{On}$ at time t and let $\mathcal{C}_{\leq t}(\textsc{On})$ denote the set of contracts bought according to $\textsc{On}$ through time t. We will use the notation $\mathcal{C}_t^*(\textsc{On}) \subseteq \mathcal{C}_{\leq t}(\textsc{On})$ to describe the set of *active* contracts at time t: i.e., contracts $C_i(t_i, r_i, d_i)$ bought by $\textsc{On}$ with $t_i \leq t < t_i + d_i$. Likewise we denote the set of contracts bought by an optimal offline algorithm $\textsc{Off}$ to cover the demand prefix $\sigma_1, ..., \sigma_t$ until time t by $\mathcal{C}_{\leq t}(\textsc{Off})$.

Since a correct algorithm must ensure that there are always sufficient resources to cover the current demand, the invariant $\sum_{C(r,d) \in \mathcal{C}_t^*(\textsc{On})} r \geq \sigma_t$ must hold at any moment of time t. We will use the *one-lookahead model* [25] frequently considered in online literature, and allow an online algorithm to buy a contract at time t *before* serving the request σ_t; however, $\textsc{On}$ does not have any information at all about $\sigma_{t'}$ for $t' > t$.

The goal is to minimize the overall price $\pi_\sigma(\textsc{On}) = \sum_{C \in \mathcal{C}_{\leq t}(\textsc{On})} \pi(C)$. More specifically, we seek to be competitive against an optimal offline algorithm and want to minimize the *competitive ratio* ρ of $\textsc{On}$: We compare the price $\pi_\sigma(\textsc{On})$ of the online algorithm $\textsc{On}$ under the external (online) demand σ, to the price $\pi_\sigma(\textsc{Off})$ paid by an optimal offline algorithm $\textsc{Off}$, which knows the entire demand σ in advance.

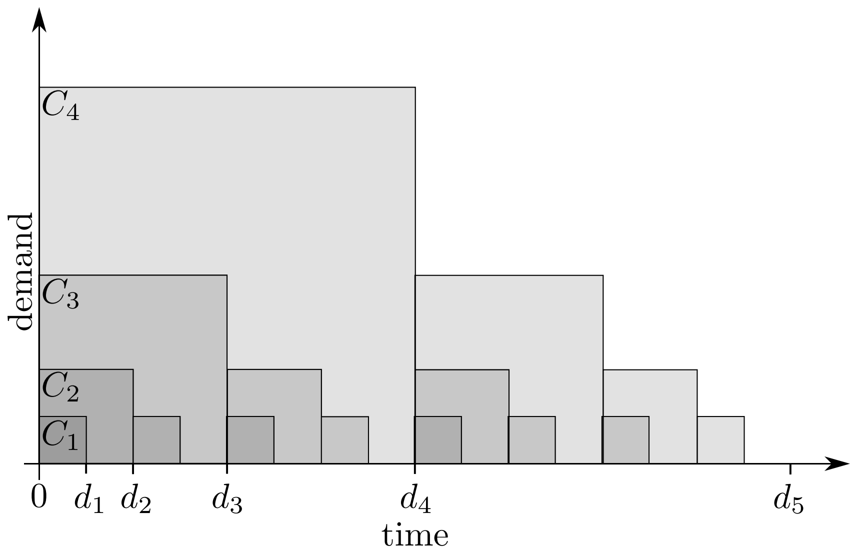

Figure 4.6: Worst-case example where $\sigma_t = 1 \quad \forall t$. While $\textsc{Off2D}$, at time d_5, buys a single contract C_5, $\textsc{On2D}$ is forced to buy all the depicted contracts, in addition to C_5. For instance, $\textsc{On2D}$ buys C_1 in every second time step.

Formally, we assume a worst-case perspective and want to minimize the (strict) competitive ratio ρ for any σ: $\rho = \max_\sigma \pi_\sigma(\textsc{On})/\pi_\sigma(\textsc{Off})$.

We are interested in long demand sequences σ; in particular, we will assume that the length of σ, $|\sigma|$, is at least as large as the largest single demand σ_t.

Our problem is a new variant of the classic Parking Permit Problem PPP [85], which we review quickly in the following. In PPP, a driver has to choose between k parking permits of different durations and costs in order to satisfy all of his/her parking needs while minimizing the total cost paid. More precisely, the driver has a sequence of days when he/she needs a parking space at a parking garage and there are k different parking permits, where each parking permit P_i allows the driver to use one parking space for d_i consecutive days at a cost of c_i. In the online version of the problem, the sequence of days when the driver will need a parking space is not known in advance.

4.2.2 Competitive Online Algorithm

This section presents the deterministic online algorithm $\textsc{On2D}$ for the PPP^2 problem. As a subroutine, in order to determine which contracts to buy at time t, $\textsc{On2D}$ uses an optimal offline algorithm $\textsc{Off2D}$ that computes optimal contracts for a prefix $\sigma_{\le t}$ of the demand through time t. In this section, we will treat $\textsc{Off2D}$ as a black box, but we will describe a polynomial-time construction later in Section 4.2.5.

In order to formally describe and analyze our algorithm, we propose a scheme that assigns bought contracts to the 2-dimensional *time-demand plane*. Our model requires that each point below the demand curve σ is covered by a contract, i.e., the mapping of contracts to demand points must be surjective.

We pursue the following strategy to assign contracts to the time-demand plane: at any time t, we order the set of active contracts by their duration, and stack the active contracts in such a way that longer contracts are embedded lower in the plane, i.e., the longest running contract $C_i(r_i, d_i)$ covers the demand from 1 to r_i, the next shorter contract $C_j(r_j, d_j)$ then covers the demand $r_i + 1$ to $r_i + r_j$, and so on. This guarantees a unique mapping of a demand point $p(time, demand)$ at time t to a contract C_i for the offline algorithm.

Our online algorithm $\textsc{On2D}$ (see Algorithm 2) is based on an oracle $\textsc{Off2D}$ computing optimal offline solutions for the demand *so far*. $\textsc{On2D}$ uses these solutions to purchase contracts at time t. Concretely, $\textsc{On2D}$ mimics the offline algorithm in an efficient way, in the sense that it only buys the optimal offline

```
Algorithm 2: Online Algorithm ON2D
Input: Demand prefix $\sigma_{\leq t} = \sigma_1, \sigma_2, ..., \sigma_t$; set of contracts $\mathcal{C}_{\leq t-1}(\text{On2D})$ bought by ON2D through time $t-1$
Output: Contracts to be bought at time $t$: $\mathcal{C}_t(\text{On2D})$
1: $\mathcal{C}_{\leq t}(\text{Off2D}) \leftarrow \text{Off2D}(\sigma_1, \sigma_2, ..., \sigma_t)$
2: for $C \in \mathcal{C}_{\leq t}(\text{Off2D})$ do
3:     if $\exists$ demand point $p$ covered by $C$ such that $p$ is not covered by $\mathcal{C}_{\leq t-1}(\text{On2D})$ then
4:         $\mathcal{C}_t(\text{On2D}).add(C)$
5: return $\mathcal{C}_t(\text{On2D})$
```

contracts covering time t if the corresponding demand is not already covered by contracts bought previously by ON2D: At each time t, ON2D compares the set of previously bought contracts $\mathcal{C}_{\leq t-1}(\text{On2D})$ with the set of contracts $\mathcal{C}_{\leq t}(\text{Off2D})$ that OFF2D would buy for an offline demand sequence $\sigma_1, ..., \sigma_t$; ON2D then only buys the contracts $C \in \mathcal{C}_{\leq t}(\text{Off2D})$ for the demand at time t that is not covered by $\mathcal{C}_{\leq t-1}(\text{On2D})$.

Example

In order to provide some intuition of the behavior of ON2D, as a case study, we consider the special scenario where contracts are perfect squares, i.e., $C_i = (2^{i-1}, 2^{i-1})$, and where the contract prices have a specific discount structure, namely $\pi(C_i) = 2 \cdot C_{i-1}$, with $\pi(C_1) = 1$. This price function ensures that OFF2D will buy at most one C_j contract before it is worthwhile to buy the next larger contract C_{j+1} for the given time interval.

Let us now study the maximal cumulative price $\Pi(C_i)$. It is easy to see that under the price function above, the demand sequence σ with a constant demand of one unit per time, maximizes $\Pi(C_i)$ for $C_i = (2^{i-1}, 2^{i-1})$ and $\pi(C_i) = 2 \cdot C_{i-1}$: higher demands imply missed opportunities to charge ON2D for smaller contracts, as already a demand of two at given time t renders it worthwhile to buy C_2, and a demand of four renders it worthwhile to buy C_3, etc.

With the given demand σ, OFF2D will end up buying each of the smaller contracts once before it buys the next larger contract. The cumulative price derived from σ according to this behavior is $\Pi(C_i) = \sum_{j=1}^{i-1} \Pi(C_j) + \pi(C_i)$. We prove this claim by induction over the contract types i. For the base case $i = 1$, $\Pi(C_i) = \pi(C_i)$ holds trivially. Assuming the induction hypothesis for i we have:

$$\begin{aligned}\Pi(C_{i+1}) &= \sum_{j=1}^{i} \Pi(C_j) + \pi(C_{i+1}) \\ &= \sum_{j=1}^{i-1} \Pi(C_j) + \Pi(C_i) + \pi(C_{i+1}) \\ &\overset{Hyp.}{=} \sum_{j=1}^{i-1} \Pi(C_j) + \sum_{j=1}^{i-1} \Pi(C_j) + \pi(C_i) + \pi(C_{i+1})\end{aligned}$$

Due to the induction hypothesis, the cost of a quarter of $2^{i+1} \times 2^{i+1}$ is maximized for $\sum_{j=1}^{i-1} \Pi(C_j) + \pi(C_i)$. In order to maximize the cost in the second quarter (at the bottom of the time-demand plane) OFF2D would need to buy $\sum_{j=1}^{i-1} \Pi(C_j)$ again, and instead of buying a second contract C_i, the pricing scheme requires the purchase of contract C_{i+1}. Therefore, buying the same contracts again (despite C_i) must lead to $\Pi(C_{i+1}) = \sum_{j=1}^{i} \Pi(C_j) + \pi(C_{i+1})$.

In summary, we have derived a worst-case sequence σ for the considered price function, for which ON2D is k-competitive.

Theorem 3. *For the special setting considered in our case study,* ON2D *is k-competitive.*

Proof. Consider the discussed worst-case sequence σ, where ON2D has to buy every contract (total cost $\Pi(C_i)$) while OFF2D can simply buy C_i at price $\pi(C_i)$. We can show that $\Pi(C_i) \leq i \cdot \pi(C_i)$ and hence

$\Pi(C_i) \le k \cdot \pi(C_i)$. According to the observed behavior of Off2D, every second contract bought by On2D is C_1 (2^{i-2} times), every fourth is C_2 (2^{i-3} times), etc., and finally On2D also buys C_i. See Figure 4.6 for an example. Thus,

$$\begin{aligned}\Pi(C_i) &= 2^{i-2} \cdot \pi(C_1) + 2^{i-3} \cdot \pi(C_2) + \dots + 1 \cdot \pi(C_i)\\ &= 2^{i-2} \cdot 2^0 + 2^{i-3} \cdot 2^1 + \dots + 2^{i-i} \cdot 2^{i-2} + 1 \cdot 2^{i-1}\\ &\le 2^{i-1} + 2^{i-1} + 2^{i-1} + \dots + 2^{i-1} + 2^{i-1}\\ &= i \cdot 2^{i-1} = i \cdot \pi(C_i)\end{aligned}$$

■

4.2.3 Analysis: Upper Bound

With these intuitions in mind, we now present a general analysis of On2D. First, we derive some simple properties of the contracts bought by the optimal offline algorithm Off2D over time.

Let us fix an arbitrary *demand point* p, i.e., a point below the σ-curve in the time-demand plane. We make the following claim: if p is covered by a certain contract C in $\mathcal{C}_{\le t}(\text{Off2D})$, p will never be covered by a smaller contract C' in $\mathcal{C}_{t'}(\text{Off2D})$ for any $t' > t$. In other words, when considering a longer offline demand sequence $\sigma_1, \dots, \sigma_{t'}$, Off2D will never buy a smaller contract than C to cover the demand point p. This property of "growing contracts" together with the assumption of disjoint intervals motivates the notion of *contract independence*, which we formalize in the lemma below:

Lemma 1 (Contract Independence). *Consider a demand point p_i covered by contract $C_i \in \mathcal{C}_{\le t}(\text{Off2D})$ and a demand point p_j covered by a distinct contract $C_j \in \mathcal{C}_{\le t}(\text{Off2D})$. Then there does not exist a contract $C \in \mathcal{C}_{t'}(\text{Off2D})$ for any $t' < t$ such that p_i, p_j are covered by C. We say that the two contracts C_i and C_j are* independent*.*

Independence between contracts is trivially ensured in our model. This allows us to introduce a simple characterization of the scenarios maximizing the competitive ratio.

Lemma 2. *The competitive ratio is maximized in a scenario where Off2D buys only one contract to satisfy the entire demand σ.*

Proof. By contradiction. Assume Off2D buys more than one contract, say C_i and C_j. Now assume that over time, On2D buys a set of (possibly smaller) contracts $C_{i'}, C_{i''}, \dots$ to cover the demand points of C_i and $C_{j'}, C_{j''}, \dots$ to cover the demand points of C_j. Thus, On2D pays $\pi(C_i)+\pi(C_{i'})+\dots$ and $\pi(C_j)+\pi(C_{j'})+\dots$ whereas Off2D pays $\pi(C_i)$ and $\pi(C_j)$; the resulting competitive ratio is $\rho_{C_i} = (\pi(C_i) + \pi(C_{i'}) + \dots)/\pi(C_i)$ for the C_i part and $\rho_{C_j} = (\pi(C_j) + \pi(C_{j'}) + \dots)/\pi(C_j)$ respectively. Since all contracts in Off2D are independent, the competitive ratio ρ of Off2D will be $\max\{\rho_{C_i}, \rho_{C_j}\}$, which would also be the case if the larger contract was the only one bought by Off2D. ■

We hence want to show that On2D will never buy too many small contracts to cover a demand for which Off2D would later only buy one contract. Concretely, let us fix any contract $C_i \in \mathcal{C}_{\le t}(\text{Off2D})$, and let us study the set of contracts S bought by On2D during the time interval $[0, t)$ which overlap with C_i in the time-demand plane. Recall that S will only contain distinct instances of the contracts (since On2D does not buy "repeated" contracts) and it will be contained in $\cup_{t'<t}\mathcal{C}_{t'}(\text{Off2D})$. By the interval and independence property, we know that contracts in S are all "inside" C_i, i.e., do not exceed its boundary in the plane. Accordingly, we can compute an upper bound on the maximum cumulative price spent on contracts in S by On2D while Off2D at time t only bought a single contract C_i at price π. In the following, let us refer to this cumulative price paid by On2D by $\Pi(C_i) = \sum_{C \in S} \pi(C)$.

Lemma 3. *The maximum cumulative price paid by On2D to cover a contract C_i, $\Pi(C_i)$, is less than or equal to $i \cdot \pi(C_i)$, for any $i \ge 0$.*

Proof. Consider a contract $C_i \in \mathcal{C}_{\leq t}(\textsc{Off2D})$ and S as defined above. Let ℓ be such that $\textsc{On2D}$ has bought ℓ contracts C_{i-1} to cover the area of C_i during time $[0,t)$, where $0 \leq \ell \leq \frac{\pi(C_i)}{\pi(C_{i-1})}$. For all other $C \in S$, we must have $C \in \{C_1, \dots, C_{i-2}\}$. Let $S' = \{C \in S,\ s.t.\ C \in \{C_1, \dots, C_{i-2}\}\}$. Hence we have $\sum_{C \in S'} \pi(C) \leq \pi(C_i) - \ell \cdot \pi(C_{i-1})$, since the area covered by all contracts in S is at most equal to the area covered by C_i, and given Assumption A1. We argue by induction on i.

Base case $i = 1$: If there is just one type of contract C_1, the online algorithm will buy the same contracts as the offline algorithm, and the claim holds trivially.

Inductive step $i > 1$: Assuming the induction hypothesis holds for all $j < i$, we have:

$$\begin{aligned}
\Pi(C_i) &= \ell \cdot \Pi(C_{i-1}) + \pi(C_i) + \sum_{C_j^{(p)} \in S'} \Pi(C_j^{(p)}) \\
&\leq \ell \cdot (i-1) \cdot \pi(C_{i-1}) + \pi(C_i) + \sum_{C_j^{(p)} \in S'} j \cdot \pi(C_j) \\
&\leq \ell \cdot (i-1) \cdot \pi(C_{i-1}) + \pi(C_i) + (i-2) \sum_{C_j^{(p)} \in S'} \pi(C_j) \\
&\leq \ell \cdot (i-1) \cdot \pi(C_{i-1}) + \pi(C_i) + \\
&\quad + \ (i-2) \left[\pi(C_i) - \ell \cdot \pi(C_{i-1})\right] \\
&= \ell \cdot \pi(C_{i-1}) + (i-1) \cdot \pi(C_i) \\
&\leq \frac{\pi(C_i)}{\pi(C_{i-1})} \cdot \pi(C_{i-1}) + (i-1) \cdot \pi(C_i) \\
&= \pi(C_i) + (i-1) \cdot \pi(C_i) = i \cdot \pi(C_i)
\end{aligned}$$

■

With these results, we can derive the competitive ratio. According to Lemma 3, for each contract $C_i^{(j)} \in \mathcal{C}_{\leq t}(\textsc{Off2D})$, the accumulated cost $\Pi(C_i^{(j)})$ is bounded by $i \cdot \pi(C_i)$. Therefore, summing up all the accumulated costs of each contract in $\mathcal{C}_{\leq t}(\textsc{Off2D})$, we get the total cost of $\textsc{On2D}$ at time t. Note that every contract bought by $\textsc{On2D}$ must be totally covered by contracts in $\mathcal{C}_t(\textsc{Off2D})$, since $\mathcal{C}_t(\textsc{Off2D})$ is an optimal solution for the entire demand sequence $\sigma_{\leq t}$ and the contract independence property holds. Since we have k different contracts and for each contract C_i in $\mathcal{C}_t(\textsc{Off2D})$, we have $\Pi(C_i) \leq i \cdot \pi(C_i) \leq k \cdot \pi(C_i)$, and:

Theorem 4. $\textsc{On2D}$ *is k-competitive, where k is the total number of contracts.*

As we will show in Section 4.2.4, this is almost optimal.

Finally, observe that restricting $\textsc{On2D}$ to Assumption A3 does not come at a large cost.

Theorem 5. *Let $\textsc{Alg}_1$ be an optimal offline algorithm for PPP^2, and let $\textsc{Alg}_2$ be an optimal offline algorithm for PPP^2 where we relax Assumption A3. Then $\pi(\textsc{Alg}_2) \leq \pi(\textsc{Alg}_1) \leq 2 \cdot \pi(\textsc{Alg}_2)$.*

Proof. Consider any contract $C_i^{(m)}(r_i, d_i)$ bought by an optimal offline algorithm for PPP^2 without Assumption A3. When time is divided into intervals of length d_i, $C_i^{(m)}$ will overlap in time with at most two contracts $C_i^{(j)}$ and $C_i^{(l)}$ of duration d_i. Therefore, we can modify the optimal solution output by $\textsc{Alg}_2$ by purchasing those two contracts instead of $C_i^{(m)}$, eventually transforming the optimal solution output by $\textsc{Alg}_2$ into a feasible solution for PPP^2 (under Assumption A3). Hence, we can guarantee that $\pi(\textsc{Alg}_2) \leq \pi(\textsc{Alg}_1) \leq 2 \cdot \pi(\textsc{Alg}_2)$. ■

Hence, since $\textsc{On2D}$ is k-competitive under Assumption A3 (Theorem 4), and since the optimal offline cost is at most a factor of two lower without the interval model (Theorem 4.2.3), we have:

Corollary 1. $\textsc{On2D}$ *is $2k$-competitive for the general PPP^2 problem without Assumption A3, where k is the number of contracts.*

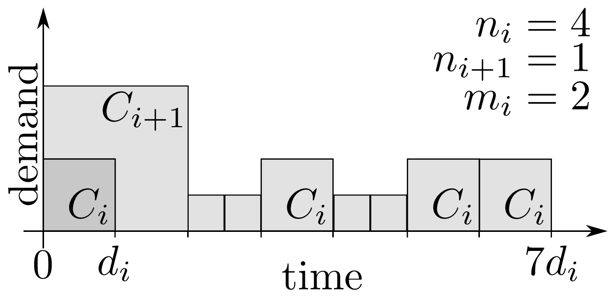

Figure 4.7: ON buys $n_i = 4$ contracts C_i and $n_{i+1} = 1$ contract C_{i+1} over seven intervals of length d_i. In two of these seven intervals ON buys several contracts smaller than C_i to cover the demand.

4.2.4 Analysis: Lower Bound

Theorem 4 is essentially the best we can hope for:

Theorem 6. *No deterministic online algorithm can achieve a competitive ratio less than $k/3$.*

The proof is the 2-dimensional analogon of the proof in [85]. We consider a scenario where the next larger available contract doubles in cost. With k being the number of different contracts, each contract is $2k$ times longer and has $2k$ times more rate, i.e., in our plane representation contracts are squares covering an area $(2k)^{2i}$.

$$\begin{aligned}\pi(C_i) &= 2^{i-1}\\ r_1 = 1; r_i &= 2k \cdot r_{i-1} = (2k)^{i-1}\\ d_1 = 1; d_i &= 2k \cdot d_{i-1} = (2k)^{i-1}\end{aligned}$$

In the following, let us focus on a simple demand which only assumes rates $\sigma_t \in \{0, 1\}$ for all t. We let the adversary schedule demand only when ON has no valid contract. For each interval $(2k)^i$ where the adversary asks for a 1-demand, ON can choose between three options (see also Figure 4.7):

1 Eventual purchase of contract C_i. Assume that this happens n_i times.

2 Eventual purchase of larger contracts $C_j, j > i$. Assume that this happens $\sum_{j>i}^{k} n_j$ times.

3 Never purchase contract C_i or any larger contracts. Assume this happens m_i times.

Therefore the sum of all contracts bought by ON is given by $\pi(\text{ON}) = \sum_{i=1}^{k} n_i \cdot \pi(C_i)$. Given an interval of length ℓ, we estimate the cost of OFF by less than buying multiples of only one kind of contract over the full interval, i.e., ℓ/d_i contracts *for any* i: $\pi(\text{OFF}) \le \pi(C_i)(m_i + \sum_{j\ge i}^{k} n_j)$. In order to derive the lower bound we first prove a minimum cost of any algorithm ON on intervals that start with a demand rate of 1.

Lemma 4. *Any* ON *must pay at least $\pi(C_i)$ on each interval of length d_i that starts with a demand rate of* 1.

Proof. By induction on the different intervals 2^{i-1}. For $i = 1$, each algorithm must at least buy a contract of type C_1 in order to cover that demand. Assume that for $i - 1$, it holds and now let us argue for i. If ON does not buy a contract of type C_i, we can divide the volume into $(2k)^2$ squares with side length d_{i-1} each, where $2k \cdot d_{i-1} = d_i$. We let each of these $2k$ intervals (at the bottom row) start with a demand of 1 which then cost at least $\pi(C_{i-1})$ due to the induction hypothesis. The total cost is at least $2k \cdot \pi(C_{i-1}) = k \cdot \pi(C_i)$ for every interval where ON does not buy a contract i and at least $\pi(C_i)$ otherwise. ■

Consider now an interval of length $(2k)^{i-1}$ where no contract of type i or higher was bought. We know from the induction that $\pi(\text{ON}) \ge m_i \cdot k \cdot \pi(C_i)$. We can derive the following lower bound:

Algorithm 3: Pre-computation of matrix M for d_k-length

```
Input: Demand sequence $\sigma_t, \ldots, \sigma_{t+d_k}$ (over interval $[t, t+d_k]$).
Output: Matrix $M$.
1: for $i = 1$ to $d_k$ do
2:     $M[i,i] \leftarrow \sigma_{t+i}$
3: for $i = 1$ to $d_k - 1$ do
4:     for $j = i+1$ to $d_k$ do
5:         $M[i,j] \leftarrow \max\{M[i,j-1], \sigma_{t+j}\}$
6: return $M$
```

$$
\begin{aligned}
k \cdot \pi(\textsc{Off}) &\le \sum_{i=0}^{k} \left[\pi(C_i)(m_i + \sum_{j \ge i}^{k} n_j) \right] && (4.1)\\
&\le \sum_{i=0}^{k} \left[n_i \sum_{j=1}^{i} \pi(C_i) + m_i \cdot \pi(C_i) \right] && (4.2)\\
&\le \sum_{i=0}^{k} \left[2n_i \cdot \pi(C_i) + m_i \cdot \pi(C_i) \right] && (4.3)\\
&\le 3 \cdot \pi(\textsc{On}) && (4.4)
\end{aligned}
$$

Inequality (4.1) is given by the cost estimation of $\textsc{Off}$ against any $\textsc{On}$ buying only one kind of contracts. Inequality (4.2) is a reorganization of the sum since $\pi(C_i)$ is multiplied by every $n_j, j \ge i$ which is also given after the reordering. Afterwards, we use the geometric sum on the cost of the contracts to derive Inequality (4.3). This leads to a lower bound of $k/3$ since $\pi(\textsc{On}) = \sum_{i=1}^{k} n_i \cdot \pi(C_i)$ and $\pi(\textsc{On}) \ge m_i \cdot k \cdot \pi(C_i)$.

4.2.5 Optimal Offline Algorithm

So far, we have treated the optimal offline algorithm on which $\textsc{On2D}$ relies as a black box. In the following, we show that offline solutions can indeed be computed in polynomial time, and present a corresponding dynamic programming algorithm $\textsc{Off2D}$.

The basic idea behind the offline algorithm $\textsc{Off2D}$ is that the optimal cost for any contract over a certain interval is obtained either by splitting the cost at some time, or by buying a long contract with a certain rate r. In the following, recall that d_k is the duration of the largest contract C_k.

$\textsc{Off2D}$ proceeds as follows: It splits time into intervals of length d_k and solves each of these interval separately using Algorithm 4. $\textsc{Off2D}$ relies on the following data structures: For each d_k-length time interval I, we precompute the maximum demand within any subinterval $[i,j]$ of I, and store this information in position $M[i,j]$ of a $d_k \times d_k$ matrix M (Algorithm 3). In particular the maximum requested demand $\hat{\sigma}$ in interval I is stored in $M[1,d_k]$. A $d_k \times d_k \times \hat{\sigma}$ matrix OPT is used to compute the optimal cost. The entry $\texttt{OPT}[i,j,\lambda]$ indicates the optimal cost of covering a demand rate of $M[i,j] - \lambda$ over the interval $[i,j]$ — i.e. λ indicates the amount of *covered demand* for $[i,j]$. Initially, all entries are set to 0.

Algorithm 3 pre-computes the matrix M over the d_k-length interval $[t, t+d_k]$, where $t = b \cdot d_k$, for integer $b \ge 0$. Lines 1-2 initialize the matrix and store the demand σ_{t+i} in entry $M[i,i]$. Lines 3-5 compute the maximum demand within any time interval $[t+i, t+j]$, $0 \le i \le j \le d_k$. The demand can be obtained by comparing the demand at time $t+j$ (i.e., σ_{t+j}) with the maximum demand between time $t+i$ and $t+j-1$, which has already been computed by our algorithm.

After obtaining the matrix M over interval $[t, t+d_k]$, we can compute the optimal solution for the PPP^2 problem over the same interval using Algorithm 4, as we show in Theorem 7:

Theorem 7. *Algorithm 4 computes an optimal offline solution for any given interval of length d_k in time $O(d_k^3 \cdot \widehat{\sigma})$, where $\widehat{\sigma}$ is the maximum demand over the interval.*

Algorithm 4: Offline Algorithm for d_k-length interval

```
Input: Precomputed matrix $M$ over interval $[t, t + d_k]$.
Output: Optimal total costs $\texttt{OPT}[i, j, \cdot]$ for all intervals within $[t, t + d_k]$.
 1: Initialize all entries in OPT to be $0$.
 2: Let $\hat{\sigma} = M[i, j]$.
 3: for $i = 1$ to $d_k$ do
 4:     for $\lambda = M[i, i] - 1$ to $0$ do
 5:         $\texttt{OPT}[i, i, \lambda] \leftarrow \min_{C(r,d) \in \mathcal{C}} \{\texttt{OPT}[i, i, \min\{\widehat{\sigma}, \lambda + r\}] + \pi(r, d)\}$
 6: for $\ell = 2$ to $d_k$ do
 7:     for $i = 1$ to $d_k - \ell + 1$ do
 8:         $j = i + \ell - 1$
 9:         for $\lambda = M[i, j] - 1$ to $0$ do
10:             $\texttt{OPT}[i, j, \lambda] \leftarrow \min_{i \le z < j} \{\texttt{OPT}[i, z, \min\{M[i, z], \lambda\}] + \texttt{OPT}[z + 1, j, \min\{M[z + 1, j], \lambda\}]\}$
11:             $\mathcal{C}' \leftarrow \{C^{(x)}(t^{(x)}, r, d) \in \mathcal{C} : t^{(x)} = b \cdot d$ for some positive integer $b$ and $t^{(x)} \le i < j \le t^{(x)} + d\}$
12:             if $\mathcal{C}'$ is not empty then
13:                 $\texttt{OPT}[i, j, \lambda] \leftarrow \min\{\texttt{OPT}[i, j, \lambda]; \min_{C(r,d) \in \mathcal{C}'} \texttt{OPT}[i, j, \min\{\hat{\sigma}, \lambda + r\}] + \pi(r, d)\}$
14: return $\texttt{OPT}[1, d_k, 0]$
```

Proof. We assume, for the sake of simplicity and without loss of generality, that $t = 0$ and the d_k-length interval we consider is $[0, d_k]$.

Correctness: By induction over the length of the subintervals $\ell = j - i + 1$ and the respective uncovered demand λ. Clearly, the claim is true for intervals $[i, i]$ ($\ell = 1$) (Lines 3-5): If $\lambda > 0$ we need at least one contract $C(r, d)$ to finish covering the demand at time i; the remaining demand at time i not covered by C must be covered optimally by other contracts, as previously computed in $\texttt{OPT}[i, i, \lambda + r]$.

Now consider a subinterval $[i, j]$ of length $\ell = j - i + 1 \geq 2$, where $1 \leq i \leq j \leq d_k$. This interval is either split into two non-overlapping subintervals of smaller length (Case I), or a long contract of length equal to or greater than ℓ that completely covers $[i, j]$ is bought, at a certain demand rate r, where $0 \leq r \leq M[i, j]$ (Case II). Given Assumption A2 and A3, for any instances of contracts $C_x^{(y)}$ and $C_p^{(q)}$, *either the duration of one contract is fully contained in the other*, or the *two contracts never overlap in time*: Hence, given that we consider all intervals $[i, j]$, including the ones that may correspond to actual instances of contracts, it is enough to consider only these two cases.

In Case I, we split the interval at time z such that the solution $\texttt{OPT}[i, z, \min\{M[i, z], \lambda\}] + \texttt{OPT}[z + 1, j, \min\{M[z + 1, j], \lambda\}]$ is minimized over all z between i and j (Line 10). Since the lengths of the two subintervals $z - i + 1$ and $j - z$ are both smaller than ℓ, $\texttt{OPT}[i, z, \lambda]$ and $\texttt{OPT}[z + 1, j, \lambda]$ already store the cost of optimal solutions for these subproblems, respectively, by the induction hypothesis. Hence $\texttt{OPT}[i, z, \lambda] + \texttt{OPT}[z + 1, j, \lambda]$ will yield the optimal solution for $\texttt{OPT}[i, j, \lambda]$ if Case I applies.

In Case II, we buy a long contract with rate r. First, we need to check which contracts with longer durations can cover $[i, j]$ fully, and store the candidate contracts in $\mathcal{C}'$. A candidate contract $C^{(x)}(t^{(x)}, r, d)$, where $t^{(x)} = b \cdot d$ according to Assumption A3, satisfies $t^{(x)} \leq i < j < t^{(x)} + d$. The algorithm picks the valid candidate contract that minimizes $\pi(r, d)$ plus the optimal cost of covering the largest remaining demand $M[i, j] - (\lambda + r)$ over $[i, j]$, which has been previously computed and stored in $\texttt{OPT}[i, j, \lambda + r]$ (Line 11).

By choosing the smaller value of Cases I and II, we obtain the optimal cost for subproblem $[i, j, \lambda]$ (Line 13).

Time Complexity: The total time complexity of OFF2D for the pre-computation part in Algorithm 3 is $O(d_k^2)$. The first part of Algorithm 4 in (Lines 3-5) takes $O(d_k \cdot k \cdot \widehat{\sigma})$ time, where $\widehat{\sigma}$ is the maximum demand for the whole time interval. The first two loops of the second part (Lines 6-7) take $O(d_k^2)$ time and the for-loop in Line 9 takes $O(\widehat{\sigma})$ time. The statement in Line 10 requires $O(d_k)$ time and Lines 11 and 13 take time $O(k)$ each. Therefore, the total time complexity is $O(d_k^3 \cdot \widehat{\sigma})$ for a subinterval with length d_k. ∎

Taking Theorem 7 into account for all intervals of length d_k in σ, and for a long enough demand sequence σ (i.e., such that $|\sigma| = \Omega(\widehat{\sigma})$, where $\widehat{\sigma}$ is the maximum demand over σ), we get the following corollary, which expresses the total running time of the offline algorithm:

Corollary 2. *Algorithm* $\textsc{Off2D}$ *runs in time* $O(|\sigma|^2 d_k^2)$.

Proof. By summing up the computation time of $\lceil|\sigma|/d_k\rceil$ subintervals of length d_k, we have an overall complexity of $O(|\sigma| \cdot d_k^2 \cdot \widehat{\sigma}) = O(|\sigma|^2 d_k^2)$, since $|\sigma| = \Omega(\widehat{\sigma})$. ■

4.2.6 Higher Dimensions

$\textsc{Off2D}$ and $\textsc{On2D}$ are designed for the two-dimensional version of the $\textsc{PPP}$ problem but they can also be extended towards a D-dimensional version of the problem, where each additional dimension (other than the time duration dimension) would indicate the rate at which one would buy a certain resource. Regarding $\textsc{Off2D}$ we need to do the following changes: For each additional dimension we need to extend the dimension of the optimal cost matrix OPT by one and add two additional loops in $\textsc{Off2D}$'s Algorithm 4. Furthermore we only need to add one additional dimension for the M matrix in Algorithm 3 which indicates the current demand dimension β, $M[i,j,\beta]$ and run the algorithm β times for the pre-computation. We show the Algorithms 5, 6 and illustrate those changes in $3D$.

Algorithm 5: Pre-computation of matrix M of length d_k for $\textsc{OffD}$-D

```
Input: Demand sequences $\sigma_t^{dim}, \dots, \sigma_{t+d_k}^{dim}$ (over interval $[t, t+d_k]$) with $dim \in \{1,2,...,D-1\}$.
Output: Matrix $M$.
1: for $i = 1$ to $d_k$ do
2:   for $dim = 1$ to $D-1$ do
3:     $M[i,i,dim] \leftarrow \sigma_{t+i}^{dim}$
4: for $i = 1$ to $d_k - 1$ do
5:   for $j = i+1$ to $d_k$ do
6:     for $dim = 1$ to $D-1$ do
7:       $M[i,j,dim] \leftarrow \max\{M[i,j-1,dim], \sigma_{t+j}^{dim}\}$
8: return $M$
```

Assume a scenario where a third dimension is added, e.g. computational and network resources over time. The contracts $C(r,r',d)$ then cover $r \times r' \times d$ cuboids. In order to adjust Algorithm 4, we add another loop after Line 4 which goes through the maximum values λ' of the additional demand (**for** $\lambda' = M[i,i,2]-1$ to 0 **do**) and change the statement in Line 5 to: $\texttt{OPT}[i,i,\lambda,\lambda'] \leftarrow \min_{C(r,r',d)\in\mathcal{C}'} \texttt{OPT}[i,i,\lambda+r,\lambda'+r'] + \pi(r,r',d)$. The same loop must also be added after Line 9 and the updates of the OPT matrix must be changed accordingly in Lines 10 and 13.

The runtime of the pre-computation in Algorithm 3 would be increased by a factor of D (i.e., by the dimension of the problem) and still be negligible regarding the overall runtime (assuming D is a constant). For Algorithm 4 the runtime would increase by a factor of $\Pi_{i\geq 2}\widehat{\sigma}^i$, where σ^i is the maximum demand for resource i, for $i \geq 1$, leading to an overall runtime of $O(d_k^3 \cdot \Pi_{i\geq 1}\widehat{\sigma}^i)$ for each interval d_k.

No changes are needed regarding $\textsc{On2D}$. It still mimics $\textsc{Off2D}$'s behavior and given the Assumptions A2 and A3, the contract independence still holds for higher dimensions. Hence, the proof for the competitive ratio of k still applies.

4.2.7 Simulations

We have conducted a small simulation study to complement our formal analysis. In this simulation, we consider k square contracts where $C_i(r_i, d_i)$ has rate and duration $r_i = d_i = 2^{i-1}$, for $1 \leq i \leq k$. The price π of a contract is a function of the rate-duration product $r_i \cdot d_i$, and we study a parameter x to vary the discount.

Algorithm 6: Offline Algorithm OFFD-D

Input: Precomputed matrix M over interval $[t, t + d_k, D]$.
Output: Optimal total costs $\texttt{OPT}[i, j, \cdot, \cdot, \ldots, \cdot]$ for all intervals within $[t, t + d_k]$.

```
1: Initialize all entries in OPT to be $0$ and $dim = D - 1$.
2: for $i = 1$ to $d_k$ do
3:   for $\lambda_1 = M[i, i, 1] - 1$ to $0$ do
4:     for $\lambda_2 = M[i, i, 2] - 1$ to $0$ do
5:       $\vdots$
6:       for $\lambda_{dim} = M[i, i, dim] - 1$ to $0$ do
7:         $\texttt{OPT}[i, i, \lambda_1, \lambda_2, \ldots, \lambda_{dim}] \leftarrow$
           $\min_{C(r_1, r_2, \ldots, r_{dim}, d) \in \mathcal{C}} \texttt{OPT}[i, i, \lambda_1 + r_1, \lambda_2 + r_2, \ldots, \lambda_{dim}] + \pi(r_1, r_2, \ldots, r_{dim}, d)$
8: for $\ell = 2$ to $d_k$ do
9:   for $i = 1$ to $d_k - \ell + 1$ do
10:    $j = i + \ell - 1$
11:    for $\lambda_1 = M[i, j, 1] - 1$ to $0$ do
12:      for $\lambda_2 = M[i, j, 2] - 1$ to $0$ do
13:        $\vdots$
14:        for $\lambda_{dim} = M[i, j, dim] - 1$ to $0$ do
15:          $\texttt{OPT}[i, j, \lambda_1, \lambda_2, \ldots, \lambda_{dim}] \leftarrow$
             $\min_{i \le z < j} \{\texttt{OPT}[i, z, \lambda_1, \lambda_2, \ldots, \lambda_{dim}] + \texttt{OPT}[z + 1, j, \lambda_1, \lambda_2, \ldots, \lambda_{dim}]\}$
16:          $\mathcal{C}' \leftarrow \{C^{(x)}(t^{(x)}, r_1, r_2, \ldots, r_{dim}, d) \in \mathcal{C} : t^{(x)} = b \cdot d$ for some positive integer $b$ and
             $t^{(x)} \le i < j \le t^{(x)} + d\}$.
17:          if $\mathcal{C}'$ is not empty then
18:            $\texttt{OPT}[i, j, \lambda_1, \lambda_2, \ldots, \lambda_{dim}] \leftarrow \min\{\texttt{OPT}[i, j, \lambda_1, \lambda_2, \ldots, \lambda_{dim}];$
               $\min_{C(r_1, r_2, \ldots, r_{dim}, d) \in \mathcal{C}'_1} \texttt{OPT}[i, j, \lambda_1 + r_1, \lambda_2 + r_2, \ldots, \lambda_{dim} + r_{dim}] + \pi(r_1, r_2, \ldots, r_{dim}, d)\}$
19: return $\texttt{OPT}[1, d_k, 0, 0, \ldots, 0]$
```

Concretely, we consider a scenario where a twice as large time-rate product is by factor $(1 + x)$ more expensive, i.e., $\pi(2 \cdot d \cdot r) = (1 + x) \cdot \pi(d \cdot r)$; we set $\pi(1) = 1$.

To generate the demand σ, we use a randomized scheme: non-zero demand requests arrive according to a Poisson distribution with parameter λ, i.e., the time between non-zero σ_t is exponentially distributed. For each non-zero request, we sample a demand value uniformly at random from the interval $[1, y]$.

Each simulation run represents 1000 time steps, and is repeated 10 times.

Impact of the request model. We first study how the competitive ratio depends on the demand arrival pattern. Figure 4.8 (*left*) plots the competitive ratio ρ as a function of the Poisson distribution parameter λ. The price model with $x = 0.5$ is used, and there are $k = 8$ contract types. First, we observe that the competitive ratio ρ is bounded by approx. 5, which is slightly lower than what we expect in the worst-case (cf Theorem 4). Another observation is that the competitive ratio decreases as λ increases. This can be explained by the fact that demand rates become sparser for increasing λ, and hence less contracts will be bought. Meanwhile, when the demand rates are sparse, the offline algorithm will have less chance to buy a larger contract. Put differently, the online algorithm will pay relatively more compared to the offline algorithm for small λ, as it purchases more small contracts.

Impact of the price model. Different price models also affect the purchasing behavior of our online algorithm. Figure 4.8 (*middle*) shows the competitive ratio ρ for different values x. (For this scenario, we set $y = 128$, $k = 8$ and $\lambda = 2$.) We see a tradeoff: for small x, until $x = 0.5$, the competitive ratio increases and then begins to decrease again. The general trend can be explained by the fact that for small x, it is worthwhile to buy larger contracts earlier, and it is hence impossible to charge ON2D much; for larger x, also an offline solution cannot profit from buying a large contract.

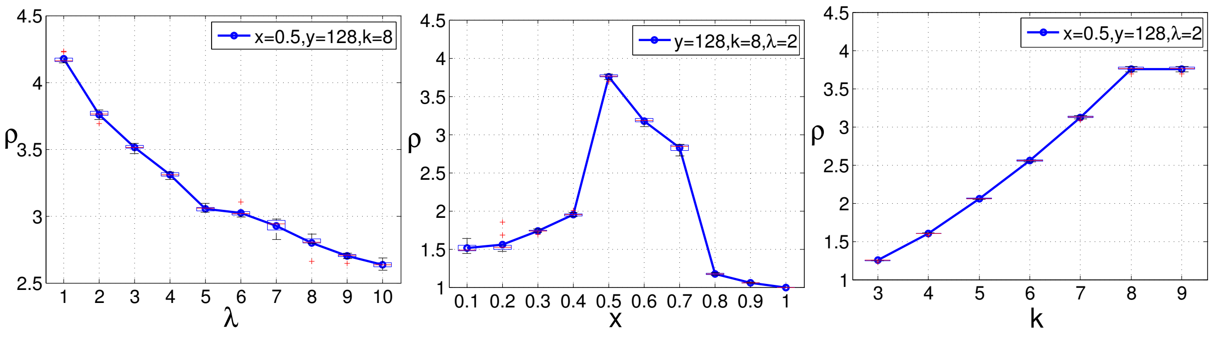

Figure 4.8: Simulation results. *Left:* Effect of request distribution (Poisson λ). *Middle:* Effect of discount x. *Right:* Effect of the number of contracts k.

Impact of the number of contracts. Finally, Figure 4.8(*right*) shows the competitive ratio as a function of the number of contracts k. (We fix $x = 0.5$, $y = 128$ and $\lambda = 2$ in this simulation.) The competitive ratio first increases as k increases but then stabilizes. This stabilization is due to the fact that when we have eight or more contracts ($k \geq 8$), the largest contract can cover the maximum rate. In the beginning, the ratio increases since the offline algorithm buys larger and larger contracts, and the online algorithm pays for many small contracts along the way.

4.2.8 Related Work

Cost reductions (due to economy-of-scale effects) are one of the main motivations behind today's trend to out-source infrastructure and software to the cloud. A large body of literature in the field focuses on resource allocation and scheduling problems. For a good overview, we refer the reader to the surveys [15, 32].

Compared to the algorithmic problems of the resource allocation and scheduling, the economical aspects are less well-understood. Kniesburges et al. [69] provide an overview of the cloud leasing research area. Different economical cloud models have been proposed and compared by various authors, e.g., by Armbrust et al. [79], Pal et al. [91], or Dash et al. [36]. Some of the studied pricing models have their origins in the context of ISP-customer relationships [105] and are also related to classic economic problems [51]. An interesting tradeoff between time and price has been studied in [59], from a scheduling complexity perspective. More generally, there are several interesting proposals for novel adaptive resource and spot market pricing schemes, e.g., [10]. Our model is motivated by architectures such as [26, 103] which allow for sub-renting and recursion.

This section assumed an online algorithm and competitive analysis perspective. Many online models, such as ski-rental problems [25], facility location problems [54], or buy-at-bulk [99] and rent-or-buy [71] problems, assume that once an item has been purchased, it remains indefinitely for no extra charge. The Parking Permit Problem [85] and the Bahncard Problem [47] are the archetype for online problems where purchases have time durations which expire regardless of whether the purchase is used or not.

The paper closest to ours is the Parking Permit Problem (PPP) paper by Meyerson [85]. Formally, PPP specifies a set of k different types of parking permits of a certain price π_i and duration d_i. In [85], Meyerson presents an asymptotically optimal deterministic online algorithm with competitive ratio $O(k)$ (together with a lower bound of $\Omega(k)$). The paper also discusses randomized algorithms and an application to Steiner forests. While we can build upon some of the techniques in [85], the rate dimension renders the problem different in nature, both from an online and an offline algorithm perspective. The Parking Permit Problem was also studied by Shouwei et al. [75]. The authors extend the classical problem with the introduction of deadlines and introduce the *Online Leasing with Deadlines* problem.

4.2.9 Summary

This section shows that at the heart of efficient cloud resource allocation lies a fundamental algorithmic problem, and we introduce the PPP^2 problem, a *2-dimensional* variant of the *online Parking Permit Problem* PPP. We present a deterministic online algorithm $\textsc{On2D}$ that provably achieves a competitive ratio of k, where k is the total number of available contracts; if we relax Assumption A3, the competitive ratio of our algorithm is $2k$. We also show that $\textsc{On2D}$ is almost optimal in the sense that no deterministic online algorithm for PPP^2 can achieve a competitive ratio lower than $k/3$. Finally, we prove that the offline version of PPP^2 can be solved in polynomial time.

We believe that our work opens interesting directions for future research.

1. *Optimality:* The obvious open question regards the gap between the upper bound k and the lower bound $k/3$ derived in this section.
2. *Relaxing the assumptions:* While the interval model only comes at the cost of a small additional approximation factor (a constant), it seems hard to remove the assumption entirely but still being able to compute optimal solutions in polynomial time: we conjecture that the problem is NP-hard. We believe that relaxing the multiplicity assumption (which is needed for the concept of contract independence in the upper bound proof of Section 4.2.3) is a more promising direction for future research.
3. *Randomized algorithms:* It will be interesting to study whether the randomized algorithms known from the classic parking permit problem can also be generalized to multiple dimensions.

Part II

Consistent Network Updates

5 Network Updates Overview

Network management is a complex task, especially at large scale [21]. Misconfiguration of even single routers can lead to, e.g., routing loops and, hence, in the worst case to the shutdown of the whole network. Indeed, it has been shown that most routing problems in, e.g., the context of BGP are caused by misconfigurations [44].

VNets will further increase the challenges for the network control plane, as many different forwarding patterns need to be handled in a fine-grained manner. Thus, in the second part of the thesis, we evaluate efficient ways of managing this increased diversity and dynamics induced by different VNet routing and forwarding patterns.

SDN (Software Defined Networking) promises to provide programmability for networks. Its enhanced control of the network makes it a "natural platform" for VNets [37]. In this chapter, we provide an overview of the SDN principle and the basics of network updates. While our approach to update networks is motivated by SDN, it is not limited to SDN.

5.1 Software Defined Networks

Configuration of network devices such as routers and switches is tedious work. Network devices often need to be configured individually and with vendor specific commands [68]. Even worse, these configurations need to be adapted to satisfy various high-level goals such as load balancing. Since this is an error-prone task, a simpler way to configure the network is needed. SDN was proposed to address these issues. It introduces programmability of networks and by that, a more high-level network management system.

The basic concept of SDN is to separate the *control plane* from the *data plane*, see Figure 5.1. Thus, the role of network devices is reduced to *forwarding devices*. The control part is shifted towards a typically abstracted and logically centralized *controller*. This controller is connected to the forwarding devices via a programming API as, e.g., *OpenFlow* [84]. This API enables the controller to communicate routing information towards the forwarding interfaces.

The controller has a complete view of the low-level details of the network. It provides the state of the network and topology abstractions towards programming APIs for *network applications*. Examples of network applications are basic tasks such as MAC learning or routing algorithms, but also more complex tasks such as load balancing or firewalls. The SDN concept has the potential to simplify network management. It provides the operator with a centralized, high-level view of the network. Thus, it should prevent him from potential errors in individually configuring the distributed low-level devices. In addition, applications can dynamically

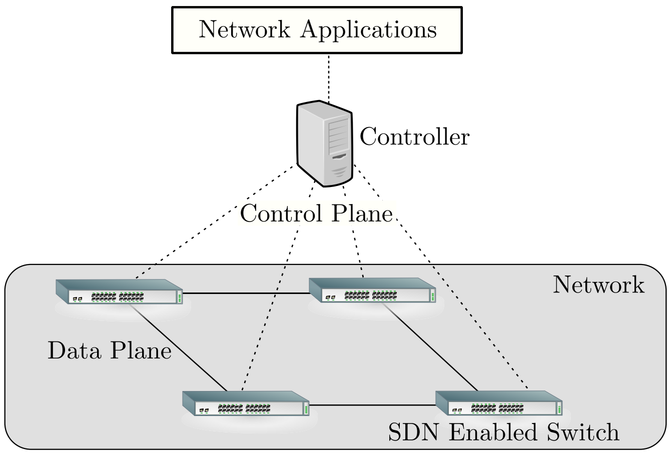

Figure 5.1: Overview of the SDN architecture, separating the control and the data plane, with a logically centralized controller on top.

react to changes within the network and reconfigure reasons. This avoids the manual reconfiguration of devices.

SDN enabled forwarding devices keep state in *flow tables*. These flow tables compose the forwarding information. Within the flow tables, each entry consists of a *match* and an *action*. Traditional networks forward packets based on their MAC (Layer 2) or IP address (Layer 3). Matches within SDN enabled forwarding devices can conceptually address any bit within the *packet header*. Thus, packets can be matched according to, e.g., a source and a destination IP address or even port numbers (Layer 4). To each match belongs an action that is performed once a packet is matched. An action forwards the packet to the outgoing ports or drops it. There is the additional option to forward packets to the controller if no match is found. The controller can decide where to forward these packets and if a new match-action pair needs to be installed. Throughout the thesis, we will refer to a match-action pair on a single forwarding device as *rule*.

These rules allow the controller to form *policies*. A policy is a sequence of rules on a set of forwarding devices. These entries can, for instance, define the path for every packet sent from a specific source IP address to a specific destination IP address.

Kreutz et al. [70] provide a comprehensive survey on the SDN architecture and applications. Initially introduced to facilitate experiments in campus networks, SDN also received a lot of attention from the industry. The *ONF - Open Networking Foundation* [7] was funded. Its goal is to promote SDN and to develop standards as, e.g., the *OpenFlow Standard*. Google presented their own implementation of an SDN WAN connecting their datacenters *B4* [63].

5.2 Network Updates

The main benefits of SDN are the programmability as well as the high-level network abstraction it offers to the operator. This concept depends on efficient algorithms installed at the controller to configure the network. Given the high-level network abstraction, a typical task is to change a policy. We refer to this task as a *network update*. An example of a network update is given in Figure 5.2. The *old policy* is shown on the left and the *new policy* is shown on the right.

Even though, the control plane is separated from the data plane and we have a logically centralized view of the network, performing a network update is not a trivial task. The reason is that a network should stay consistent during the update despite its distributed nature. Messages might be delayed, lost, and the update time of a single switch might also vary [65] which can cause problems even if the update is planned and executed well.

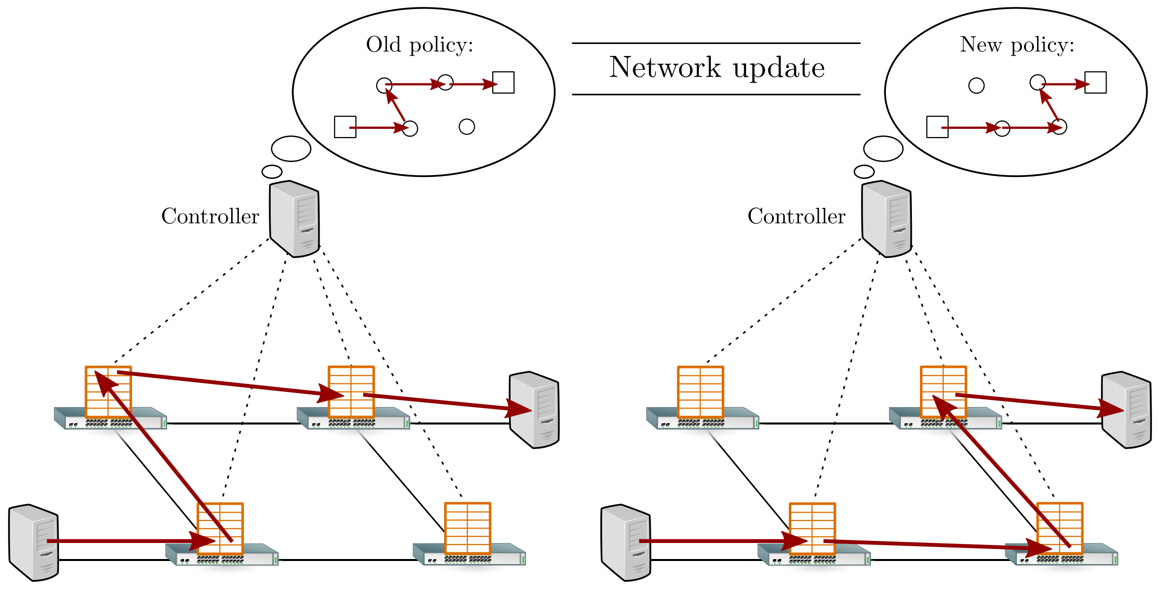

Figure 5.2: Network update. The rules for the old policy (*left*) are exchanged with the rules for the new policy (*right*).

To prevent network outages, e.g., caused by Layer 2 loops [1, 2], it is important to ensure network consistency during an update. Mahajan et al. [82] give an overview of several different network consistency properties, which we briefly discuss:

- *Eventual consistency* is always given as long as the old and the new policies are consistent.
- *Drop-freedom* ensures that no packet is dropped.
- *Loop-freedom* ensures that no packet loops.
- *Memory limit* ensures enough memory on the network devices to install the rules.
- *Bandwidth limit* ensures that no link is exposed to extensive traffic exceeding a limit.

Next, we discuss three different approaches to update networks, which all adhere to the three basic properties, eventual consistency, drop-freedom, and loop-freedom. Bandwidth limit is a property, which can be studied independently for these approaches. In terms of memory usage, the approaches differ.

Per Packet Consistency: Reitblatt et al. [95] propose a way to update a network which guarantees a per packet consistency. This per packet consistency is based on a *two-phase commit approach*. Within the two-phase commit, packets are not only forwarded according to the simple match-action rule but also according to a version number. This version number can be stored in an unused part of the packet header, e.g., the VLAN tag. It is used to differentiate between the old and the new policy. Thus, every packet is forwarded according to a single policy. It is tagged at the entry point and hence, forwarded according to its flag at every single switch. The controller can then send out the updates, which consist of the same match part from the match-action rule, but a different tag, to every switch involved in the new policy. As soon as every single switch installed the rule in its flow table, the controller sends an update to the entry point which triggers a retagging of packets from the old to the new tag. Hence, every packet is forwarded according to the new policy, which makes it possible to delete the rules from the old policy if they are not needed anymore. This two-phase commit approach has the benefit that the network stays consistent as long as the policies are consistent.

End-to-End Correctness: Ghorbani et al. [50] argue for an even stricter form of consistency. The authors propose a per path consistency. A per packet consistency might cause a wrong classification in the presence of middleboxes, e.g., firewalls. Even if a packet is consistently forwarded according to a single policy, a different policy for the traffic on the way back might lead to packets traversing different physical firewalls. Thus, the traffic might be blocked. These issues caused by the one-to-many mapping of a single logical entity (the firewall) to several different physical entities. Hence, they propose an end-to-end correctness, which extends the per packet consistency ideas with a partially ordered event set, which helps in the one-to-many mappings.

Overall, both per packet consistency and end-to-end correctness are heavyweight mechanisms. They are both based on the usage of additional tags in each packet. The usage of these tags is undesirable since there is limited room for tags in packet headers. The increased deployment of middleboxes furthermore increases the probability of a misconfiguration if headers are being rewritten. Another issue with the per packet consistency is that there are only very late first effects on the network. The controller needs to wait for every single switch to install the rule before packets can be retagged and hence, take a new path. The tagging also leads to situations where both, the old and the new rule need to be kept in the precious flow table memory during the update. This is a waste of memory if an update is not a more specific match than the old rule, but the same match with a different action.

Transient Consistency: A third approach is introduced by Mahajan et al. [82] and is based on a lightweight, dynamic form of updating a network. Instead of using tags, switches are updated directly. To avoid inconsistencies, not all of the switches are updated in parallel. The controller chooses a set of switches, which can be updated simultaneously in any given order without violating any consistency properties. Once the update is done for these switches, the controller sends out the update to the next set of switches. Thus, packets are forwarded according to a mix of the old and the new policy. This leads to earlier impacts on the network compared to the two-phase commit protocol. The set of updated nodes is chosen such that, given any ordering of switches of the set being updated, the network stays consistent. The consistency is guaranteed, even if the update on a single switch is delayed infinitely long. Note that in this case the complete policy update would not be completed until the issue is resolved.

Mahajan et al. focus on maximizing the number of updated switches per update set and propose a set of basic consistency properties. During this thesis, we further investigate on different update strategies and their complexity in the round-based model, while we also extend the consistency properties in order to cope with middleboxes in the network.

Bandwidth limitations have been studied independently of the update algorithms. Jin et al. [65] study congestion free network updates. The setting is a concurrent update of multiple policies. The challenge is to find congestion free updates on a per policy level. This differs from the discussed algorithms, which only update single policies. It is a complex problem, as there exist several possible schedules, some policies depend on others to be updated. Otherwise, the system might end up in a deadlock where no congestion free update exists. Thus, the authors construct the system *Dionysus*, which schedules policies according to a dependency graph. Depending on the speed of a single policy update, Dionysus schedules the updates of the next policies. This leads to a faster overall update time compared to a static scheduling.

6 Introducing a Round-Based Network Update Model

The network update problem as introduced in Section 5.2 has several different approaches to update a network. Most related work is based on the two-phase commit approach, which on the positive side, offers per packet consistency: Each packet is either forwarded according to the old or to the new policy. This provides a basic consistency guarantee as long as the policies are consistent. The downside of this approach is that it requires tags in a header field of the packets, rendering it a heavyweight mechanism. Mahajan et al. [82] proposed a *dynamic* update approach on a per switch granularity, providing *transient consistency*. Updates are scheduled into rounds such that the updates per round can be executed in any arbitrary order without violating any consistency properties. These round schedules are necessary, as measurement studies [65, 72] show that the switch update time vary by up to an order of magnitude.

Throughout the second part of the thesis, we evaluate the complexity and the efficiency of dynamic, round-based network updates. This chapter provides a detailed model of the dynamic network update problem. We introduce the formal definition of the problem and a reduction. Moreover, we show two different representations which simplify the reasoning about the problem. As the problem essentially is a graph algorithmic problem, we refer to forwarding devices as nodes and to links as edges. We define several classifications on the nodes and the edges and provide mechanisms to further simplify the representation during an update schedule.

The dynamic update problem is defined as follows: We are given a network and two policies π_1 (the *old policy*) and π_2 (the *new policy*). Both π_1 and π_2 are simple directed paths. Initially, packets are forwarded (using the *old rules*, henceforth also called *old edges*) along π_1, and eventually they should be forwarded according to the new rules of π_2. Packets should never be delayed or dropped at a node: whenever a packet arrives at a node, a matching forwarding rule should be present. Note that throughout this work we only consider switches, routers or middleboxes as relevant parts of the network for the update problem and call them nodes.

Model and Reduction: Without loss of generality, we assume that π_1 and π_2 lead from a source s to a destination d. Since nodes appearing only in one or none of the two paths are trivially updatable, we focus on the network G induced by the nodes V which are part of *both* policies π_1 *and* π_2, i.e., $V = \{v : v \in \pi_1 \wedge v \in \pi_2\}$. Thus, we can represent the policies as $\pi_1 = (s = v_0, v_2, \dots, v_{\ell-1} = d)$ and $\pi_2 = (s = v_1, \pi(v_2), \dots, \pi(v_{\ell-2}), v_{\ell-1} = d)$, for some permutation $\pi : V \setminus \{s,d\} \to V \setminus \{s,d\}$ and some number ℓ. In fact, we can represent policies in an even more compact way: we are actually only concerned about the nodes $U \subseteq V$ which need to be updated. Let, for each node $v \in V$, *out*$_1(v)$ (resp. *in*$_1(v)$) denote the successor (resp. predecessor) according to policy π_1, and *out*$_2(v)$ (resp. *in*$_2(v)$) denote the successor (resp. predecessor) according to policy π_2. We define s to

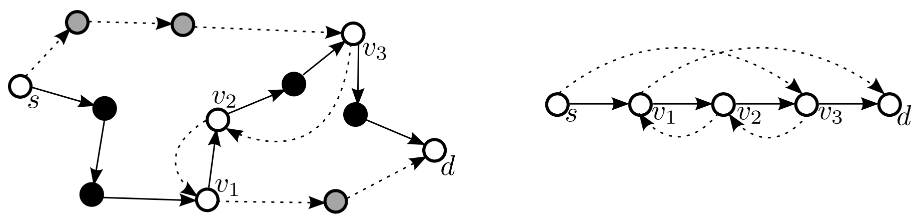

Figure 6.1: Overview of model and reduction. The network on the *left* is reduced to the line representation on the *right*. The solid lines show the old policy π_1 and the dashed lines show the new policy π_2. Nodes shown in white are the only ones which are part on both paths, and hence relevant for the problem.

be the first node (say, on π_1) with $out_1(v) \neq out_2(v)$, and d to be the last node with $in_1(v) \neq in_2(v)$. We are interested in the set of to-be-updated nodes $U = \{v \in V : out_1(v) \neq out_2(v)\}$, and define $n = |U|$. Given this reduction, in the following, we assume that V only consists of interesting nodes ($U = V$).

Line representation: Fig. 6.1 illustrates our model: We are given two policies (the old rules of π_1 are *solid*, the new ones of π_2 are *dashed*), see Fig. 6.1 (*left*). We focus on the updateable nodes, which are shared by the two policies. Thus, in our example, the update problem can be reduced to the 5-node chain graph in Fig. 6.1 (*right*). Throughout this work, we stick to this representation, and indicate the old policy π_1 using *solid lines*, and the new policy π_2 using *dashed lines*. Moreover, we depict the initial network configuration (before the update) such that the old policy goes from left to right.

Node Merging and Tree Representation: During the thesis, we make use of two additional concepts: *node merging* and *tree representation*. When updating a node v, we can merge it with the node $out_2(v)$ it pointed to with its dashed edge. This can safely be done after each round, due to the irrelevance of already updated nodes (they will simply forward packets to the next node, without influencing the remaining problem at hand). As we will see, while the initial network configuration consists of two paths, in later rounds, the already updated solid edges may no longer form a line from left to right, but rather an arbitrary directed tree, with tree edges directed towards the destination d; due to the node merging, the in-degree (from the solid edges) may also increase while the out-degree and in-degree from the dashed edges remains one. Moreover, note that while the destination d will always be the root of the tree, the source s does not necessarily have to be at the leaf all the time (due to merging).

Node Ordering: Nodes on the same branch (or on the line) can be ordered. We say that $u < v$, if u and v are on the same branch and v is closer to the destination d. The distance is defined as the path length (according to the currently installed rules—the solid lines) from a node to the destination. Two nodes which are not on the same branch cannot be ordered.

Classifying Edges: In the following we call an edge (u, v) of the new policy π_2 *forward*, if $u < v$, resp. *backward*, if $v < u$. This classification only works if u, v are on the same branch. Edges in-between two different branches of the tree are called *horizontal* edges. Note that this classification may change during update rounds. A backward edge (u, v), for instance, can change into a forward edge, if the edge (v, w) is updated, with $u < w$. The node merging from v and w then changes the direction of the edge. It is also convenient to name nodes after their outgoing dashed edges (e.g., *forward* or *backward*); similarly, it is sometimes convenient to say that we *update an edge* when we update the corresponding node. In addition, we treat the terms *edge* and *rule*, as synonyms in this work.

Figure 6.2 shows an example for node merging and tree representation. The left part shows a standard line representation of the scenario. The right side of the figure depicts the situation after an update of s. Node merging allows us to merge s and $out_2(s) = v_3$ which leads to a tree representation. The edge between s, v_3

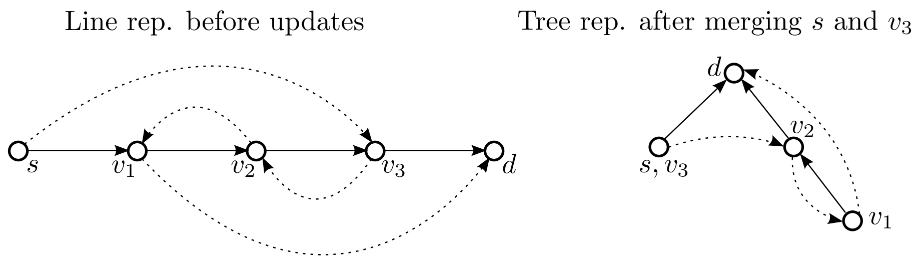

Figure 6.2: Example for node merging and tree representation. After updating s it can be merged with $out_2(s) = v_3$. This would lead to tree representation of the scenario.

and v_2 is a horizontal edge. The edge (v_1, d) keeps its forward orientation and (v_2, v_1) keeps its backward orientation. An update of node v_1 instead of node s would have changed the orientation of the edge (v_2, v_1). The nodes v_1 and d would have been merged, which leads to a changed classification of (v_2, v_1) from backward to forward. The edge (v_3, v_2) would have kept its backward classification.

7 Loop-free Network Updates

Exploiting the benefits in a logically centralized network management is non-trivial. A fundamental problem regards the consistent implementation of *policy updates*: To update a policy π_1 to a policy π_2, the controller needs to communicate the new forwarding rules to all nodes. However, as both the transmission as well as the installation of rules take time and are subject to variance [65], inconsistencies can be introduced during the update: For example, the same packet may still be forwarded according to the old rules (of π_1) at some nodes while it is forwarded already according to the new rules (of π_2) at others. The resulting *actual* routes may transiently violate basic consistency properties such as *loop-freedom* [82].

This chapter specifically tackles the challenge of *loop-free* network updates. Loop freedom is an important property within a network, especially on Layer 2, as there do not exist mechanisms to detect these loops. Prominent examples where such loops caused complete outages are, e.g., GitHub [2] and Amazon [1]. In both cases the outages occurred on planned operations within the network and not due to hardware failures, which shows the importance of consistent network updates.

As discussed in Chapter 6, we want to ensure these loop-free network updates, without tagging. Hence, communicating updates to nodes dynamically in a staged manner: The controller first updates only a *safe subset* of nodes $V_1 \subseteq V$. After these nodes asynchronously installed the new rules, they send an acknowledgement to the controller, which then schedules the next subset $V_2 \subseteq V$ of nodes to update, until the final subset V_k completes the policy update. This protocol does not require packet tagging, and, as has been argued in [82], also has the advantage that some of the edges of π_2 become available earlier to packets: there is no need to wait for the full installation of π_2.

This chapter studies *fast* loop-free network updates, i.e., updates which require a minimal number of controller interactions while providing transient consistency guarantees. We consider a model where network policies can follow arbitrary paths and are not necessarily destination-based (arguably a key benefit of SDN [45]). We ask: *How many communication rounds k are needed to update a network in a (transiently) loop-free manner?*

Contribution

We show that just aiming to "greedily" update a *maximum* number of nodes in each round (as proposed in previous work [82], however, for a different model) may result in $\Omega(n)$-round schedules in instances which actually can be solved in $O(1)$ rounds; even worse, a *single* greedy round may inherently delay the schedule by a factor of $\Omega(n)$ more rounds. In addition, we give a formal NP-hardness proof for maximizing the number of updates and, on the positive side, identify a class of network update problems that allow for optimal or almost optimal polynomial-time algorithms We show that focussing on minimizing the number of rounds is difficult

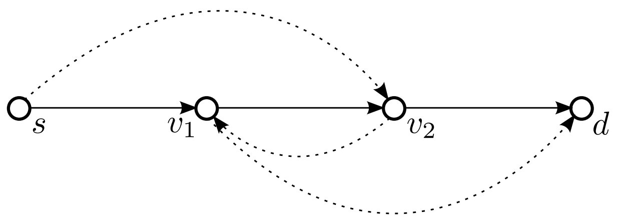

Figure 7.1: Example of a loop-free dynamic network update. Nodes s, v_1 will be updated in a first round, before v_2 can be updated in the second round. Old path: solid line; new path: dashed line.

as well. In particular, we show that while deciding whether a k-round schedule exists is trivial for $k = 2$, it is already NP-complete for $k = 3$. Moreover, we show that there exist problem instances which require $\Omega(n)$ rounds, where n is the network size.

Given these negative results, we propose an attractive alternative to the utterly strict loop-free requirement: *relaxed loop-freedom*. Relaxed loop-freedom is motivated by the observation that loops are only really problematic if they occur on the (changing) path between source and destination: topological loops in other parts of the network will never receive any new packets. We argue that relaxed loop-freedom not only expresses better the actually desired consistency in practice, but we also show that it comes with interesting benefits: We prove that $O(\log n)$-round relaxed loop-free schedules always exist, and can also be computed efficiently, and we present an elegant algorithm accordingly.

7.1 Loop-Freedom

Before we evaluate different update strategies, we formally introduce (strong) loop-freedom. We also provide a practically motivated alternative definition: relaxed loop-freedom.

7.1.1 Strong Loop-Freedom

We want to find a *schedule* $U_1, U_2, \dots, U_k$ with minimum k, i.e., a sequence of subsets $U_t \subseteq U$ where the subsets form a partition of U (i.e., $U = U_1 \uplus U_2 \uplus \dots \uplus U_k$), with the property that for any round t, given that the updates $U_{t'}$ for $t' < t$ have been made, all updates U_t can be performed "asynchronously", that is, in an arbitrary order without violating loop-freedom. That is, consistent paths will be maintained for any subset of updated nodes, independently of how long individual updates may take.

More formally, let $U_{<t} = \bigcup_{i=1,\dots,t-1} U_i$ denote the set of nodes which have already been updated before round t, and let $U_{\le t}$, $U_{>t}$ etc. be defined analogously. Since updates during round t occur asynchronously, an arbitrary subset of nodes $X \subseteq U_t$ may already have been updated while the nodes $\overline{X} = U_t \setminus X$ still use the old rules, resulting in a temporary forwarding graph $G_t(U, X, E_t)$ over nodes U, where $E_t = \mathit{out}_1(U_{>t} \cup \overline{X}) \cup \mathit{out}_2(U_{<t} \cup X)$. We require that the update schedule $U_1, U_2, \dots, U_k$ fulfills the property that for all t and for any $X \subseteq U_t$, $G_t(U, X, E_t)$ is loop-free.

Later in this work, we will sometimes refer to this definition of loop-freedom as the *Strong Loop-Freedom* (SLF), to distinguish it from *Relaxed Loop-Freedom* (RLF). By default, throughout this work, the term loop-freedom without additional qualifier will refer to the strong variant.

Update Example. Fig. 7.1 shows a simple network update. The solid line shows the old policy π_1 from s to d and the dashed path shows the new policy π_2. How can we update the policy π_1 to π_2? A simple solution is to update all nodes *concurrently*. However, as the controller needs to send these commands over the asynchronous network, they may not arrive simultaneously at the switches, which can result in inconsistent states. For example, if v_2 is updated before v_1 and s are updated, a temporary forwarding path may emerge

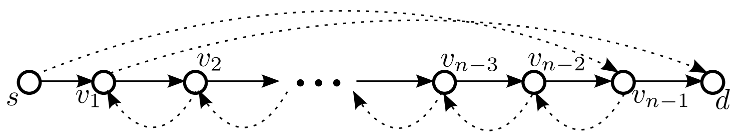

Figure 7.2: RLF vs SLF: An SLF schedule needs to update backward edges one by one from left to right, requiring $\Omega(n)$ rounds; for RLF, an $O(1)$-round schedule exists.

which violates loop-freedom: packets originating at s will be sent to v_2 and from there in a loop between v_1 and v_2.

This illustrates the challenge of updating the SDN, which is an inherently asynchronous and distributed system. One solution to overcome this problem would be to perform the update in *two (communication) rounds*: in the first round, only s and v_1 ($U_1 = \{s, v_1\}$) are updated, and in a second round, once these updates have been performed and acknowledged, the controller also updates v_2 ($U_2 = \{v_2\}$).

7.1.2 Relaxed Loop-Freedom

In this section, we also propose a weaker notion of loop-freedom: *Relaxed Loop-Freedom* (RLF). Relaxed loop-freedom is motivated by the practical observation that transient loops are not very harmful if they do not occur between the source s and the destination d. If relaxed loop-freedom is preserved, only a constant number of packets can loop: we will never push new packets into a loop "at line rate". In other words, even if switches acknowledge new updates late (or never), new packets will not enter loops. Concretely, and similar to the definition of SLF, we require the update schedule to fulfill the property that for all rounds t and for any subset X, the temporary forwarding graph $G_t(U, X, E'_t)$ is loop-free. The difference is that we only care about the subset E'_t of E_t consisting of edges *reachable from the source* s.

RLF Example. To highlight the difference between SLF and RLF, Fig. 7.2 presents an example where a relaxed 3-round loop-free update schedule exists: in round 1 all forward edges are updated, in round 2 all backward edges except for the last one (v_{n-1}, v_{n-2}) are updated, and in round 3, the last backward edge is updated. In contrast, a strong loop-free schedule needs to go through the backward edges one by one, $v_2, v_3, \cdots, v_{n-1}$: updating v_i before v_{i-1} results in a loop. Thus, $n-2$ (as d does not need to be updated) rounds are required in this case: a factor $\Omega(n)$ more than RLF. This is worst possible.

7.2 It is bad being greedy

The speed of a network update is measured in terms of communication rounds in this work: the number of times a controller needs to send updates to a subset of network elements. As we show in the following, previous objectives [82] which greedily maximize the number of updated links in a *single* round may unnecessarily delay the policy update. Furthermore we show that computing a maximum set of updatable nodes is NP-hard for both SLF and RLF.

7.2.1 Greedy Updates Delay

Updating a network according to the objective of maximizing the number of nodes per round can delay the overall update policy in terms of number of rounds and hence, arguable overall update time. We will give two worst case scenarios: one for each definition of loop-freedom (SLF and RLF).

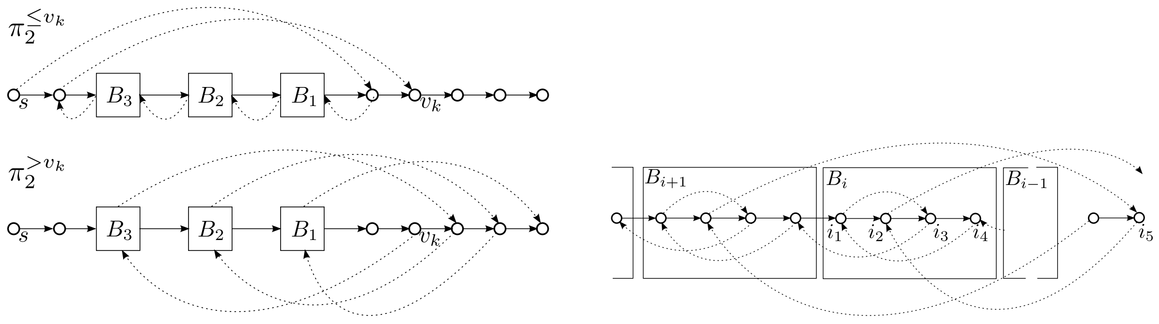

Figure 7.3: (SLF-)Pattern of a scenario where maximizing the number of updates per round will result in a $\Omega(n)$-round schedule, although a $O(1)$-round schedule would be possible. *Left:* An overview where $\pi_2^{\leq v_k}$ shows the edges of the new policy before v_k and $\pi_2^{>v_k}$ those behind v_k. *Right:* A detailed representation of the blocks B_i.

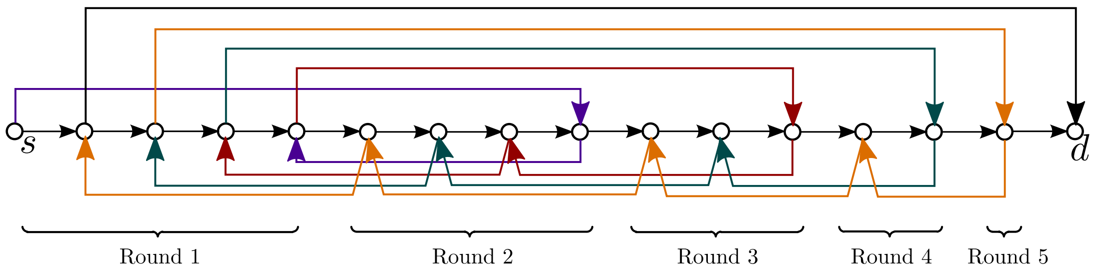

Figure 7.4: (RLF-)Pattern of a scenario where maximizing the number of updates per round will result in a $\Omega(\sqrt{n})$-round schedule, although a $O(1)$-round schedule would be possible. The colored edges represent the new path and are used for better visibility. The new path traverses the colors in the following order: purple, red, green, orange.

Delay in SLF. In certain situations a single greedy update step can increase the number of rounds for an update by a factor of up to $\Omega(n)$. Fig. 7.3 shows a scenario where a greedy update takes $\Omega(n)$ rounds even though an $O(1)$ round solution exists. The left side shows the general structure of the scenario which consists of several blocks B_i (more details on the right side). These blocks are connected via backward edges one by one, e.g., see the edge emerging from i_3. If a greedy algorithm picks all forward edges to be updated in a first round, it will include the nodes i_1 and i_2 as well as their representatives in the other blocks. The update of the i_1-type nodes essentially leads to a situation reminiscent of the one shown in Fig. 7.2, where many backward rules must be updated one after the other. Delaying the i_1-type nodes on the other hand will make it possible to update most of the backward edges in the next round, since the cycle is broken by the edges outgoing from the i_2-type nodes. This allows for an update in 4 rounds, independent of n. In case of the greedy algorithm, each additional block will increase the number of rounds by two. Each block consists of 4 nodes within the block and an additional node for connectivity to the right part of the line, resulting in $2n/5$ rounds: up to $n/10$ additional rounds are required.

Delay in RLF. Figure 7.4 shows a pattern where a greedy update strategy delays the update time in terms of rounds by a factor of $\Omega(\sqrt{n})$. The example shown takes five rounds to update in a greedy schedule. In the first round every forward edge is updated. From there on, there can always be a set of backward edges being updated. None of the later backward edges can be updated, as there will always be the possibility of a loop, since these nodes are always part of the path towards d. After each of the forward edges, a series of backward edges is started which is interleaving with each previous forward edge (see also color pattern in the figure). If this scenario had to be extended for an additional round, the edge pointing towards d must point to a node close before d. From here the sequence of backward edges is created to a node close before s from which we create a new edge to d. Hence, an additional round requires the same amount of nodes as the amount of rounds which the scenario needed, leading to $\Omega(\sqrt{n})$ rounds. There exists a strategy, which can update the scenario in three rounds. First, update the edge pointing towards d. Now every other node is on a different branch than the $s-d$ branch and hence can be updated in the second round. In the third round an update of s concludes the network update.

7.2.2 Greedy Updates are NP-Hard

Not only can a greedy update increase the number of rounds needed for a network by a factor up to $\Omega(n)$, but we also prove that maximizing updates for the loop-free network update problem is NP-hard, by presenting a polynomial-time reduction from the NP-hard *Minimum Hitting Set* problem. This proof is similar for both consistency models: SLF and RLF, and we can present the two variants together.

The inputs to the hitting set problem are:

1. A universe of m elements $\mathcal{E} = \{\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_m\}$.
2. A set $S = \{S_1, S_2, S_3, ..., S_k\}$ of k subsets $S_i \subseteq \mathcal{E}$.

The objective is to find a subset $\mathcal{E}' \subseteq \mathcal{E}$ of minimal size, such that each set S_i includes at least one element from $\mathcal{E}'$: $\forall S_i \in S : S_i \cap \mathcal{E}' \neq \emptyset$. In the following, we assume that elements are unique and can be ordered $\varepsilon_1 < \varepsilon_2 \ldots < \varepsilon_m$.

Theorem 8. *Maximizing the updates for the loop-free network update problem is NP-hard.*

The idea of the reduction is to create, in polynomial time, a legal network update instance where the problem of choosing a maximum set of nodes, which can be updated concurrently, is equivalent to choosing a minimal hitting set. While in the initial network configuration, essentially describing two paths from s to d, a maximal update set can be chosen in polynomial time (simply update all but only forwarding edges), we show in the following that already in the second round, a computationally hard problem instance can arise.

More concretely, based on a hitting set instance, we aim to construct a network update instance of the following form, see Figure 7.5. Note that throughout this proof we, instead of using dashed lines for the new policy, use colored lines to make it easier to follow the proof. For each element $\varepsilon \in \mathcal{E}$, we create a pair of branches ε^{in} and ε^{out}, i.e., $2m$ branches in total. To model the RLF case, in addition to the $\mathcal{E}$ branches, we add a source-destination branch, from s to d, depicted on the right in the figure.

We introduce the following to-be-updated new edges:

1. **Set Edges (SEs):** The first type of edges models sets. Let us refer to the (ordered) elements in a given set S_i by $\varepsilon_1^{(i)} < \varepsilon_2^{(i)} < \varepsilon_3^{(i)} \cdots$. For each set $S_i \in S$, we now create $m+1$ edges from each $\varepsilon_j^{(i)}$ to $\varepsilon_{j+1}^{(i)}$, in a modulo fashion. That is, we also introduce $m+1$ edges from the last element to the first element of the set. These edges start at the out branch of the smaller index and end at the in branch of the larger index. There are no requirements on how the edges of different sets are placed with respect to each other, as long as they are not mixed. Moreover, only one instance of multiple equivalent SEs arising in multiple sets must be kept.
2. **Anti-selector Edges (AEs):** These m edges constitute the decision problem of whether an element should be included in the minimum hitting set. AEs are created as follows: From the top of each in branch we create a *single* edge to the bottom of the corresponding out branch. That is, we ensure that an update of the edge from ε_i^{in} to ε_i^{out} is equivalent to $\varepsilon_i \notin \mathcal{E}'$, or, equivalently, every $\varepsilon_i \in \mathcal{E}'$ will not be included in the update set.
3. **Weak Edges (WEs):** These edges are only needed for the relaxed loop-free case. They connect the s-d branch to the other branches in such a way that no loops are missed. In other words, the edges aim to emulate a strong loop-free scenario by introducing artificial sources at the bottom of each branch. To achieve this, we create a certain number of edges from the s-branch to the bottom of every in branch. The precise amount is explained at the detailed construction part of creating parallel edges. See Figure 7.5 *bottom-left* for an example.

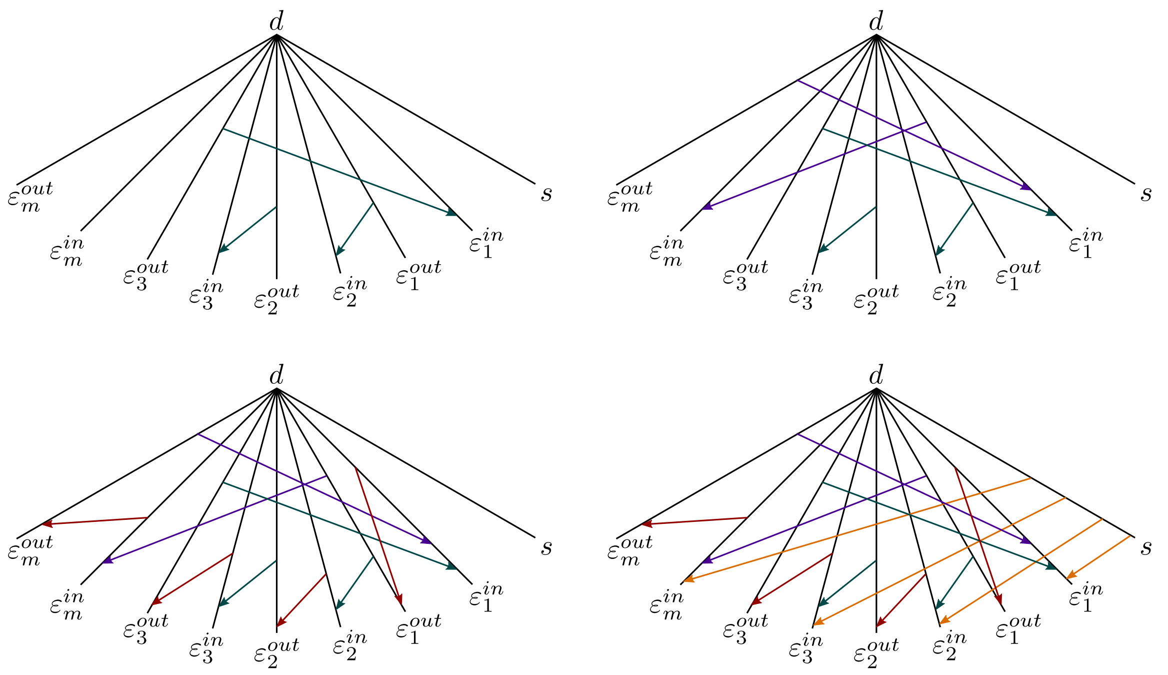

Figure 7.5: Example: Construction of network update instance given a hitting set instance with $\mathcal{E} = \{1, 2, 3, \ldots, m\}$ and $S = \{\{1, 2, 3\}, \{1, m\}\}$. Each element $\varepsilon \in \mathcal{E}$ is represented by a pair of branches, one called outgoing (out) and one incoming (in). Moreover, we add a branch representing the $s - d$ path on the very right. The black branches represent already installed rules (either old or updated in the first round), and new rules (colored) are between the branches. There are three types of to-be-updated, colored edges: one type represents the sets (green and purple), one type represents element selector edges (between in and out branch, red), and one type is required to connect the $s - d$ path to the elements (orange). We prove that such a scenario can be reached after one update round where all (and only) forward edges are updated. *Top-left:* Each green edge represents $m + 1$ edges, and is used to describe the set $\{1, 2, 3\}$: $(1, 2), (2, 3), (3, 1)$. *Top-right:* Each purple edge represents $m + 1$ edges and is used for the set $\{1, m\}$: $(1, m), (m, 1)$. *Bottom-left:* The red edges are single edges and are the element selector edges, representing the decision if an element is part of $\mathcal{E}'$ or not. *Bottom-right:* Each orange edge visualizes $m \cdot (m + 1)$ edges from the s-branch to the incoming branches of every $\varepsilon \in \mathcal{E}$.

The rational is as follows. If no *Anti-selector Edges* (AEs) are updated, all *Weak Edges* (WEs) as well as all *Set Edges* (SEs) can be updated simultaneously, without introducing a loop. However, since there are in total exactly m AEs but each set of SEs are $m + 1$ edges (hence they will all be updated), we can conclude that the problem reduces to selecting a maximal number of element AEs which do not introduce a loop. The set of non-updated AEs constitutes the selected sets, the hitting set: There must be at least one element for which there is an AE, preventing the loop. By maximizing the number of chosen AEs (maximal update set) we minimize the hitting set.

Let us consider an example: In Figure 7.5 *bottom-right*, if for a set S_i every AE of $\varepsilon_i \in S_i$ is updated, a cycle is created: updating edges ε_1^{in} and ε_m^{in} results in a cycle with the $m + 1$ edges from ε_1^{out} and ε_m^{out}.

Note that the resulting network update instance is of polynomial size (and can also be derived in polynomial time).

In the remainder of the proof, we show that the described network update instance is indeed legal, e.g., we have a single path from source to destination, and can be obtained after one update round.

Concepts and Gadgets

Before we describe the details of the construction, we first make some fundamental observations regarding greedy updates.

Creating Forward Edges: Delayer First we describe the delayer concept, which is required to establish forwarding edges for the second round. Observe that forwarding edges (v_1, v_2), with $v_1 < v_2$, are always updated by a greedy algorithm in the first round. A *delayer* is used to construct a forward edge (v_1, v_2), with $v_1 < v_2$, that is created in the second round.

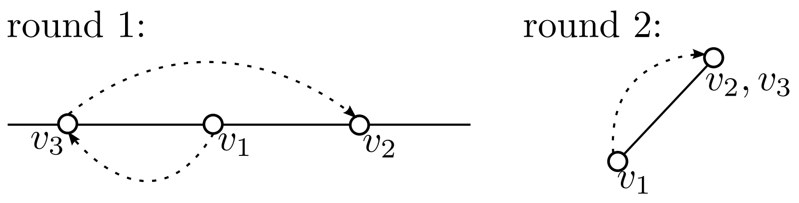

Figure 7.6: Delayer concept: A forwarding edge (v_1, v_2v_3) can be created in round 2 using a helper node v_3.

The delayer for edge (v_1, v_2) consists of two edges: an edge pointing backwards to v_3 from v_1 with $v_3 < v_1$, plus an edge pointing from there to v_2. The forward edge (v_3, v_2) will be updated in the first round, which yields an edge (v_1, v_2) due to merging (see Figure 7.6).

Creating Branches We next describe how to create the in and out branches as well as the s branch pointing to the destination d (recall Figure 7.5). This can be achieved as follows: From a node close to the source s, we create a path of forward edges, which ends at the destination. Each of these forward edges will be updated in the first round, and hence merged with its respective successor, which will be the destination for the very last forward edge. The nodes belonging to these forward edges are called *branching nodes*. Every node in between two *branching nodes* will be part of a new branch pointing to the destination. See Figure 7.7 for an example. The rightmost node before the *branching node* on the line will also be the topmost node on the branch after the first round update (as long as it has an outgoing backward edge, hence not being updated in the first round). Therefore, throughout this section we use the terms right and high (rightmost-topmost) and left-low for the first and second round interchangeably.

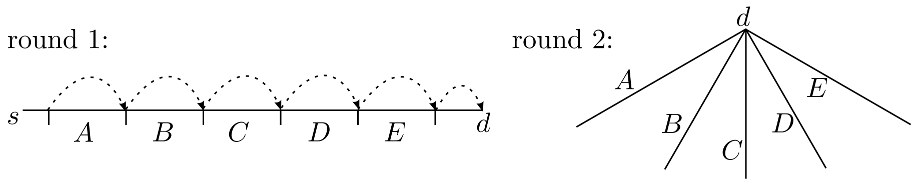

Figure 7.7: Creating branches after a greedy update of forward edges.

Introducing Special Segments As we have discussed, we split the line (old path) into disjoint segments which will become independent branches at the beginning of the second round. In addition, there will be two special segments, one at the beginning and one at the end. The first will not even become an independent branch at the beginning of the second round, but is merely used to realize the delayer edges.

Behind the very last segment (ε_1^{in}) and just before d, there is a second special segment, which we call *weak*: it is needed to create the branch with the source s at the bottom and its connections to the other ε_i^{in} branches.

Creating Parallel Edges In our construction, SEs come in groups of $m+1$ edges. These edges must eventually be part of a legal network update path, and must be connected in a loop-free manner. In other words, to create the desired problem instance, we need to find a way to connect two branches b_1 and b_2 with $m+1$ edges, such that there is a single complete path from s to d. Furthermore, these edges should not form a loop.

The corresponding edges can be constructed as follows, henceforth called the *zigzag-approach* (Figure 7.8): Split the branch b_2 into two different parts. The first part b_{2-b} on the left side (respectively bottom of the branch) is used to complete the path but can only be reached over backward edges. The second part b_{2-t} receives the incoming edges from the other branch, b_1. Start at a node, say v_{o-1} on b_1. Here create an edge to a node of b_{2-t}, say v_{i-1} and from there a backward edge to a node of b_{2-b}, say v'_{i-1}. Afterwards use a delayed edge to connect to v_{o-1}'s right (respective to the line) neighbor, v_{o-2}. From here, create the next edge to v_{i-1}'s right neighbor, v_{i-2} and the backward edge to v'_{i-2} on b_{2-b} again. Repeat this procedure $m+1$ times.

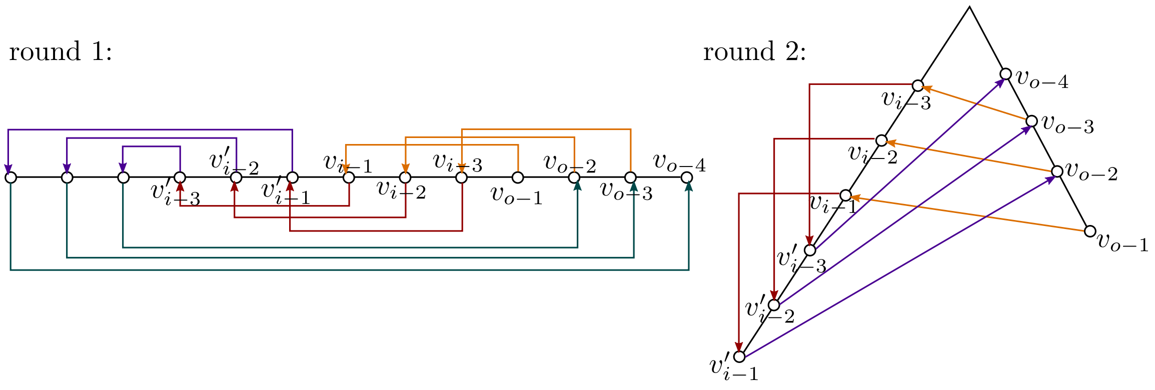

Figure 7.8: Connecting two branches with 3 edges. The backward edges shown in red assure that there will not be a way back in the second round from the *in* branch to the *out* branch.

This zigzag construction indeed ensures loop-freedom. To see this, note that all incoming edges from the b_1 branch will always connect to the b_{2-t} part of b_2. From here the way back to b_1 (or potentially any other branch that connects with b_{2-t}) can only be completed if any of the backward edges from b_{2-t} to b_{2-b} has been updated. This cannot be the case at all for the strong loop freedom definition, since no backward edge can ever be updated and the edge is backward in the first and the second round. For relaxed loop freedom it also cannot be updated in the first round since it would create a loop on the $s-d$ path, which is a line of all nodes in the first round. In the second round it will not be included since we make sure that a maximum update always includes the WEs, which will be incoming at the very left side of b_{2-t} and hence cannot be updated in the same round as any backward edge on this branch.

In order to ensure that all the WEs will always be included, we create $m\cdot(m+1)$ WEs to every *in* branch. This is always more than the amount of backward edges on a single branch b_2 since they are only created as a path completion for the SEs. We have at most $(m-1)\cdot(m+1)$ SEs incoming in a case where this node is connected to every other node (but itself). Choosing the WEs will immediately force that none of the backward edges from b_{2-t} to b_{2-b} will be included, as they might cause a cycle on a path that might be in between s and d.

The $m\cdot(m+1)$ WEs to a branch b can be created simpler. Here we do not need to take care about other branches being reached from the *weak* branch. Hence, we can create the way back to the *weak* branch without the detour over the b_t part. This is because the WEs will always be the incoming edges on the leftmost part of b_t without the possibility of any other parallel edges making use of them.

Connecting the Pieces

Given these concepts and building blocks, we are able to complete the construction of our problem instance.

Realizing the Delayer The first created segment, *temp*, serves for edges that are created using the *delayer* concept. This is due to our construction: every node that is created in this interval in our construction is a forward node and therefore updated in the first greedy round. The *temp* segment is located right after the source s on the line.

Realizing the Branches We create two segments for each $\varepsilon \in \mathcal{E}$, one out and one in, and sort them in descending global order (and depict them from left to right) with respect to $\varepsilon \in \mathcal{E}$, with the out segment closer to s than the in segment for each ε, i.e. $temp, \varepsilon_m^{out}, \varepsilon_m^{in}, \cdots, \varepsilon_2^{out}, \varepsilon_2^{in}, \varepsilon_1^{out}, \varepsilon_1^{in}$.

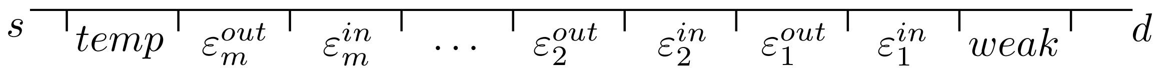

Figure 7.9: Illustration of how to split the old line into segments according to the amount of needed branches in the second round.

Connecting the Path We now create the new path from the source s to the destination d through all the different segments. This path requires additional edges. We ensure that these edges can always be updated and hence do not violate the selector properties. Moreover, we ensure that they do not introduce a loop.

In order to create a branch with s at the bottom (to ensure that the proof also holds for relaxed loop-freedom), we start our path from the source s to a node w_b on the very left part of the *weak* segment. From here we need to create the $m \cdot (m+1)$ connections to every other ε_i^{in} branch, more precisely to the very left of the top part of this branch ε_{i-t}^{in}: the Weak Edges (WEs). Starting from w_b, we create the $m \cdot (m+1)$ zigzag edges we postulated earlier (see Section 7.2.2) to the ε_1^{in} segment. Once this is done, we repeat this process for the remaining ε_i^{in} connecting them in the same order block by block, as they are ordered on the line. See Figure 7.10.

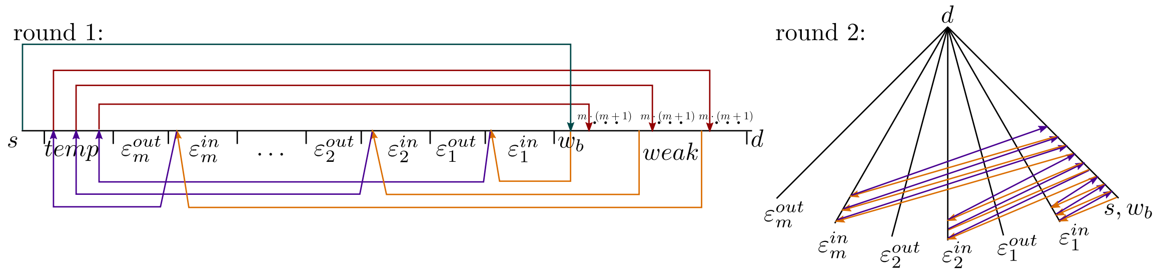

Figure 7.10: Creating the branch with the source at the bottom and $m \cdot (m+1)$ connections to each ε_i^{in} segment of the line, as shown in Section 7.2.2. The $m \cdot (m+1)$ connections are visualized as a single edge in the first round to enhance visibility.

At the beginning of the second round, we now have a branch with the source s at the bottom and $m+1$ edges to each of the ε_i^{in} branches. The next step is to connect the out branches with the in branches (the Set Edges). For each set $S_j \in S$ and each pair $\varepsilon_i, \varepsilon_l \in S_j$ with no $\varepsilon' \in S_j, \varepsilon_i < \varepsilon' < \varepsilon_l$, we create $m+1$ edges from ε_i^{out} to ε_l^{in}, more precisely to the top part ε_{l-t}^{in} somewhere above the WEs. Each pair $\varepsilon_i, \varepsilon_l$ only needs to connect once with the $m+1$ edges, even if it occurs in several different sets of S. The last element ε_i of a set S_j additionally needs to be connected to the first element of the set (the modulo edges).

After the $m+1$ connections to ε_m^{in}, the path returns at the rightmost (or highest in the (s,d)-branch) node in the *weak* segment. From here we create a backward edge to the left part of ε_1^{out}. Here, we create $m+1$ connections to every ε_i^{in}, which is the next larger element in any of the sets. An example is shown in Figure 7.11.

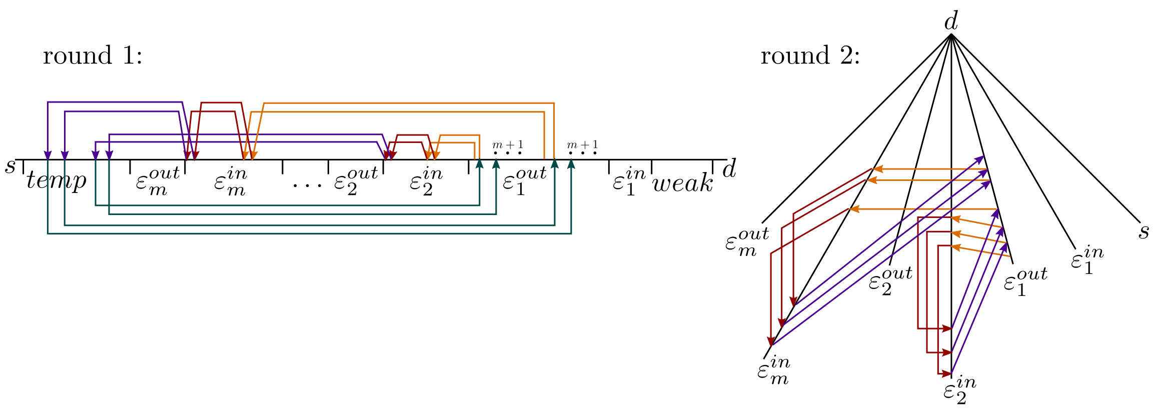

Figure 7.11: Connecting the ε_1^{out} branch with the branches ε_2^{in}, ε_3^{in}, ε_m^{in}. This scenario would be created if there were the sets: $\{1,2,...\}$,$\{1,3,...\}$, $\{1,m\}$. The orange edges show the outgoing edges from ε_1^{out}. The red edges are the backward edges from the top part of a branch ε_i^{in} to its bottom part (ε_{i-t}^{in} to ε_{i-b}^{in}). The purple edges are the way back from ε_i^{in} to ε_1^{out} and are needed to complete the path.

To complete the $m+1$ connections for every pair, we proceed as follows: we connect the ε_1^{out} branch to all required in-branches, then add the edge from ε_1^{out} to the ε_2^{out} branch, then add the edges from the ε_2^{out} branch to all required in-branches, etc. Generally, we interleave adding the edges from the ε_i^{out} branch to all required in-branches and then add the i-out to $(i+1)$-out edge. Until the path arrives at the end of the last out branch, ε_m^{out}:

- *Step A - Create the set specific $m+1$ edges:* Here we do $m+1$ connects to every successor in the respective sets (at most once per pair). If this element is the largest element in a set, it needs to be connected to the in part of the smallest element of this set again. Here the delayer concept needs to be used for the modulo edges.
- *Step B - Connecting the out branches:* In order to create the next $m+1$ connections from the next out segment ε_{i+1}^{out}, we need to connect it from our current out segment ε_i^{out}. The edge therefore needs to point to the rightmost part of ε_{i+1}^{out}. Since this edge is always a backward edge in the first round (we start closer to the destination and move backward towards the source), it will turn out to be an edge which points to the very top of ε_{i+1}^{out} at the beginning of the second round. This assures that there are no loops created, since the only way is going directly towards the destination. From here we create an edge pointing to the very left side of ε_{i+1}^{out} (evolving to a backward rule from top to bottom of the branch in the second round, hence not being part of the update set in the first nor the second round).

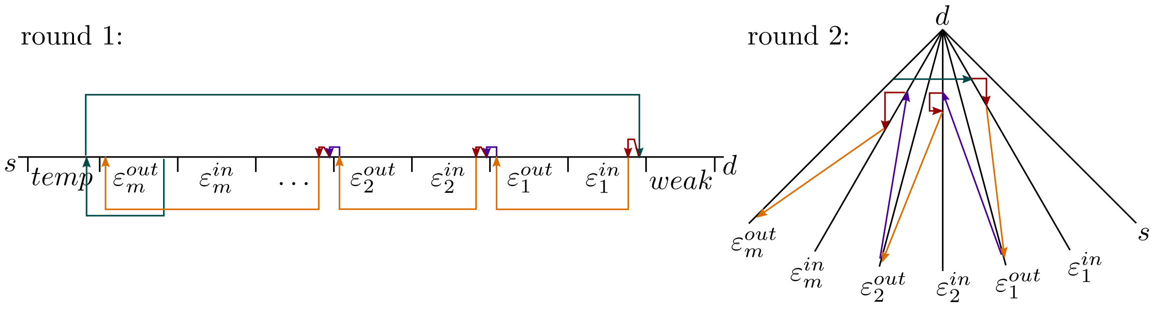

Figure 7.12: Connecting the in and out branches of every ε_i, shown in orange. The edges shown in purple are needed to keep the path complete and the backward edges in red are needed to ensure that only the destination can be reached from that point in the second round.

To finish the construction, we need to add the anti-selector edges (AEs), and connect the in and out branches of every single ε_i with each other. The goal is to create, for each given i, an edge from the top of each ε_i^{in} to the bottom of each ε_i^{out}. This way, if this edge is included in the update, a loop may be formed: as every incoming edge to ε_i^{in} arrives below the AEs start point and every outgoing edge on ε_i^{out} is above AEs destination. The decision to not include one of these edges is equivalent to $\varepsilon_i \in \mathcal{E}'$ in the minimum hitting set problem. In order to keep the path connected we also need to include edges from ε_i^{out} to ε_{i+1}^{in}, compare Figure 7.12. These edges will point to the top of ε_{i+1}^{in} and therefore do not create loops, since the only way is going directly to the destination. From here, we create another backward edge to its left neighbor such that there is no possible other way than traversing towards d from this point. Without this backward edge it might be possible to create loops, since it would create connections between branches, which are not both in a set S_i of the hitting set problem. Therefore, an update of one of the additional connector edges never leads to a loop and hence, they can all be included in the update set of the second round.

The construction of these edges is straightforward. From the end of the current path which is located on the ε_m^{out} segment, we create a delayed edge (over *temp*) to the very right part of the ε_1^{in} segment. From here, we construct the path as described with a short backward edge to its left neighbor and then to the very left part of the ε_i^{out} segment and again to the very right part of the ε_{i+1}^{in} segment afterwards, until we arrive at the very left part of the ε_m^{out} segment.

The only thing left is to create the segments and branches for the second round. From ε_m^{out}, we create a backward edge to the *temp* part. From here, we use the branching concept and connect all intermediate nodes in between the single parts that we created on the line (see Figure 7.13).

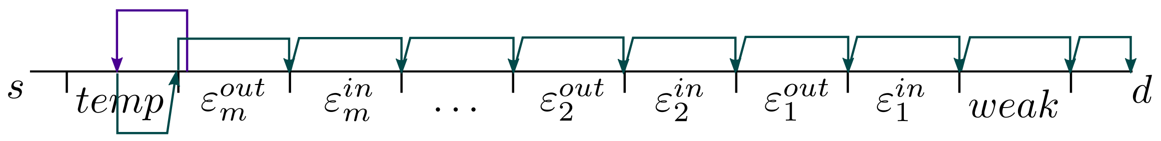

Figure 7.13: Connecting the segments with forward edges. This creates a single branch from the dst for every segment due to the merging. The edge shown in purple is connecting this step with the step before.

Overview We showed how to construct valid network update problems for any instance of the NP-hard *Minimum Hitting Set* problem. Within the construction, we ensured that after a greedy first round update, we end up in a situation where choosing the maximum set of updateable nodes is equivalent to choosing the minimal hitting set. Hence, we have not only shown that trying to find a maximum set of updateable nodes can increase the overall update time in terms of numbers of rounds by up to a factor of $\Omega(n)$ but that it is NP-hard to even find these solutions. This result holds true for both SLF and RLF.

7.2.3 Polynomial-Time Algorithms

While the computational hardness is disappointing, we can show that there exist several interesting specialized and approximative algorithms.

An Optimal Algorithm There are settings where an optimal solution can be computed quickly. For instance, it is easy to see that in the first round, in a configuration with two paths, updating all forward edges is optimal: Forward edges never introduce any loop, and at the same time we know that backward edges can never be updated in the first round, as this edge alone (i.e., taking effect first), immediately introduces a loop.

We first present an algorithm for SLF, and later extend it to RLF.

Theorem 9. *A maximum, SLF update set can be computed in polynomial-time in trees with two leaves.*

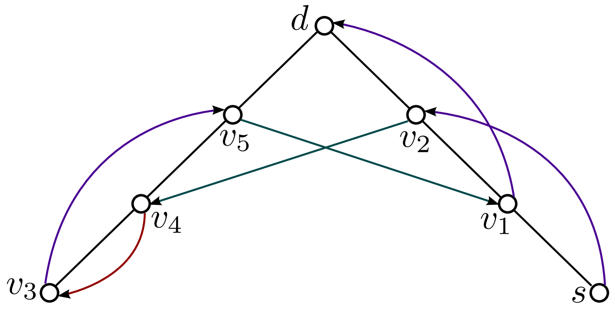

Figure 7.14: Concept of horizontal edges shown in green. Both horizontal edges (v_2, v_4) and (v_5, v_1) are crossing each other. The backward edge (v_4, v_3) is shown in red and the forward edges in purple.

Proof. Suppose the policy graph $G = (V, E)$ (the union of old and new policy edges) is given as a straight line drawing Π in the 2-dimensional Euclidean plane, such that the old edges of the 2-branch tree form two disjoint segments which meet at the root of the tree (the destination), and such that each node is mapped to a unique location. Given the graph, such a drawing resp. embedding in the plane can be computed efficiently.

Recall that there are three types of new edges in the graph (see also Figure 7.14): forward edges (F), backward edges (B) and horizontal edges (H), hence $E = H \cup B \cup F$. Moreover, recall that forward edges can always be updated while backward edges can never be updated in SLF. Thus, the problem boils down to selecting a maximum subset of H, pointing from one branch to the other. If there is a simple loop $C \in G$ such that $H^C = E(C) \cap H \neq \varnothing$, then $|H^C| = 2$ and we say that the two edges $e_1, e_2 \in H^C$ cross each other, written $e_1 \times e_2$. See Figure 7.14 for an example of crossing edges. Note that there could be other edges which intersect w.r.t. the drawing Π, but those are not important for us.

Now create an auxiliary graph $G' = (V', E')$ where $V' = \{v_e \mid e \in H\}$, $E' = \{(v_{e_1}, v_{e_2}) \mid e_1, e_2 \in H : e_1 \times e_2\}$. The graph G' is bipartite, and therefore finding a minimum vertex cover $VC \in V(G)$ is equivalent to finding maximum matching, which can be done in polynomial time. Let $H' = \{e \mid e \in H : v_e \in VC\}$, then the set H' is a minimum size subset of H which is not updatable. Therefore the set $H \setminus H'$ is the maximum size subset of H which we can update in a SLF manner.

We conclude the proof by observing that all these algorithmic steps can be computed in polynomial time. ∎

Given the algorithm for strong loop-freedom, we next show that the result also holds for relaxed loop-freedom.

Theorem 10. *A maximum, RLF update set can be computed in polynomial-time in trees with two leaves.*

Proof. We prove the theorem by presenting a polynomial-time reduction to the strong loop-free case. Let us fix the path (i.e., branch) in the tree consisting of the currently active edges which includes both the source and the destination: $P_d^s = (s = v_0, \dots, v_n = d)$. See Figure 7.14 for an illustration. Note that in the branch which contains s, d there may exist some vertices which have a path to s: those vertices are irrelevant for our construction and we just consider the path P_d^s of the old policy starting at s.

Let us refer to the entire path in the other branch by $P_2 = (u_1, \dots, u_n)$, omitting the vertex d. Here, node u_1 is the node with the lowest y-coordinate in the drawing Π. In this case, we can update B edges as long as they are not in any path from s to d. Therefore, the objective is to find the maximum subset $S \subseteq H \cup B$ which is not part of any loop reachable from s.

Without loss of generality, we can assume that there is no B edge which connects two vertices of the path P_s^d: we cannot update those edges anyway, and hence we can ignore them. If we simulate B edges with H edges, then the the problem becomes equivalent to the SLF one which is in P. To see this, suppose $B = \{e_1, \dots, e_k\}$, create a new graph G' out of G by adding k vertices $\{v^{e_1}, \dots, v^{e_k}\}$ to P_d^s to obtain $P_{s,d}^k = (s = v_1, v^{e_1}, \dots, v^{e_k}, v_1, \dots, v_n = d)$, and a set of edges $H' = \{(u, v^{e_i}) \mid e_i \in B, u = tail(e_i)\}$, where the tail of an edge $e = (u, v)$ is u. After that, we delete all edges in B. We can now find the maximum set of the horizontal edges in G' which can be updated using the same algorithm as we had for SLF. If any edge

$H^e \in H'$ has been chosen in the algorithm for SLF in the G', we choose $e \in E(G)$ for the update as well. These edges together with all forward edges and the chosen edges from the set H in G' give us the maximum set of edges $\mathcal{H} \in E(G)$ which we can safely be updated in the RLF model in G. Let $\bar{\mathcal{H}} := E(G) - \mathcal{H}$.

Notice that there is no loop reachable from s which uses only edges in $H \cup F$ in $G_{opt} = (V(G), \mathcal{H})$, by the construction of G'. Moreover, there is no loop in G_{opt} which uses edges in B and which is reachable from s. To see the correctness of the second claim, suppose an edge $e = (u, v) \in B$ is chosen such that there is a path P which goes through an edge $e' \in H$ and connects s to u. Then, in G' there was an edge $e'' = (u, v^{e'})$. But at least one of the edges e' and e'' has been eliminated for the update in G' then, by the construction of the algorithm, either there is no edge like e, or there is no path like P which goes through e'.

We proved that the solution is valid. For the optimality, we just note that there is a one-to-one relationship between simple loops of G which are reachable from s, and loops of G'. This means that if we make G' loop-free, we transfer G to the graph which has no loop reachable from s. So any optimal solution for G' is an optimal solution for G. ■

Approximation Algorithms In scenarios for which there is no optimal polynomial time scheduling algorithm, at least good approximations may exist. It is easy to see that the problem for strong loop-freedom (for SLF) is $1/2$-approximable in general, as the problem boils down to finding a maximum subset of H edges which are safe to update, and at least half of the H edges are pointing out to the left resp. right, and we can take the majority. However, for a small number of leaves, even better approximations are possible.

Theorem 11. *The optimal SLF schedule is $2/3$-approximable in polynomial time in scenarios with exactly three leaves. For scenarios with four leaves, there exists a polynomial-time $7/12$-approximation algorithm.*

Proof. We use an approximation preserving reduction to the d-hitting set problem which is $\Sigma_{i=1}^{d} 1/i - 1/2$-approximable [39], and particularly, we use a 3-hitting set which gives us a $2/3$-approximation algorithm.

Let $G = (V, E)$ be the update graph with at most three leaves and let H be the set of the horizontal edges. For every closed simple loop $C \subseteq G = (V, E)$ we have $C^H = E(C) \cap H \neq \emptyset$. Furthermore $C^H \leq 3$. Having these observation we construct our hitting set as follows. Let $|H| = m$ and let F be a mapping one to one mapping $F : H \to [m]$. For each simple loop C_j let $C_i^H = \{s_1, s_2, s_3\}$, create a subset $S_i = \{F(s_1), F(s_2), F(s_3)\}$. Note that if $|C_i^H| = 2$ then we have a subset S_i of size 2. There are at most $|H|^3$ simple loops, as choosing any set of size at most 3 edges from E forces at most one simple loop so we have totally $\binom{m}{3}$ loops with 3 edges in E and $\binom{m}{2}$ loops with two edges in E. Furthermore the hitting set for $S_1, \ldots, S_t$ gives a minimum set of update edges to be removed, on the other hand every subset S_i is of cardinality at most 3. So it gives $4/3$-approximation on the size of subset $H' \subseteq H$ which we do not update. On the other hand in the optimal solution H^{opt} we have $|H'| \leq H$ or in the other word $|H^{opt}| \geq |H'|$, so the approximation factor will be at least $(1 - 1/3)|opt|$ which is $2/3$-approximation as claimed. For the 4 leaves similar argument works so we omit the proof. ■

We also give a simple $1/3$-approximation for RLF in an arbitrary update tree.

Lemma 5. *Given an update graph G. There is a $1/3$-approximation algorithm which runs in polynomial time to find a maximum set of update edges for RLF.*

Proof. For the sake of simplicity, suppose that there is no B edge in the branch which contains both source and destination, as we cannot update any of such edges. We can update all F edges, but we need to decide which B and H edges to update. There are two cases:

1. $|B| \geq |H|/2$: Then we do not update any H edge, but we update all the B edges, This gives us at least a $|B|/(|H| + |B|)$-approximation, which is at least a $1/3$-approximation.
2. $|B| < |H|/2$: We arrange the H edges in an straight line drawing of G and take at least half of the edges, which all point from right to left (or vice versa). This again this gives us a $\frac{|H|/2}{|H|+|B|} \geq 1/3$ approximation polynomial time algorithm.

■

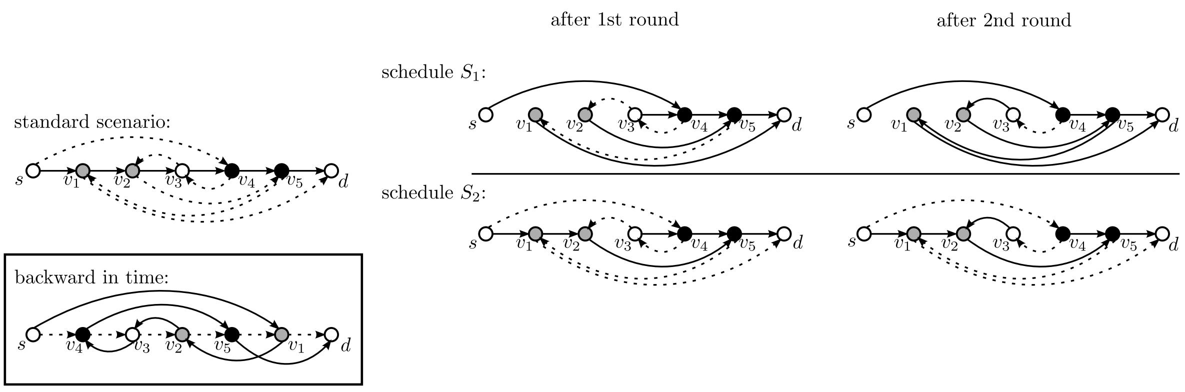

Figure 7.15: *Left:* "Looking backward in time", an example with reversed update pattern (from *dashed* to *solid* path). We obtain the following classification: v_1 is **FF**; v_2, v_3 are **FB**; v_4 is **BB** and v_5, v_6 are **BF**. *Right:* Intuition why node updates can be moved from round 2 to round 1 or 3. There are two different valid update schedules for the standard scenario. Schedule S_1 is updating everything as early as possible, e.g., **FB** node v_2 in round 1 and **BF** node v_6 in round 2. Schedule S_2 is updating everything as late as possible, e.g., v_2 in round 2 and v_6 in round 3. We depict updated nodes without their outgoing solid edges (no new packets will be sent this way), and dashed edges turn into solid edges.

7.3 Fast Updates Are Difficult

We have seen that maximizing the updates is not only hard to compute but can also delay the overall update, therefore we turn to minimizing the number of rounds instead and ask: "How many rounds are needed to update a network in a (strongly) loop-free manner?" On the one hand, the problem seems difficult: the problem of breaking cycles even in a single round, is related to the well-known NP-hard Feedback Arc Set Problem. On the other hand, our graphs have a very special structure, as they essentially only consist of two simple paths (namely the old and the new policy).

In this section, we show that updating networks quickly is difficult, even for such simple graphs: while problem instances allowing for 2-round schedules are trivial (Section 7.3.1), deciding whether 3-round schedules exist is NP-complete (Section 7.3.2). Also recall our example from Fig. 7.2 which shows that there exist problem instances which cannot be updated in less than $\Omega(n)$ rounds.

7.3.1 2-Round is Easy

Before we show how to find 2-round update schedules efficiently, let us introduce the following edge (resp. node) classification, which will be useful more generally. We already discussed the notion of *forward* and *backward* dashed edges (resp. nodes), indicating whether a dashed edge points in the same direction as the solid edge. This distinction is useful as, for example, it is always safe to update any number of forward-pointing edges: they can never introduce any loops. However, we can also classify edges from the other side, from the destination and "looking backward in time": as if we were updating edges from the dashed ("new" π_2) rules to the solid ("old" π_1) ones, starting with the last round. Given this backward perspective, we can classify the old (*solid*) rules as *backward* or *forward* relative to the new ones (*dashed*): we just need to draw the new policy as a straight path and see, if the old rule points forward or backward.

Based on this classification, we propose two-letter codes to describe the nodes—the first letter will denote, whether the outgoing dashed edge points forward (**F**) or backward (**B**). Similarly, the second letter will describe the solid edge relative to the dashed path. Now, it is easy to see that in the last round, we can update any subset of rules which are either **BF** or **FF**, just like in the first round where we can update any **FB** or **FF**. An example can be seen in Fig. 7.15 on the left.

Given this intuition, we can determine whether two rounds are sufficient: if there is any **BB** edge, it can neither be updated in the first round, nor in the last, so two rounds are not enough. Otherwise, we update **FB**s in the first round, **BF**s in the second round, and have complete freedom on when to update the **FF** nodes.

7.3.2 3-Round is Hard

Unfortunately, it is already NP-complete to decide whether a problem instance has a 3-round update schedule.

Theorem 12. *Deciding whether a $k = 3$-round schedule exists is NP-complete.*

The k-round problem is certainly in NP: the correctness of a schedule can be verified easily. The hardness proof proceeds as follows. First we make a couple of observations which allow us to narrow the ground for choosing 3-round update schedules, reducing the problem to the selection of edge subsets. Second, we will present a slight modification of $3\text{-}\textsc{Sat}$ and—using gadgets—transform it into an instance of the edge selection problem. Finally, the graph built using the gadgets will be patched up to a proper instance of the network update problem (namely, two paths traversing the same set of nodes).

Classifying Nodes

When we aim for three rounds, the **FB** nodes can be updated in the first or second round. As we will observe in the following, it is however never *necessary* to update **FB** nodes in the second round: everything can just as well be done in the first round.

Lemma 6. *If there exists a 3-round update schedule $\mathtt{S}$ which updates any nodes $V' \subseteq V$ of type **FB**, then there is also a 3-round update schedule which updates all nodes of V' in the first round. The same holds true for nodes of type **FF**.*

Proof. Consider the temporary forwarding graph $G_t(X) = (U, X, E_t)$ during the t^{th} round update of $\mathtt{S}$, for $t \in \{1, 2\}$. Since $\mathtt{S}$ is correct, both $G_1(X) = (U, X, E_1)$ and $G_2(X) = (U, X, E_2)$ are loop-free, for any subset $X \subseteq U_t$. By moving updates of forwarding nodes **FB** and **FF** from round 2 to round 1, we will make G_2 only sparser, and will hence not introduce loops. However, also G_1 will remain loop-free, as the forwarding edges **F**· respect the topological order of π_1. ■

The same argument also holds in the other direction, using our "backward perspective": We can move **BF** (and **FF**) updates to the last round. Therefore, without loss of generality, we focus our analysis on schedules where all the **BB** nodes are updated in the middle (i.e. second) round, all **FB** nodes in the first round, and all the **BF** nodes in the last round. Thus, the problem boils down to finding a distribution of the **FF** updates to the first and the third round. As we will show in the following, finding such a distribution is NP-hard.

Fig. 7.15 provides intuition for why **FB** updates can be moved into the first round and **BF** updates in the third round. The right part shows two different 3-round schedules for a given scenario. The **FB** node v_3 needs to be updated in the first round in any valid 3-round schedule, since the only **BB** node v_4 needs to be updated in the second round. Schedule S_2 updates the **FB** node v_2 in round 2 and schedule S_1 shows that it would also be possible to update it in the first round. The **BF** node v_6 is updated in round 2 in S_1 and delayed to round 3 in S_3. According to Lemma 6, there also exists a schedule S_3 updating every **FB** node in the first round and every **BF** node in the third round $(U_1 = \{v_1, v_2, v_3\}, U_2 = \{v_4\}, U_3 = \{v_5, v_6\})$.

In order to be able to update every **BB** node in the second round, one needs to be careful which (of the **FF**) nodes to update in the first and which in the third round. Fig. 7.16 shows a snippet of a line where the **BB** node v_6 needs to be updated in the second round. An update of **FF** node v_4 in the first round would enable this update for the second round, but updating the **FF** node v_3 as well would render an update of v_6 impossible. Node v_5 is **B**· and cannot be updated in the first round, and hence an update of v_6 would result in a loop $(v_3 \to v_5 \to v_6 \to v_3)$.

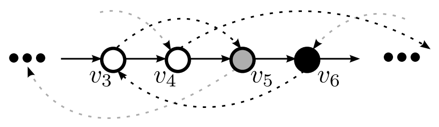

Figure 7.16: Choosing the right set of **FF** nodes is important. An update of only v_4 would enable the **BB** node v_6 to be updated in the second round. An additional update of v_3 would then lead to a loop (note that v_5 will definitely not be updated in the first round).

Modifying 3-CNF

For our reduction, we take an instance of the 3-SAT problem, $\mathcal{C}$, which we will eventually transform into an instance of a network update problem that is updatable in 3 rounds, if and only if the formula is satisfiable. However, we will first modify $\mathcal{C}$, using a standard construction, and replace each appearance of a variable in $\mathcal{C}$ using a new variable: concretely, a variable appearing λ times in $\mathcal{C}$ decays into $\lambda + 4$ new variables. By this trick, we will reduce the number of times any (new) variable appears in the (new) formula, allowing us to implement the low in- and out-degree requirements of our network update problem.

We create the following clauses:

1. For every variable x, we create variables

$$x_0, x_1, \ldots, x_{p_x}, x_l, \overline{x}_0, \overline{x}_1, \ldots, \overline{x}_{n_x}, \overline{x}_l,$$

   where p_x is the number of positive appearances of x, and n_x the number of negative appearances. In every clause we replace the literals with the appropriate new variables (from the collections $x_1, \ldots, x_{p_x}$ and $\overline{x}_1, \ldots, \overline{x}_{n_x}$). Also, for every original variable x we add an "*assignment clause*" $(x_0 \vee \overline{x}_0)$.

2. For every original variable we add "*implication clauses*" $(x_i \rightarrow x_{i+1})$ for $i = 0 \ldots p_x - 1$ and $(\overline{x}_i \rightarrow \overline{x}_{i+1})$ for $i = 0 \ldots n_x - 1$; the last implications, for $i = p_x$ resp. $i = n_x$ must lead to x_l and $\overline{x}_l$ respectively $((x_{p_x} \rightarrow x_l)$ and $(\overline{x}_{n_x} \rightarrow \overline{x}_l))$.

3. Finally, for every original variable x, we add an "*exclusive clause*" $(\neg x_l \vee \neg \overline{x}_l)$.

For each variable x, with the *assignment clause*, we ensure that at least one literal is true; with the *exclusive clause* we ensure that at most one literal is true; and with the *implication clause*, we ensure that the value is consistently preserved through all clones.

It is straightforward to translate any satisfying assignment of variables of one formula to the other, therefore the satisfiability problem for the new formula is equivalent to the original one. We will refer to the modified formula by $\mathcal{C}'$.

Creating and Connecting the Gadgets

For the reduction, we will create (network) gadgets representing the different clauses. Concretely, first, for every variable x_i in $\mathcal{C}'$, we create a node x_i, which will be of type **FF** (we will refer to the node using the variable's name). The idea is that updating the node in the first round will correspond to the positive valuation of the variable. In general, we will create for each gadget a path of solid edges pointing upward; eventually, we will connect these paths from left to right (using solid edges), to establish policy π_1.

Every clause $\mathcal{K}$ is encoded as a gadget in the graph using a separate solid path (drawn as a vertical line pointing upwards) with the variable-related (x_i) **FF** nodes on it. Above those nodes on the path, there is a **BB** node, $v_1^{\mathcal{K}}$, the starting point of a backward, dashed edge that will end just below the variables with a node $v_2^{\mathcal{K}}$ (Fig. 7.17 *left*). The backward edge and the solid path form a cycle, which needs to be disconnected in the first round. The only way to do this, is by updating at least one of the variable–related edges. Obviously, the dashed, forward edges starting at the **FF** nodes inside the clause must reach outside the clause-related backward edge $(v_1^{\mathcal{K}}, v_2^{\mathcal{K}})$. In fact, they will end just below the nodes representing the variables that are followed

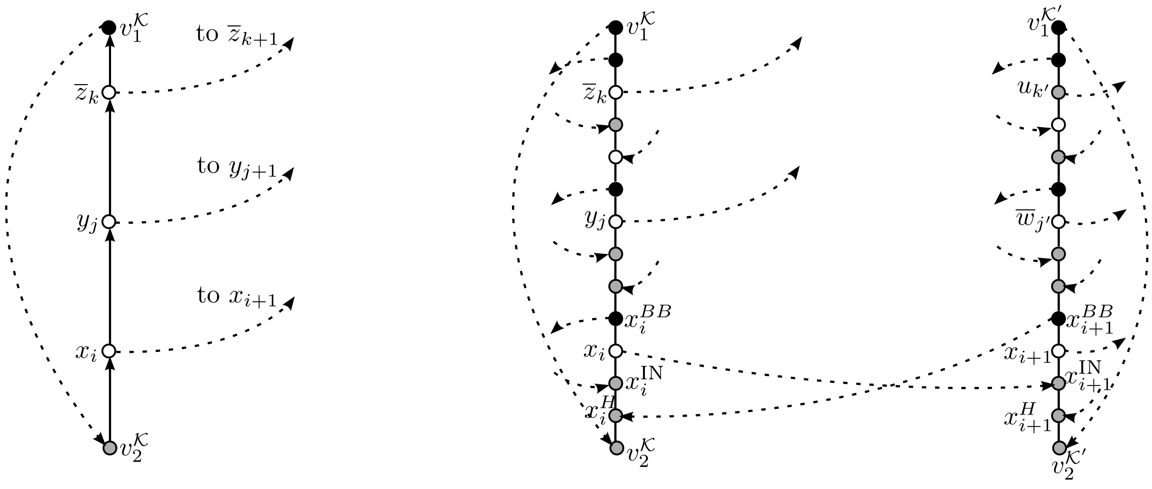

Figure 7.17: *Left:* Gadget for clause $x_i \vee y_j \vee \overline{z}_k$. At least one node needs to be updated to prevent the loop over $v_1^{\mathcal{K}}$ and $v_2^{\mathcal{K}}$. *Right:* More details about the gadget including also the implication clause $x_i \to x_{i+1}$ representation, and a second clause $x_{i+1} \vee \overline{w}_{j'} \vee u_{k'}$. It is assured that x_{i+1} is updated if x_1 is updated, otherwise the **BB** edge from x_{i+1}^{BB} would form a cycle. White nodes will eventually be **FF**, black nodes **BB**. The grey nodes will later be assured to be of type **B**·, to guarantee that they cannot be updated in the first round.

in the implications (see Fig. 7.17 on the *right*), so the dashed edge starting at the node x_i will point to the node x_{i+1} in a gadget representing another clause (actually it points to a special node x_{i+1}^{IN} that serves as a connecting point: we will present the details in the next paragraph; the last x_i will point to x_l situated in the exclusive gadget clause for x, which we describe later). For convenience, we order the clauses from left to right, and name the variables x_i, y_i, and z_i with increasing i from the left to the right according to this order. Thus, every dashed edge connecting two different gadgets points rightwards when it is a forward **F**· edge, and leftwards when it is a backward **B**· edge.

For each implication clause $\mathcal{K} = (x_i \to x_{i+1})$, we already have the nodes representing the two variables x_i and x_{i+1} (lying on two separate solid paths belonging to their respective gadgets) and a dashed edge from the antecedent, x_i, to a new node x_{i+1}^{IN} placed below the consequent one. The gadget (Fig. 7.17 *right*) assures that if x_i is updated in the first round, then x_{i+1} must be updated as well, or there will be a cycle in the second round ($x_i \to x_{i+1}^{\text{IN}} \to x_{i+1} \to x_{i+1}^{BB} \to x_i^H \to x_i^{\text{IN}} \to x_i$): we draw a new node x_{i+1}^{BB} of type **BB** slightly above x_{i+1} (on its solid path) and a dashed edge pointing from it to another new helper node (to meet the in-degree constraint of the network update problem), x_i^H, slightly below x_i (in the figure we draw it below x_i^{IN} as well).

Then for every exclusive clause $\mathcal{K}_x = (\neg x_l \vee \neg \overline{x}_l)$ (shown in Fig. 7.18), we draw four solid paths. On the first, the **FF** node $\overline{x}_l$ is drawn and a dashed edge pointing from it to another helper node v_1 lying on the third solid path. Similarly, x_l on the second path points, with its forward dashed edge, towards v_3, which we place as the last of the four solid paths. Above v_1 and v_3 we draw another pair of **BB** nodes, v_2 and v_4 respectively. Then v_2 points back to the second solid path with its backward dashed edge, to another new node, x_l^H placed just below x_l on the first path. In the same manner, the backward edge starting at v_4 ends with $\overline{x}_l^H$ below $\overline{x}_l$. This way, updating both x_l and $\overline{x}_l$ in the first round will result in a cycle in the second round, since, as we know, all **BB**s *must* be updated in the second round. The cycle which can exist in the second round includes the following nodes $\overline{x}_l, v_1, v_2, x_l^H, x_l^{\text{IN}}, x_l, v_3, v_4, \overline{x}_l^H, \overline{x}_l^{\text{IN}}, \overline{x}_l$. x_l and $\overline{x}_l$ have been updated in the first round, the nodes v_2 and v_4 have been updated in the second round and the rest of the nodes in the cycle (the grey nodes) have not been updated yet. It will be later assured that they are of type **B**· and therefore cannot be updated in the first round, hence making a scenario possible where they are delayed until the end of the second round. Therefore an update of both x_l and $\overline{x}_l$ is not possible in the first round.

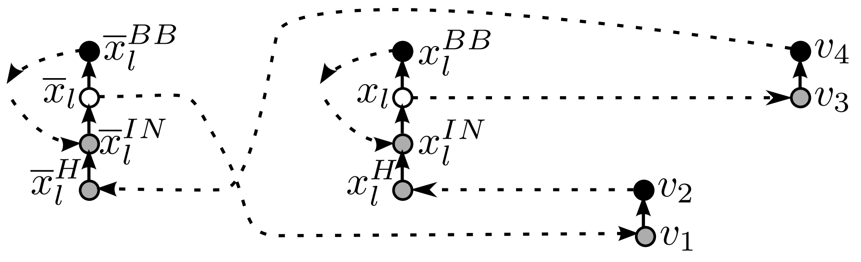

Figure 7.18: Gadget for exclusive clause. An update of either x_l or $\overline{x}_l$ prevents the other one from being updateable: the **BB** nodes v_2 and v_4 would form a cycle in the second round.

While the composition of gadgets described so far is not yet a proper instance of a network update problem, we can already make some observations about the graph.

Theorem 13. *If setting $V_{\mathcal{T}} \subset Var(\mathcal{C}')$ to $\mathtt{true}$ satisfies the formula, then there is no cycle ($\Rightarrow$). Moreover, a cycle-free update schedule gives us a satisfying variable assignment ($\Leftarrow$).*

Proof. We prove the two directions $\Rightarrow$ and $\Leftarrow$ in turn.

$\Rightarrow$: Cycles are composed of: dashed edges starting at $V_{\mathcal{T}}$ nodes, solid edges starting at any other nodes to get somewhere, and any edges starting at **BB** nodes to get back. We will show that by following an arbitrary path consisting only of the listed edge types, we will never return to the starting point of the path. If the path ever chooses to take an **FF** updated dashed edge (starting at x_i), it will need to continue with edges starting at x_{i+1} up to x_l (this is ensured by the implications), and there is no way back from there: it cannot constitute a cycle. Conversely, a path which does not take any **FF** dashed edges would not be able to jump from one of the solid, vertical paths to another one more to the right, so if it returns to the starting point, it must use nodes lying on one of the solid paths. At the same time, a cycle on one of the solid paths would mean that one of the clauses is not satisfied, which contradicts the definition of $V_{\mathcal{T}}$.

$\Leftarrow$: Clearly, the construction assures that if the formula $\mathcal{C}'$ is not satisfiable, when we have a selection of **FF** nodes which make the situation with *all* **BB** edges (which must be updated) acyclic then each clause must be true: it contains a true variable showing a path out of the cycle. ■

Connecting the Pieces

The presented gadgets leave us with a number of independent solid paths and many dashed edges starting at nodes of particular types (**FF** or **BB**). In order for the network to represent a valid problem instance, we need to connect the solid paths as well as the dashed paths. Our goal is to connect the solid path from left to right (and vertical lines are from bottom to top). The dashed path will be more complicated.

Let us first focus on connecting the *dashed* edges to a path. From the endpoint of each dashed edge, we will draw a backward dashed edge to a completely new node (one for each) placed far left from our solid paths. Hence, all nodes in R — the set of new nodes — will appear earlier in the concluding solid path: edges pointing to R are backward, edges pointing away from R forward. Then we connect all the resulting 2-length dashed paths (including the previously constructed dashed ones, and the new ones pointing to R), using forward dashed edges starting at the new nodes, as described in the following.

Some of the nodes in our gadgets were of type **BB** while the others were **FF**. Recall, that these type-properties are fairly local: we only need to look at the next node on the solid path and determine if it is preceding on the dashed path. To preserve the types of the nodes, we must therefore connect the 2-length paths in a correct order — first come the **FF** dashed edges, then the clause-related downward-pointing **BB** edges and in the end implication-related horizontal **BB** edges. In each of these groups the edges starting more to the left should precede those more to the right. Also – to ensure, that all the type assignment clause-related edges indeed start with a **BB** node – above each of those nodes $v_1^{\mathcal{C}}$, in their respective gadgets, we draw a new node $v_b^{\mathcal{C}}$. On

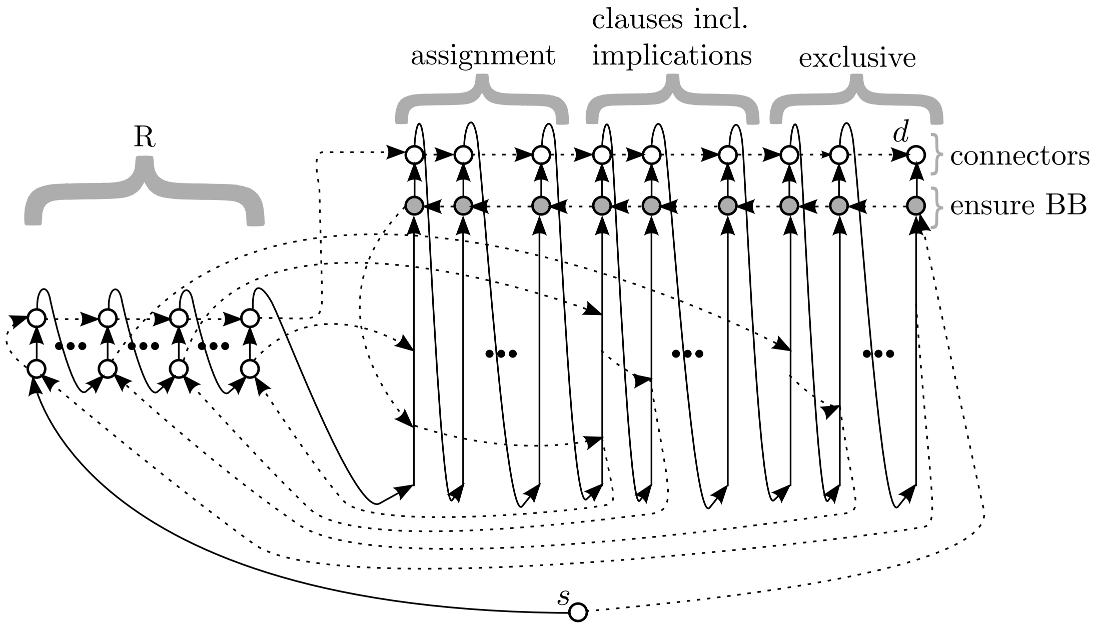

Figure 7.19: Overview of how the path is connected. The grey nodes are used to connect everything into one solid path. They also join the dashed path at the last nodes. This way, all nodes in R (*white cycles*) are of type **FF**.

each of the four solid paths used in the gadgets for the exclusive clauses $\mathcal{K}_x$, we do the same: we create nodes ${v_b^{\mathcal{K}_x}}_1, \ldots, {v_b^{\mathcal{K}_x}}_4$. Then we connect all the v_b's into a dashed path going from right to left. The path must be connected to the beginning of the dashed path we composed before, which will ensure the **BB** property of the previous nodes: the solid edge now points backwards relative to the dashed path. Each of the new $v_b^{\mathcal{C}}$ nodes will be of type **BF**. The nodes in R are ordered so that the dashed path ends at the leftmost node.

The nodes of R are positioned in a row, followed by our vertical solid paths. We draw a new node above each of them, connect it with a solid edge and connect the new node with what is next in the row, from the top of a vertical path to the bottom of the next one (Fig. 7.19). This way, we finally have one solid path. The new nodes are connected by a chain of forward dashed edges (so they can all be updated). In the end we add a starting node, which points with the solid edge to the leftmost R node, and with the dashed edge to the beginning of the dashed path which is the beginning of the path we constructed to ensure the **BB** properties (this point is **BF**).

It is important to note that in the last steps we have not jeopardized the reduction by introducing disconnections of the gadget-**BB** edges, nor have we created any loops that cannot be easily broken (by updating all the empty nodes in Fig. 7.19). Therefore, the possibility of making the second round cycle-free in our instance is still equivalent to the satisfiability of $\mathcal{C}'$, which makes the 3-round network update problem NP-hard.

7.4 Relaxed Loop-Free Updates Are Tractable

Given the potentially large number of rounds required to update a network in a strongly loop-free manner, we now propose to relax loop-freedom to only include *actually used paths*, between source and destination. We believe that this is an attractive alternative: although some unlucky packets currently on transit on an edge may end up in a (temporary) loop, we will never route any packets entering the network at the source into a loop. Moreover, as we will see, relaxing the loop-freedom is also attractive because it enables fast and computationally tractable updates. (Recall also the example in Fig. 7.2 which permitted a 3-round solution for RLF while SLF required $n-1$ rounds.) In particular, we will present a fast and elegant algorithm which

Algorithm 7: *Peacock*

```
Input: initial network $G_0$, set of to-be-updated nodes $U$
Output: (relaxed) loop-free schedule $(U_1, U_2, \dots, U_k)$
 1: $G \leftarrow G_0$, $t \leftarrow 0$, for all $t$: $U_t \leftarrow \varnothing$
 2: while ($G$ contains more than one node) do
 3:     $t++$
 4:     $X \leftarrow U \setminus U_{<t}$
 5:     if ($t$ odd) then
 6:         sort dashed forward edges in $out_2(X)$
 7:         for $u \in X$, starting with max forward distance do
 8:             if ($\nexists v \in U_t$ s.t. $(v < u < out_2(v)) \vee (v < out_2(u) < out_2(v))$) then
 9:                 add $u$ to $U_t$
10:     else
11:         add to $U_t$ all nodes not on the path from $s$ to $d$
12: return $(U_1, U_2, \dots, U_t)$
```

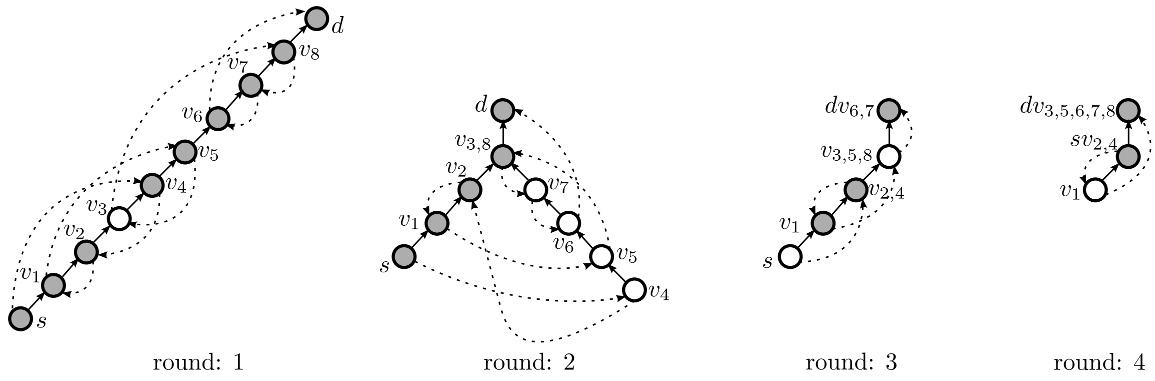

Figure 7.20: Example execution of *Peacock*. Updated nodes are shown in white. The initial network is a line (on the *left*). An update of the node with the largest distance v_3 and the merging of v_3 and v_8 leads to a tree shown for round 2. Here the nodes $v_4 - v_7$ can be updated since they are not on the $s - d$ path. This results in a line again, shown for round 3. In round 4, v_1 will be updated before the last node, which will be updated in round 5.

never requires more than $O(\log n)$ rounds: a potentially large gain given the $\Omega(n)$ lower bound for stronger models.

Before presenting the algorithm in detail, note that during its execution, our algorithm will repeatedly perform *node merging*: merge a node v with the node $out_2(v)$ after its update. The merging can also be done for several subsequent nodes in a single round if all of them are being update, or merging single nodes one by one in already merged nodes.

The proposed algorithm *Peacock*[1] is based on repeated node merging, and hence *tree shrinking*: starting from the line, it constructs various trees of decreasing sizes, until only a single node is left. At this point, the update is complete and the algorithm terminates. As we will see, *Peacock* manages to decrease the remaining network size by at least a constant factor, for each *pair of consecutive rounds*, resulting in the $O(\log n)$-round upper bound.

Concretely, *Peacock* toggles between two simple strategies:

1. **Shortcut:** In odd rounds (i.e., in the 1st, 3rd, etc. round), *Peacock* tries to reduce the distance between source s and destination d as much as possible, by updating a disjoint set of "far-reaching" (dashed)

[1]The name of the algorithm is due to its branch resp. "feather" spreading strategy.

forward edges: we define the *distance* of a dashed edge as the number of solid edges it skips on the current path from s to d. The idea is that by updating these far-reaching edges, we obtain a tree with many branches (of which only one contains the s-d path).

2. **Prune (and re-establish line):** In the even rounds (i.e., in the 2^{nd}, 4^{th}, etc. round), *Peacock* updates all nodes which are not on the current path from s to d. Since in the preceding odd round we shortened the length of the path from s to d, we can now update a significant number of nodes (namely a constant fraction of the still to-be-updated ones), and due to the subsequent merging operation, the resulting network size is significantly reduced. Intriguingly, the even round, after pruning and merging nodes, will always result in a simple line network again. Based on this line, we can easily determine the next set of far-reaching updatable edges again, enabling a subsequent "productive" even round.

Algorithm 7 gives the formal listing for *Peacock* and Fig. 7.20 illustrates an example. In the first round there is only one node (v_3) updated. *Peacock* is in the **Shortcut** phase and updates the "far reaching" edges. Once it adds node v_3 there is no other dashed forward edge remaining which is not interfering with the update of v_3. Hence *Peacock* switches to the **Prune** phase in round 2 and updates every node which is not on the $s-d$ path (v_4, v_5, v_6, v_7). *Peacock* then uses the **Shortcut** strategy again in round 3.

Theorem 14. Peacock *solves any problem instance in $O(\log n)$ rounds.*

Proof. We will make use of two helper lemmas, one targeting odd rounds (the extent to which the distance from s to d can be shortened) and one targeting even rounds (the number of nodes which can be pruned to produce a smaller resulting tree). We will see that after each pair of a consecutive odd and even round, only a constant fraction of nodes is left due to merging.

Lemma 7. *In each odd round,* Peacock *reduces the number of nodes on the solid path from s to d by $n_t/3$, where n_t is the number of nodes on the path.*

Proof. Peacock orders the nodes in decreasing order of distance, i.e., the number of solid edges they bridge. Including a node v (and its dashed edge), may block other nodes (resp. their intervals) from being scheduled in this round. However, due to the descending distance order, the set of blocked dashed edges span at most twice the distance from v to *out*$_2(v)$ on the current path: since we choose a maximal distance edge (say of distance x), edges entering or exiting the corresponding interval may block at most an additional distance of $2x$. Assuming that these distances cannot be covered by any other updates, *Peacock* loses at most twice the distance which it covered. This leaves, in the worst case, at most $2n_t/3$ nodes on the path from s to d. ■

Lemma 8. Peacock *can simultaneously update all nodes which are not on the path from s to d. The subsequent merge operation, re-establishes the line topology. .*

Proof. First, we observe that by updating these nodes, we cannot introduce any loop, since we do not touch any outgoing dashed edges. Dashed edges, at any time, must form a simple path. Each branch which is currently not on the s-d path will therefore point with at least one new rule to the s-d branch. All nodes of the branch can hence be merged with the respective nodes of the new rules on the s-d branch: a line topology. Also note that the source s does not necessarily have to be at the leaf of a tree. But also in this case, it is possible to update everything on the branch below s. Imagine a node u' which is not on the (solid $s-d$) path. Due to *node merging*, this node will be merged with $out(u')$, which itself is now either part of the $s-d$ path, or will be updated together with another node. Thus, we will successively merge nodes until a node (necessarily) lies on the $s-d$ path and will not be updated. This leads to a line with s as a leaf. ■

Lemma 7 shows that *Peacock* reduces the number of nodes on the s-d path by $n_r/3$ if the underlying network is a line. All of these nodes are not part of the s-d path in the next round, and on different branches. This shows that an update of these nodes is possible in even rounds without introducing a (relaxed) loop. Since, according to Lemma 8, an update of every node but those on the s-d path leads to a line again, we have shown that the number of remaining nodes is reduced by a third every second round. The number of rounds is hence logarithmic. ■

7.5 Related Work

Our work is motivated by the SDN paradigm, and especially its traffic engineering flexibilities and its support for a programmatic, dynamic, yet formally verifiable network management. [53] Indeed, a more flexible traffic engineering, that is, selection of forwarding policies, is considered one of the main motivations for SDN, and has been studied intensively over the last years. [45, 56] Our work is orthogonal to this line of research, in the sense that in our model, the policies are *given* and can be arbitrary.

The problem of updating [27, 65, 77, 82, 95], synthesizing [83] and checking [66] policies [88] as well as routes [28] has also been studied intensively. In their seminal work, Reitblatt et al. [95] initiated the study of network updates providing strong, per-packet consistency guarantees, and the authors also presented a 2-phase commit protocol. This protocol also forms the basis of the distributed control plane implementation in [27].

Mahajan and Wattenhofer [82] started investigating weaker transient consistency properties—in particular also (strong) loop-freedom—for destination-based routing policies. Mahajan and Wattenhofer proposed an algorithm to "greedily" select a maximum number of edges which can be used early during the policy installation process. Our work builds upon [82], but focuses on an alternative, round-based model to measure policy installation times, and also shows that a greedy strategy can lead to a large number of communication rounds. The measurement studies in [65] and [72] provide empirical evidence for the non-negligible time and high variance of switch updates, further motivating our work.

7.6 Summary

We show that dynamically updating a network is not a simple task. In fact, both objectives, trying to maximize the updates per round as well as trying to minimize the number of rounds turn out to be NP-hard. Nevertheless it is important to choose the right objective as we also find that trying to maximize the number of updates for SLF might unrevokably increase the number of rounds needed by up to a factor of $\Omega(n)$. However, we also describe fast (approximation-) algorithms for maximizing the number of updates per round for specific problem instances.

We introduce RLF as a more practical version of the SLF constraint and show that maximizing the number of updates for RLF can increase the number of rounds needed by up to a factor of $\Omega(\sqrt{n})$. Finally we introduce *Peacock*, which polynomialy solves any problem instance within $O(\log n)$ rounds.

We believe that our work opens interesting questions for future research. Most importantly, it would be interesting to derive $\omega(1)$-round lower bounds, or show that $O(1)$-round schedules for relaxed loop-free problems always exist. Our computational experiments (using mixed integer programs) indicate that larger problem instances require more rounds. So far, the worst problem instance (consisting of 1,000 nodes) we found requires seven rounds.

8 Waypoint Enforced Network Updates

Our results in Chapter 7 show that performing dynamic updates under the loop-freedom constraint is a complex problem. Unfortunately, loop freedom is not the only constraint, which needs to be taken into account. More and more network functionality is virtualized and employed anywhere in the network. We hence identify and study a new fundamental transient property, namely *Waypoint Enforcement* (WPE). WPE is an important property in today's increasingly virtualized networks where functionality is introduced also in the network core. For example, in security-critical environments (e.g., in a financial institution), it is required that packets traverse certain checkpoints, for instance, an access control function (implemented by e.g., a middlebox [104], an SDN match-action switch [73], or an NFV function [41]), before entering a protected network domain. In other words, to prevent a bad packet from entering a protected domain, not only the old policy π_1 as well as the new policy π_2 must ensure WPE, but also any other transient configuration or policy combination that may arise during the network update. So far, waypoints could only be enforced using the two-phase commit approach, which by definition implies that new links can never be used earlier [82].

Contribution

This section introduces an important new transient consistency property, namely Waypoint Enforcement (WPE), and shows that at the heart of the WPE property lie a number of interesting fundamental problems. In particular, we show that WPE may easily be violated if no care is taken. Motivated by this observation, we present an algorithm WAYUP that provably updates policies in a consistent manner, while minimizing the number of controller interactions.

We show that WPE cannot always be implemented in a wait-free manner, in the sense that the controller must rely on an upper bound estimation for the maximal packet latency in the network. Moreover, the transient Waypoint Enforcement WPE property may conflict with Loop-Freedom LF, in the sense that both properties may not be implementable simultaneously. Subsequently, we formally prove that that deciding whether a network update schedule satisfying both consistency properties, LF and WPE, exists, is NP-hard. We hence, present an optimal policy update algorithm OPTROUNDS, which provably requires the minimum number of communication rounds while ensuring both WPE and LF, whenever this is possible, to investigate the problem scale on small- to mid-sized scenarios.

To measure the "speed" of a network update algorithm, we again use the metric *round complexity*: the number of sequential controller interactions needed during an update. We believe that optimizing the round complexity is natural given the time it takes to update an individual OpenFlow switch today (see, e.g. [65]). Especially for scaling service chains on large NFV-enabled networks, as, e.g., envisioned by the UNIFY project [9], quickly

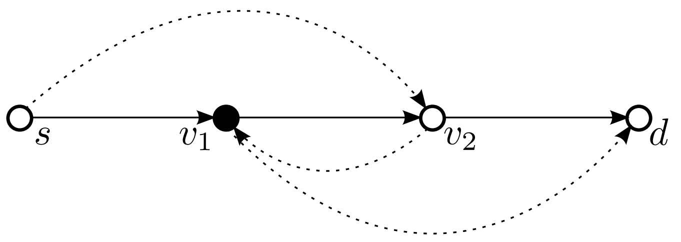

Figure 8.1: Updating all nodes in one round may violate WPE.

updating policies while guaranteeing correctness will be of importance [90]. Moreover, this section also shows that the optimization of existing metrics, like the number of currently updated links [82], may fail to install the policy entirely: a deadlock configuration may occur where the policy installation cannot be completed.

Example

In order to acquaint ourselves with the problem of fast and consistent network updates fulfilling Waypoint Enforcement (WPE) and Loop-Freedom (LF), in this section, we consider a simple example. If not otherwise mentioned, the results in this chapter hold true for both definitions of loop-freedom (RLF and SLF), and we hence, refer simply to LF.

We stick to the model introduced in Chapter 7, hence updating an old policy π_1 (*solid line*) to a new policy π_2 (*dashed line*). In Figure 8.1 an example including a waypoint is given. The old policy π_1 connects four nodes, from left to right (depicted as a *straight, solid line*, $s \to v_1 \to v_2 \to d$); the new policy π_2 is shown as a *dashed line*. The second node, v_1 (in *black*), represents the waypoint which needs to be enforced.

How does the introduction of a waypoint changes the strategy of updating the policy π_1 to π_2? The simple solution to update all nodes *concurrently* has been already ruled out as it does not enforce LF. Indeed it also may cause problems with a waypoint. For example, if s is updated before v_1 and v_2 are updated, a temporary forwarding path may emerge which violates WPE: packets originating at s will be sent to v_2 and from there to the destination d—the waypoint v_1 is *bypassed*.

The solution to perform the update in *two (communication) rounds* this time also does not hold: in the first round, only v_1 and v_2 are updated, and in a second round the controller also updates s. Note that this 2-round strategy indeed maintains the waypoint at any time during the policy update. However, the resulting solution may still be problematic, as it violates *loop-freedom*: if the update for node v_2 arrives before the update at node v_1, packets may be forwarded in a loop, from node v_1 to v_2 and back.

Both Waypoint Enforcement WPE as well as Loop-Freedom LF can be ensured (for this specific example) in a *three-round* update: in the first round, only v_1 is updated, in the next round v_2, and finally s.

We, in this chapter, are interested in dynamic consistent network updates which ensure not only LF but also WPE.

8.1 Ensuring Only Waypoint Enforcement

It turns out that the transient enforcement of a waypoint is non-trivial. We first show an interesting negative result: it is not possible to implement WPE in a "wait-free manner", in the following sense: a controller does not only need to wait until the switches have acknowledged the policy updates of round i before sending out the updates of round $i+1$, but the controller also needs some estimate of the maximal packet latency: if a packet can take an arbitrary amount of time to traverse the network, it is never safe to send out a policy update for certain scenarios. We are not aware of any other transient property for which such a negative result exists. For ease of presentation, we will use the notation $\pi_i^{<wp}$ to refer to the first part of the route given by

Algorithm 8: WAYUP

```
1 Input: old policy $\pi_1$, new policy $\pi_2$, threshold $\theta$
2 update nodes of $\pi_2$ which are not in $\pi_1$
3 update nodes of $\pi_1^{>wp}$ with backw. rules in $\pi_2^{<wp}$
4 update remaining nodes of $\pi_2^{<wp}$
5 wait $\theta$
6 update nodes of $\pi_2^{>wp}$
```

policy π_i, namely the sub-path from the source to the waypoint, and $\pi_i^{>wp}$ to refer to the second part from the waypoint to the destination.

Theorem 15. *In an asynchronous environment, a new policy can never be installed without risking the violation of WPE, if a node is part of $\pi_1^{<wp}$ and $\pi_2^{>wp}$.*

Proof. Consider the example in Figure 8.1 again, but imagine that the waypoint is on node v_2 and not on node v_1, and assume the following update strategy: in the first round, s and v_2 are updated, and in the second round, v_1. This strategy clearly ensures WPE, *if (but only if)* the updates of round 2 are sent out after packets forwarded according to the rules before round 1 have left the system. However, if packets can incur arbitrary delays, then there might always be packets left which are still traversing the old (*solid*) path from s to v_1. These packets have not been routed via the waypoint (v_2) so far but will be sent out to v_3 by v_1 in the new path, violating the WPE property. This problem also exists for any other update strategy. ■

Fortunately, in practice, packets do not incur arbitrary delays, and Theorem 15 may only be of theoretical interest: it is often safe to provide an update algorithm with some good upper bound θ on the maximal packet latency. The upper bound θ can be seen as a parameter to tune the safety margin: the higher θ, the higher the probability that any packet is actually waypoint enforced.

With these concepts in mind, we now describe our algorithm WAYUP which always ensures correct network updates, i.e., updates which consistently implement WPE if the maximal packet transmission time is bounded by θ. We define $v_1 \prec_{\pi_i} v_2$ to express that a node v_1 is visited before v_2 on π_i and accordingly, an update rule (v_2, v_1) with $v_1 \prec_{\pi_1} v_2$ is a *backward rule* (see also Chapter 6).

The round complexity of WAYUP is *four*: in the first round, all nodes are updated which were not part of the old policy π_1, and therefore do not have an impact on current packets (as shown in Figure 6.1). In the second round, each node behind the waypoint (i.e., $\pi_1^{>wp}$) which is part of $\pi_2^{<wp}$ and which has a backward rule, is updated. This allows us to update the remaining nodes from $\pi_2^{<wp}$ in the third round, since each packet which is sent "behind" the waypoint will eventually come back, according to the consistency properties of the new policy. After this round, the algorithm will wait θ time to ensure that no packet is on $\pi_1^{<wp}$ anymore. In the fourth round it is possible to update all nodes of $\pi_2^{>wp}$ in one round, because the update cannot interfere with $\pi_2^{<wp}$ anymore, and hence it cannot violate WPE.

Theorem 16. WAYUP *takes four rounds and guarantees the WPE property at any time.*

Proof. The round complexity follows from the algorithm definition. The transient consistency can be proved line-by-line: Line 2 of Algorithm 8 cannot violate WPE since no packet is crossing any of these nodes. Line 3 does not interfere with $\pi_1^{<wp}$ and therefore each packet will still be sent via $\pi_1^{<wp}$ towards the waypoint. As long as π_2 is consistent, any packet that reaches any node of $\pi_2^{<wp}$ will eventually reach the waypoint during Line 4, since all backward rules are already updated and no rule will bypass them. In Line 6, WPE is already guaranteed, since $\pi_2^{<wp}$ is already in place and θ time has elapsed. ■

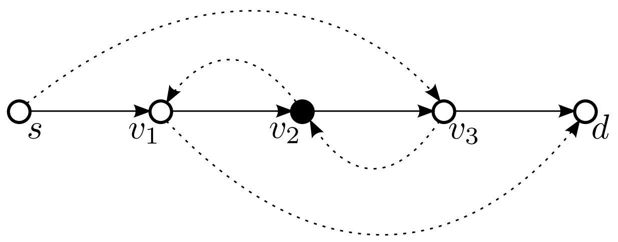

Figure 8.2: WPE and LF may conflict.

8.2 Incorporating Loop-freedom

The update strategy WAYUP presented in the previous section provably fulfills WPE at any time. However, it may violate LF. In this section we determine if WPE and LF can always coexist for dynamic updates and study the complexity on deciding if a scenario is solvable adhering to both properties.

8.2.1 Loop-freedom and Waypoint Enforcement May Conflict

This section also starts with a negative result: WPE and LF may conflict, i.e., it is sometimes impossible to simultaneously guarantee both properties.

Theorem 17. *WPE and LF may conflict.*

Proof. Consider the example depicted in Figure 8.2. Clearly, the source s can only be updated once v_3 is updated, otherwise packets will be sent to d directly, which violates WPE. An update of v_3 can only be scheduled after an update of v_2 without risking the violation of LF. However, v_2 needs to wait for v_1 to be updated for the same reasons. This leaves an update of v_1 as the last possibility, which however violates WPE again. Hence there is no update schedule which does not violate either WPE or LF. ■

Fortunately, in practice, such conflicts can be identified, and if they exist, can be resolved with other mechanisms (e.g., by sacrificing speed and using the PPC algorithm described in [95]). In the following, we will focus on algorithms which find efficient policy updates for scenarios where WPE and LF do not conflict. A naive approach to find such a consistent update may be to split the update into two distinct parts: the part *before* and the part *after* the waypoint, i.e., $\pi_2^{<wp}$ and $\pi_2^{>wp}$, and use a LF update algorithm on both parts (e.g., *Peacock*). Unfortunately, this approach can fail: only if $\pi_2^{>wp}$ has no overlaps with $\pi_1^{<wp}$ and vice versa, and if $\pi_2^{<wp}$ has no overlaps with $\pi_1^{>wp}$, it is safe to update both paths in parallel. This result even holds for a consecutive update of $\pi_2^{<wp}$ and $\pi_2^{>wp}$. The new policy shown in Figure 8.1 cannot be updated without inducing loops on $\pi_1^{>wp}$ (i.e., $s_2 \rightarrow s_3 \rightarrow s_2$) if $\pi_2^{<wp}$ is updated first; and a similar example can be constructed for a schedule where $\pi_2^{>wp}$ is updated first.

Comparing Objectives

We have already shown in Section 7.2 that a single greedy round in SLF may change the required number of rounds from $O(1)$ to $\Omega(n)$. Even worse with the addition of waypoints, a greedy algorithm may not only increase the number of rounds, but it may fail to find a valid solution entirely. The only possible updates for the scenario in Figure 8.3 are the updates s and v_1. Updating both of them leads to a situation where no more nodes can be updated, since they are either violating WPE (v_2 and v_3) or LF (v_4 and v_5). The only possible update schedule delays s and only updates v_1.

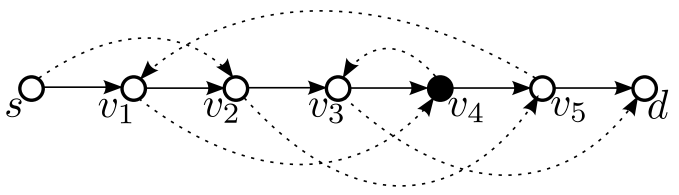

Figure 8.3: Scenario in which a single round of greedy updates renders the scenario to be unsolvable.

8.2.2 Determining if a Scenario is Solvable is NP-Hard

Given the knowledge that LF and WPE might conflict and even applying the wrong update strategy might leed to unsolvable scenarios, we are interested in identifying these scenarios. As shown in the following, this turns out to be an NP-hard problem. Hence, in this section we present how to construct a network update instance according to a 3-SAT formula which is updatable if and only if the 3-SAT formula is satisfiable. We refer to the 3-SAT formula as $\mathcal{C}$ and to network update instance as $G(\mathcal{C})$.

Notation

In our reduction we assume that each clause in 3-SAT has exactly 3 literals. We will denote the number of variables as k and the number of clauses as m. The variables will be denoted as $x_1, x_2, \ldots, x_k$ and the clauses as $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_m$. We will denote the total number of clauses with variable x_i as m_i, number of clauses with literal x_i as p_i and number of clauses with literal $\neg x_i$ as q_i. We will also denote the clauses with literal x_i as $P^i_1, P^i_2, \ldots, P^i_{p_i}$ and the clauses with literal $\neg x_i$ as $Q^i_1, Q^i_2, \ldots, Q^i_{p_i}$.

General Structure

In the constructed instance there will be a destination d, waypoint wp and auxiliary nodes u_1, u_2, u_3 and δ. For each variable and each clause in 3-SAT we create a gadget. In addition, for each clause we add three nodes d^i_1, d^i_2, d^i_3 and for each variable we add m nodes $r^1_j, \ldots, r^m_j$. The source of the path will be the first node in the first clause gadget. The order of gadgets and nodes is presented in Figure 8.4.

In a gadget for variable x_i there will be a set of nodes connected with clauses containing x_i. Updating one of those edges connecting a clause with the gadget will allow to untangle the corresponding clause (we will define untangling in Section 8.2.2; generally speaking in order to untangle a clause we need to update one of its edges which corresponds to satisfying it in 3-SAT formula). The variable gadgets will be constructed such that until all clauses are untangled only the edges corresponding to one literal (x_i or $\neg x_i$) can be updated. Therefore the constructed instance will be solvable only if we can untangle all clauses using one literal for each variable, which corresponds to satisfying 3-SAT formula.

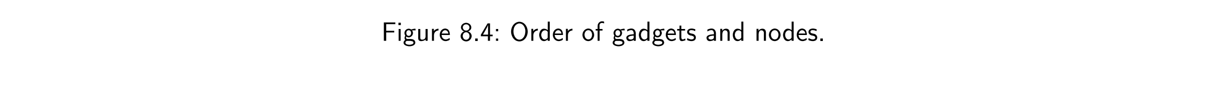

Figure 8.4: Order of gadgets and nodes.

Variable Gadgets

For each variable x_j we construct a gadget consisting of four nodes $v^j_1, v^j_2, v^j_3, v^j_4$. These nodes are connected with the edges (v^j_1, v^j_3) and (v^j_2, v^j_4) and there is an edge from v^j_4 to first node of next variable gadget, v^{j+1}_1 (and in case of the last variable gadget there is an edge from v^k_4 to δ). Note that these nodes v^j_4 and v^{j+1}_1

would be combined in our model, but it makes no difference for the problem if we keep both and the proof is easier to follow if we treat them seperately. In the gadget there are also nodes $y_1^j, y_2^j, \dots, y_{p_i}^j$ between v_1^j and v_2^j and nodes $z_1^j, z_2^j, \dots, z_{q_i}^j$ between v_3^j and v_4^j. Clauses $P_1^j, P_2^j, \dots, P_{p_i}^j$ will be connected to $y_1^j, y_2^j, \dots, y_{p_i}^j$, and updating an edge y_i^j will allow clause P_i^j to become untangled. Similarly clauses $Q_1^j, Q_2^j, \dots, Q_{q_i}^j$ will be connected to $z_1^j, z_2^j, \dots, z_{q_i}^j$. In turn nodes $y_1^j, y_2^j, \dots, y_{p_i}^j$ can be updated only if v_1^j is updated, and $z_1^j, z_2^j, \dots, z_{q_i}^j$ if v_2^j is updated and v_1^j is not updated (or all clauses are already untangled). That will allow us to conclude the value of x_j based on whether before all clauses become untangled v_1^j is updated or not. A construction of the gadget is presented in Figure 8.5.

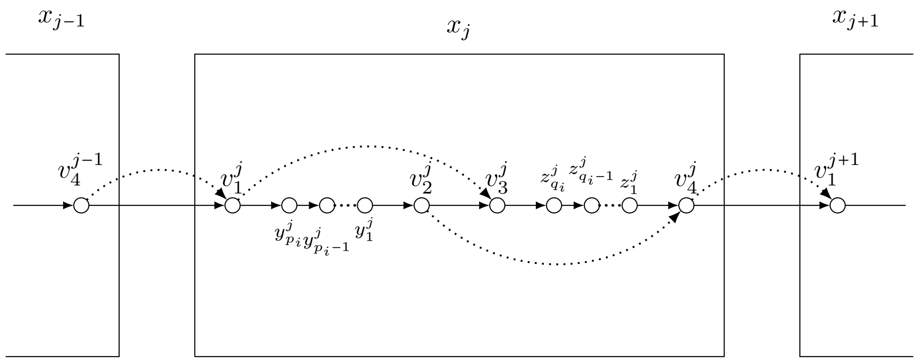

Figure 8.5: Construction of a variable gadget for x_j.

Clause Gadgets

For each clause $\mathcal{K}_i$ we construct a gadget consisting of nodes $c_1^i, c_2^i, \dots, c_6^i$. Also for each clause we add three nodes (outside of the gadget, close to d, see also Figure 8.4) d_1^i, d_2^i, d_3^i. For each $j \in \{1,2,3\}$ we add edges (c_j^i, d_j^i) and (d_j^i, c_{j+3}^i). The purpose of nodes d_1^i, d_2^i and d_3^i is to delay the update of nodes c_1^i, c_2^i and c_3^i until all the clauses are untangled. The construction of a clause gadget is shown in Figure 8.6.

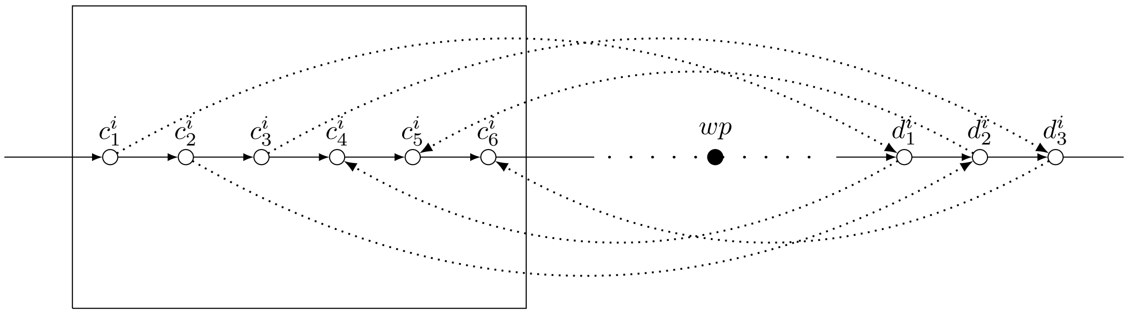

Figure 8.6: Construction of a clause gadget.

Connecting the Gadgets

Let us consider variable x_i. Let $\mathcal{K}_j = P_a^i$ be any clause containing literal x_i. Then we connect one of the nodes c_4^j, c_5^j, c_6^j to node y_a^i, and this node to respectively c_1^{j+1}, c_2^{j+1} or c_3^{j+1} (if $\mathcal{K}_j$ is the last clause than to u_1, u_2 or u_3 instead). Note, that the nodes $y_1^i, \dots, y_{p_i}^i$ are in reverse order than clauses $P_1^i, \dots, P_{p_i}^i$ (so the earliest clause is connected to the last node). We proceed similarly with clauses $Q_1^i, \dots, Q_{q_i}^i$ and nodes $z_1^i, \dots, z_{q_i}^i$.

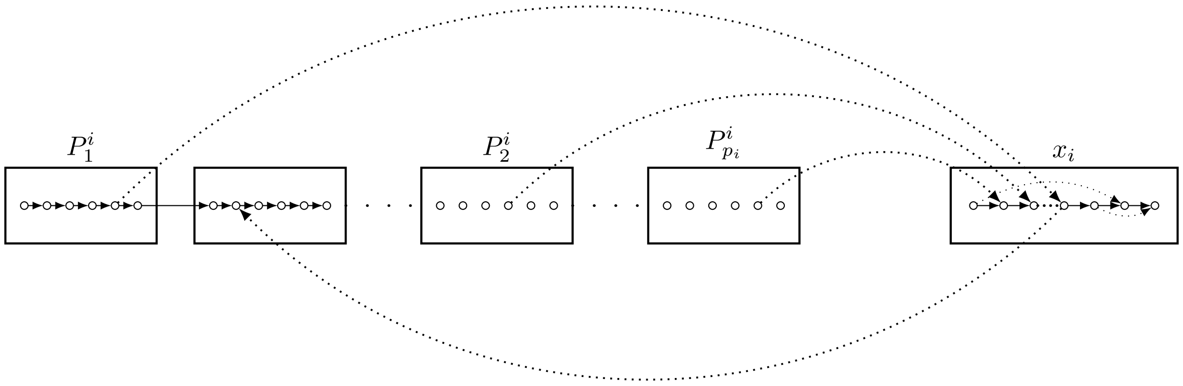

Figure 8.7: Edges to connect clauses.

Connecting the Whole Graph

In addition to the gadgets we need to connect the path to the destination d, the waypoint node wp and the three nodes u_1, u_2, u_3 which will be in the old policy just before the waypoint. We add edges from u_3 to wp, from wp to v_1^1, from u_1 to c_2^1 and from u_2 to c_3^1. After every i-th clause gadget we create $k+1$ nodes $r_1^i, r_2^i, \ldots, r_{k+1}^i$ in reverse order, i.e. r_{k+1}^i is the first node after the gadget and r_1^i is the last. For each variable x_i we create a path starting in v_3^i, then going through nodes $r_i^m, r_i^{m-1}, \ldots, r_i^1$ and ending in v_2^i. We also create a similar path starting in δ, then going through nodes $r_{k+1}^m, r_{k+1}^{m-1}, \ldots, r_{k+1}^1$ and ending in v_d^i. All these edges are shown in Figure 8.8.

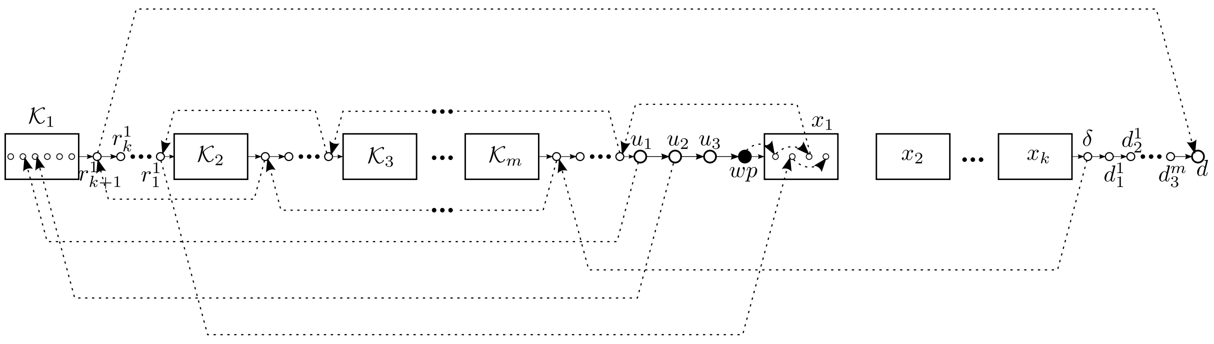

Figure 8.8: Connecting all paths.

Proof of Correctness

In this section we prove, that the reduction is correct. We say that a clause (or clause gadget) is untangled if at least one of the nodes c_4^i, c_5^i or c_6^i is updated. We say that a clause is tangled if it is not untangled.

Theorem 18. *If $\mathcal{C}$ is satisfiable then there is a schedule for $G(\mathcal{C})$ which satisfies SLF and WPE.*

Proof. Let σ be an assignment that satisfies $\mathcal{C}$. Then based on σ we show how to update all nodes in $G(\mathcal{C})$ without violating SLF or WPE. The nodes will be updated according to the following round schedule:

1. For each variable x_i we update v_2^i. Also if $\sigma(x_i) = 1$ we update v_1^i (which makes the update of v_2^i irrelevant as it bypasses v_2^i).
2. For each variable x_i we update either nodes $y_1^i, \dots, y_{p_i}^i$, if $\sigma(x_i) = 1$, or nodes $z_1^i, \dots, z_{q_i}^i$ otherwise.
3. Since for each clause $\mathcal{K}_j$ there is at least one literal that satisfies it, we update one of nodes c_4^j, c_5^j, c_6^j which is connected to that literal. The path after these updates is shown on Figure 8.9.
4. We update nodes r_j^i for all i, j. This can be done, since every clause has at least one outgoing edge and every r_j^i edge has a clause in between.
5. We update nodes v_3^i, for all i, and node δ, which connects the path updated in round 4 with the reachable parts behind the waypoint.
6. We update those nodes v_1^i that were not updated earlier, as the path starting at v_3^i is now loop-free.
7. We update those nodes y_j^i and z_j^i that were not updated earlier.
8. We update those nodes c_4^j, c_5^j and c_6^j that were not updated earlier.
9. We update nodes d_1^j, d_2^j, d_3^j, for all j.
10. We update nodes c_1^j, c_2^j, c_3^j, for all j.
11. We update nodes u_1, u_2, u_3 and wp.

Note that none of those updates violates WPE or SLF.

■

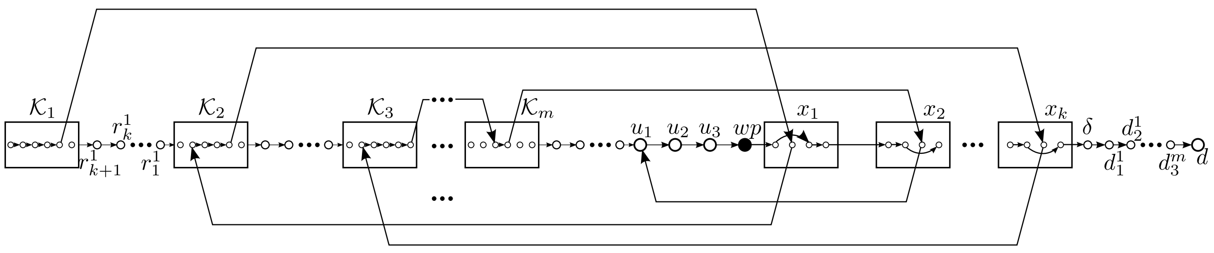

Figure 8.9: The path after three rounds of updating according to schedule in proof of Theorem 18.

Theorem 19. *If there is a schedule for $G(\mathcal{C})$ which satisfies RLF and WPE then $\mathcal{C}$ is satisfiable.*

We will start by proving the following lemma:

Lemma 9. *In any correct order of updating edges, as long as some clause gadgets remain tangled, the following conditions hold:*

1. *A node y_j^i can be updated only if node v_1^i is updated. A node z_j^i can be updated only if node v_2^i is updated. Nodes z_j^i and v_1^i cannot be both updated.*
2. *A node r_j^i, for any $i \in \{1, \dots, m\}$ and $j \in \{1, \dots, k+1\}$, can be updated only if i-th clause gadget is untangled.*
3. *A node c_j^i, for $j \in \{4, 5, 6\}$ can be updated only if its successor is already updated or if there is $h \in \{4, 5, 6\}$ such that $h < j$ and c_h^i is already updated.*
4. *A node v_3^i, for any i, can be updated if for all $j \in \{0, 1, \dots, m\}$ r_i^j is updated or if v_2^i is updated, but v_1^i is not. The same applies to node δ.*
5. *Nodes d_1^i and d_2^i and d_3^i, for any i, cannot be updated.*

6. *Node c_3^i cannot be updated. Node c_2^i can be updated only if c_6^{i-1} and its successor are updated, c_5^{i-1} or its successor are not updated and c_4^{i-1} or its successor are not updated. c_1^i can be updated only if c_4^{i-1} or its successor are not updated and either c_6^{i-1} and its successor or c_5^{i-1} and its successor are updated.*

Before proving the lemma, let us make some observations about what these conditions mean in terms of path traversed by packets. The Conditions 1 and 4 guarantee, that if a packet is in v_1^i or v_2^i, for some i, then it will be forwarded to node δ without going through wp. That is because it uses edges from v_1^j and v_2^j to bypass any backward edges. Then the Condition 5 guarantees that it goes from δ to d without passing through waypoint.

The Conditions 2, 3 and 6 guarantee that a packet will traverse from the source through all the clauses until it reaches the waypoint. That is because for each clause if it is untangled, then the packet will be forwarded from some c_j^i to y_l^a, and then, as y_l^a must have been updated before c_j^i, it goes back to c_{j-3}^{i+1}. On the other hand if the clause is tangled, the packet will go through $r_{k+1}^i, r_k^i, \dots, r_0^i$ (none of them is updated, since the clause is tangled) to c_1^{i+1}.

The Conditions 2 and 4 guarantee that as long as not all clauses are untangled, δ cannot be updated and v_3^i can be updated only if the path from source to destination does not go through that node.

Let us also notice that if Conditions 5 and 6 hold, then a packet can enter a clause gadget only through nodes c_1^i, c_2^i and c_3^i, and it is afterwards forwarded to node c_4^i. Therefore if it is enough to show that a packet enters a clause gadget twice to prove loop freedom violation.

Proof. Let us take any order of updating edges, and consider the first update that violates one of the conditions. If we update any node other than v_1^i at most one condition is violated. So firstly let's assume that only one condition is violated and consider the cases for which condition it is.

1. Let us assume that y_j^i is updated, but v_1^i is not. Then the packet goes through all clauses, and then through all previous variable gadgets. Upon entering the gadget for x_i it goes through an edge from y_j^i to $\mathcal{K}_i$ and therefore violates loop freedom. The case when z_j^i is updated is similar.

2. Let us assume that r_j^i is updated, but i-th clause is tangled. Then the packet goes through all clause gadgets up to $\mathcal{K}_i$, then it is forwarded to r_j^i. Then there are two possibilities. Firstly it may be forwarded back to the r_l^1, for $l \le j$, and from there to the gadget for x_l. But because Conditions 1, 4 and 5 are satisfied it would go the end without passing through waypoint. The other case is that it is forwarded back to r_l^a, for some $a < i$ and $l \le j$, and then reenters some clause gadget, which would violate loop freedom.

3. Let us assume that c_j^i is updated but its successor y_a^l and c_h^i, for all $h \in \{4,5,6\}$ such that $h < j$, are not. Then the packet traverses through c_j^i without passing through the waypoint, and then it goes to y_a^l. Then it may either be forwarded to v_2^l, which means that it would be forwarded to d without passing through the waypoint, because Conditions 1, 4 and 5 are satisfied, or it goes from some node y_g^l to gadget $\mathcal{K}_f$. But then $f \le i$, because of the order of nodes $y_{p_l}^l, \dots, y_1^l$, so it would go to a gadget that was already visited, and therefore violate loop freedom. The case when successor of c_j^i is z_a^l, for some l, a is similar.

4. Let us assume that v_3^i is updated, but there is some r_i^j which is not updated and either v_1^i is updated or v_2^i is not. Then the packet, after going through waypoint, reaches the gadget for x_i. Then, because v_1^i is updated or v_2^i is not, it is forwarded to v_3^i. From there it traverses through some backward edges, before it enters some clause gadget (it cannot take backward edges until it goes to r_a^1 and then go forward to some variable gadget, because Condition 2 holds and not all clauses are untangled). Since all clause gadgets were already visited, it violates loop freedom.

5. If d_j^i is updated, then the packet traverses through all clause gadgets and variable gadgets until it reaches d_j^i. From there it goes back to c_{j+3}^i. Then there are two possibilities: if it will be forwarded to the next clause gadget it will violate loop freedom, because all clauses were already visited. Otherwise, if c_{j+3}^i is updated, the packet is forwarded to some node y_a^l (or z_a^l). From there it can either be forwarded to some clause gadget or to x_{l+1}. In both cases it violates loop freedom.

6. The condition guarantees that if the packet traverses through c_j^i, for $j \in \{1, 2, 3\}$, then this node cannot be updated. Otherwise the packet after going to $\mathcal{K}_i$ (without going through the waypoint) would go to d_j^i and from there to the destination.

Finally let us consider what happens when we update v_1^i and it violates Conditions 1 and 4. Then the case is similar to violating only Condition 4, that is the packet traverses through all the clauses to the waypoint, and from there to v_1^i and next to v_3^i. From there it goes through some backward edges and reenter some clause, which violates loop freedom. ■

Now we are ready to prove Theorem 19.

Proof. Let us assume that there is a schedule for $G(\mathcal{C})$. Then let us look at the update which untangles the last clause (that is before this update there was an tangled clause, and after this update all clauses are untangled). Then Condition 1 guarantees, that for each variable there is no node corresponding to positive literal (node y_a^i) and a node corresponding to negative literal (node z_l^i) that are both updated. That is because updating node y_a^i requires that v_1^i is updated, whereas updating node z_l^i requires that v_1^i is not updated. Therefore in the assignment of variables in $\mathcal{C}$ we set x_i to 1, if at least one of nodes y_a^i, for any a, is updated, or to 0 otherwise. Then because all clauses are untangled, and untangling an clause requires that at least one literal has value 1, this assignment satisfies all formulas in $\mathcal{C}$. ■

Theorem 20. *There is a schedule for $G(\mathcal{C})$ satisfying RLF and WPE iff $\mathcal{C}$ is satisfiable and iff there is a schedule for $G(\mathcal{C})$ satisfying SLF and WPE.*

Proof. We have shown that existence of a schedule satisfying RLF and WPE implies that $\mathcal{C}$ is satisfiable. We have also shown that if $\mathcal{C}$ is satisfiable then there is a schedule satisfying SLF and WPE. Because a schedule satisfying SLF and WPE is also a schedule satisfying RLF and WPE, therefore these three statements are equivalent. ■

8.3 Exact Algorithm

Despite the NP-hardness proven in Section 8.2.2, we are interested in which order of magnitude these scenarios occur. Hence, in the following, we present the Mixed-Integer Program (MIP) OptRounds, which generates an update scheme requiring the minimal number of rounds. OptRounds cannot only generate optimal schedules but also determine if a scenario is solvable. We implement OptRounds to work with the RLF model.

According to the line representation presented in Chapter 6, policies π_1 and π_2 are described as (simple) paths E_{π_1} and E_{π_2} on the common set of nodes V. Both E_{π_1} and E_{π_2} connect the start node $s \in V$ to the target node $t \in V$.

As the task of the MIP 1 is to find the minimal number of rounds, we generally allow for $|V| - 1$ many rounds, denoted as $\mathcal{R} = \{1, \dots, |V| - 1\}$. We use binary variables $x_v^r \in \{0, 1\}$ to indicate whether the forwarding policy of node $v \in V$ is updated in round $r \in \mathcal{R}$ or not. Constraint 2 enforces the switching policy of each node to be changed in exactly one of the rounds. The objective to minimize the number of rounds is realized by minimizing $R \geq 0$ which is lower bounded by all the rounds in which an update is performed (see Constraint 1).

Given the assignment of node updates to rounds, the Constraints 3 and 4 set variables $y_e^r \in [0, 1]$ accordingly to indicate whether the edge $e \in E_{\pi_1} \cup E_{\pi_2}$ is contained after the successful execution of updates in round $r \in \mathcal{R}$. In the following we show how to enforce both the LF and the WPE properties.

Enforcing LF To enforce the LF (more specific: RLF in this case) property, we need to guarantee transient states between rounds to be loop-free. To this end, we first define variables $a_v^r \in \{0,1\}$ to indicate whether a node $v \in V$ may be reachable or *accessible* from the start $s \in V$ under any order of updates between rounds $r-1$ and r. The variables are set to 1 if, and only if, there exists a (simple) path from s towards $v \in V$ using edges of either the previous round or the current round (see Constraints 5 - 7). Similarly, and based on this reachability information, the variables $y_{u,v}^{r-1 \vee r} \in \{0,1\}$ are set to 1 if the edge $(u,v) \in E$ may be used in transient states, namely if the edge existed in round $r-1$ or r and u could be reached (see Constraints 8 and 9). Lastly, to ensure that a flow cannot be forced onto loops, we employ well-known Miller-Tucker-Zemlin constraints (see 10) with corresponding leveling variables $l_v^r \in [0, |V|-1]$: if traffic may be sent along edge $(u,v) \in E$, i.e., if $y_{u,v}^{r-1 \vee r} = 1$ holds, $l_v^r \geq l_u^r + 1$ is enforced, thereby not allowing for cyclic dependencies.

Enforcing WPE For enforcing WPE a similar reachability construction is employed (cf. Constraints 5 - 7). We define variables $\overline{a}_v^r \in \{0,1\}$ indicating whether node $v \in V$ can be reached from the start without passing the waypoint. To this end, we introduce the set of all edges $E_{\overline{\mathsf{WP}}} \subset E$ not incident to the waypoint and propagate reachability information only along these edges (see Constraints 11-13). Lastly, Constraint 14 ensures that no packet can arrive at t without passing the waypoint.

Mixed-Integer Program 1: Optimal Rounds

$$
\begin{array}{rlr}
\min R & & \text{(Obj)}\\
R \geq r \cdot x_v^r & r \in \mathcal{R}, v \in V & (1)\\
1 = \sum_{r \in \mathcal{R}} x_v^r & v \in V & (2)\\
y_{u,v}^r = 1 - \sum_{r' \leq r} x_u^r & r \in \mathcal{R}, (u,v) \in E_{\pi_1} & (3)\\
y_{u,v}^r = \sum_{r' \leq r} x_u^r & r \in \mathcal{R}, (u,v) \in E_{\pi_2} & (4)\\
a_s^r = 1 & r \in \mathcal{R} & (5)\\
a_v^r \geq a_u^r + y_{u,v}^{r-1} - 1 & r \in \mathcal{R}, (u,v) \in E & (6)\\
a_v^r \geq a_u^r + y_{u,v}^{r} - 1 & r \in \mathcal{R}, (u,v) \in E & (7)\\
y_{u,v}^{r-1 \vee r} \geq a_u^r + y_{u,v}^{r-1} - 1 & r \in \mathcal{R}, (u,v) \in E & (8)\\
y_{u,v}^{r-1 \vee r} \geq a_u^r + y_{u,v}^{r} - 1 & r \in \mathcal{R}, (u,v) \in E & (9)\\
y_{u,v}^{r-1 \vee r} \leq \dfrac{l_v^r - l_u^r - 1}{|V|-1} + 1 & r \in \mathcal{R}, (u,v) \in E & (10)\\
\overline{a}_s^r = 1 & r \in \mathcal{R} & (11)\\
\overline{a}_v^r \geq \overline{a}_u^r + y_{u,v}^{r-1} - 1 & r \in \mathcal{R}, (u,v) \in E_{\overline{\mathsf{WP}}} & (12)\\
\overline{a}_v^r \geq \overline{a}_u^r + y_{u,v}^{r} - 1 & r \in \mathcal{R}, (u,v) \in E_{\overline{\mathsf{WP}}} & (13)\\
\overline{a}_t^r = 0 & r \in \mathcal{R} & (14)
\end{array}
$$

8.4 Computational Results

In our computational evaluation, we are interested in the number of scenarios in which no solution for an update schedule can be found. This is either due to conflicting WPE and LF (see Theorem 17) or due to the chosen objective (as shown in Section 8.2). Figure 8.10 shows the percentage of solvable scenarios as a function of the problem size in terms of nodes. Every scenario which could be solved for objective O_1 (max link) could also be solved for objective O_2 (min rounds) via OptRounds. OptRounds also finds additional solutions in up to 13% of all problem instances for problem sizes of up to 25 nodes. The number of additional solutions is increasing due to a smaller probability for (objective based) deadlocks in smaller instances until 25

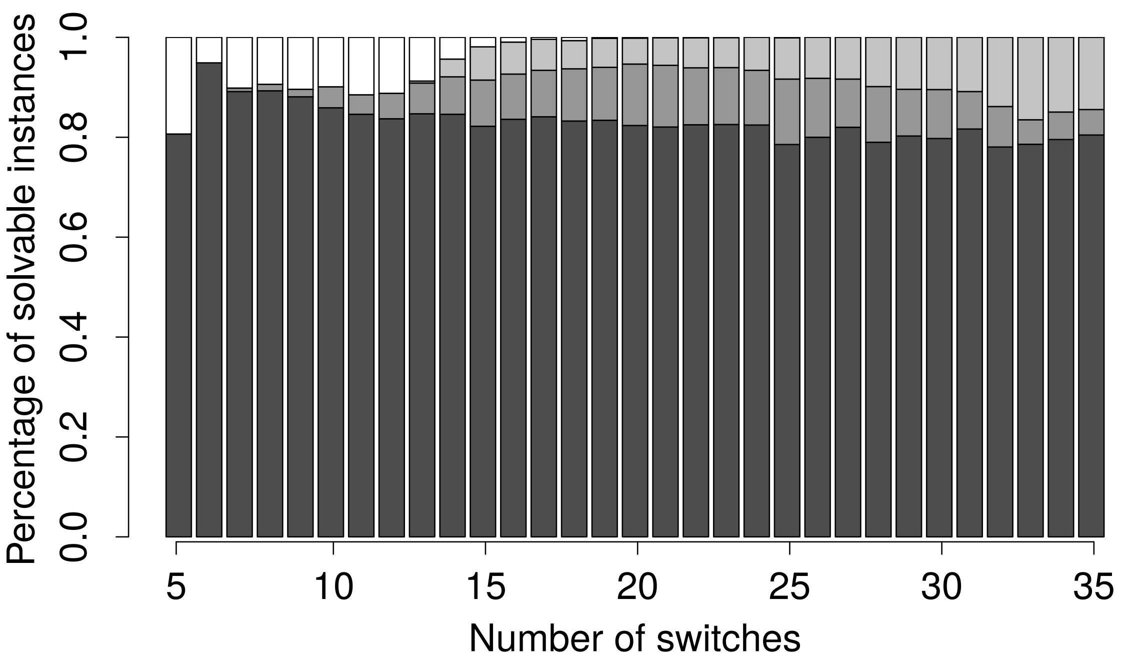

Figure 8.10: Percentage of solvable scenarios per number of nodes (1000 scenarios per size). The bars indicate from dark to bright: (1) Solvable regarding O_1; (2) additionally solvable with O_2 within 600 seconds; (3) not classified after 600 seconds; (4) not solvable. (O_1 - max updates; O_2 - min rounds).

nodes. The decrease for larger instances is a consequence of the capped runtime and the increased problem size which leads to a larger fraction of unclassified instances.

A similar trend can be observed for the total amount of solvable instances which is slightly increasing with the scenario size in the beginning (starting at the 7 nodes scenario). The percentage of solvable instances starts at roughly 90% (except for the scenario with 5 nodes), and increases towards -95% for scenarios with $14-25$ nodes, before it decreases againg due to the larger fraction of unclassified instances.

Whenever OptRounds terminated within 600 seconds, its median runtime was less than 1 second for networks with less than 15 nodes. The median runtime for solvable scenarios increases roughly linearly in the number of nodes (90 seconds for 35 nodes), and stayed below 1 second for not solvable scenarios.

The simulations show that there is a significant fraction of unsolvable scenarios until roughly 13 nodes, where the number of unclassified scenarios increases. A possibility to solve these scenarios without violating WPE and LF is by introducing additional edges,so-called *helper rules* [82]. However these can only be utilized when an independent path exists, i.e., a path which does not interleave with the new and the old path from the source to the waypoint.

8.5 Summary

As networks are becoming more and more virtualized and software-defined, new functionality is introduced also in the network core. The virtualization flexibilities raise the question of how to ensure that packets are always correctly forwarded through the corresponding network functions or waypoints.

This chapter shows both the limitations as well as the opportunities of fast policy updates ensuring waypoints. In particular, we introduce the transient consistency property waypoint enforcement WPE and show that WPE may easily be violated if schedules simply focus on loop-free update schedules. We provide a first algorithm WayUp which ensures WPE and show that WPE cannot always guaranteed in a wait-free manner. Moreover, we show that WPE can conflict with other consistency properties such as loop-freedom LF and that it is NP-hard to determine if a scenario is solvable for both properties. We find that the choice of the update strategy is important as well, as the wrong strategy ends up in deadlocks in certain scenarios even though there exists a solution. Hence, we present an algorithm which finds all potential conflicts (for small- to mid-sized problem spaces), and computes provably fast updates whenever this is possible.

9 Conclusion and Outlook

The Internet suffers from *ossification* [13]. While there is innovation on the application layer and on the lower layers, network innovation on the IP layer is difficult. On this "inter-"network layer all provider and stakeholder need to agree on changes. An appealing approach to enable innovation in the network core is the network virtualization paradigm. Network virtualization envisions a world where physical networks can be arbitrarily sliced into virtual networks running their own protocols (optimized toward their specific purpose) and providing performance isolation guarantees. This also allows for simpler service deployment and reliability. Multiple VNets are embedded on a shared physical infrastructure, which promises efficient resource utilization.

However, network virtualization also comes with challenges. Providing network performance guarantees over a shared substrate is non-trivial and exploiting the algorithmic flexibilities is hard. More fundamentally, it is not even clear what should be the interface between tenant and provider and how VNets should be specified. The embedding problem can be seen as a multi-dimensional packing problem (across multiple resources), and the resource allocation over time can be seen as a multi-dimensional parking permit problem. This also comes with economical implications, where "larger contracts have discounts".

Thus, network virtualization challenges not only concern embedding and admission control over time, but also consistent dynamic management over time. VNets require a smart hypervisor or control plane, which manages the resources and allows for adjustments: since the substrate is shared, it is non-trivial to ensure a correct operation while providing transient consistency guarantees such as drop-freedom or loop-freedom.

9.1 Summary

This work addresses two important challenges of network virtualization. The first part studies the economic implications of virtual networks. The second part studies algorithms to efficiently manage the dynamic operations on the network control plane.

VNet Market: In this thesis, we argue that a VNet is often more than "just a graph": especially in wide-area settings, geographic and temporal aspects matter, and resources are heterogeneous. Thus, VNets can be specified in different details and these differences are enhanced, as customers technical backgrounds differ. Hence, the provider has different degrees of flexibility embedding the VNet, which impacts the embedding cost. In particular, there is an interesting tradeoff between specifying requirements and flexibility and hence utilization of embedding.

To study this tradeoff, we define the *specificity* of a VNet. It characterizes the amount of choices a provider has in embedding this VNet. The impact of the specificity on the embedding cost is defined by the *Price of Specificity (PoS)*. The PoS is the ratio of the embedding cost of a VNet compared to the embedding cost of an unspecified VNet with the same topology. To avoid artifacts from suboptimal embeddings, we use a mixed integer program to compute optimal embeddings. We find that the PoS especially increases for highly specified VNets. This increase is not given in scenarios where the specificity allows the embedding on roughly 50% of the physical resources.

To shed light on the economic implications, we consider datacenters, a big market today. We study specifications in terms of VNet deadlines in a distributed datacenter market. We find that the social welfare is increased in scenarios with heterogeneous flexibilities. Nevertheless, customers with strict deadlines might suffer. Based on our previous observations that different specifications impact the embedding cost, we propose a *demand-specific pricing scheme (DSP)*. DSP is designed for the *virtual cluster (VC)* abstraction and allows VCs to be freely specified in their size and bandwidth requirements. Accordingly, we present an embedding algorithm *Tetris*, which takes the heterogeneous VC requests into account. Our evaluation shows that both DSP and Tetris can lead to higher resource utilization.

Pricing schemes such as DSP potentially introduce discounts. Buying more resources for larger amount of time yield further discounts and hence, a resource broker benefits in particular. We study resource buying strategies for a broker in scenarios with unknown demand. This problem is a multi-dimensional variant to the classical online parking permit problem. We present the deterministic online algorithm On2D and prove the asymptotical optimality of the algorithm.

Network Updates: Embedding multiple VNets on top of the physical infrastructure increases the load on the control plane. Especially long lived VNets increase the difficulty of managing and updating networks, as changes occur due to, e.g., changed specifications or load balancing. Misconfiguration of network devices can, in the worst case, shutdown the network. Thus, in the second part of the thesis, we study efficient algorithms to update the network consistently. A network update consists of a policy update, which involves the change of subsequent match-action rules on several forwarding devices.

We study a dynamic network update approach, which divides the update into rounds where a subset of devices is updated. This is a comparatively lightweight approach, as it is independent of tags. Packets are forwarded according to a mix of old and new rules. The challenge of any update algorithm is to find schedules for network devices, which keep the network consistent, i.e., adhering to properties such as drop-freedom or loop-freedom at any time during the update. We observe that the problem can be simplified and introduce a practically motivated variant of (strong) loop-freedom (SLF)—relaxed loop-freedom (RLF). We evaluate dynamic updates regarding both variants based on two different objectives: 1. maximizing the number of updates per round and 2. minimizing the number of rounds.

We prove that it is NP-hard to find an update schedule, which maximizes the number of updates per round. This result holds true for both SLF and RLF. On the positive side, we identify a class of network update problems that allow for optimal or almost optimal polynomial-time algorithms. However, the maximization strategy can increase the overall update time in terms of rounds by up to a factor of $\Omega(n)$. We find that minimizing the number of rounds is a complex problem as well. We prove that it is NP-hard to determine if an instance is solvable within three rounds for SLF. For RLF, we present *Peacock*, which computes update schedules providing $O(\log(n))$ round schedules in polynomial time.

Today's increasingly virtualized networks allow for flexible deployment of virtualized middleboxes such as firewalls in the network. Hence, we introduce a transient consistency property: waypoint enforcement (WPE). We provide an algorithm WayUp, which updates any scenario in a WPE-consistent manner. We show that WPE can conflict with loop-freedom and prove that it is NP-hard to determine if a scenario is solvable adhering to both properties. We provide a mixed integer program OptRounds to investigate on the scale of this conflict in small- to middle-sized scenarios.

9.2 Future Work

We identify the following directions for future work.

Pricing WAN Scenario: The thesis provides insights on the impact of a VNets specificity on its embedding cost. Furthermore, it provides a pricing scheme for the datacenter scenario and buying strategies for brokers. An interesting open question is how to price VNets in a WAN scenario. Given geographical constraints of customers, the network in a WAN scenario often belongs to different providers. To find a pricing is challenging, as it requires cooperation between the involved providers.

Embedding heterogeneous VNets: Embedding algorithms for datacenters such as [18, 98] often minimize the resource costs. Our work on specificities has shown that embedding heterogeneous VNets within the datacenter might benefit from other objectives as well. Heterogeneous requests might lead to fragmentation within the datacenter, and, hence wasted resources.

Multiple Policy Updates - Reduce Messages: Our approach for updates in networks handles each policy individually. In larger networks, several policy update requests can arrive concurrently. To reduce the control load, it is worthwhile to compute joint concurrent update schedules. Hence, a network device receives one instead of several update messages. First insights of this problem show that the number of messages cannot always be reduced to one per device. We found that it is NP-hard to minimize the number of messages. Thus, there is a need for efficient approximations to reduce the control plane load.

Multiple Policy Updates - Bandwidth Limitations: Dionysus [65] evaluates congestion free network updates of multiple policies or flows. The authors assume policy updates similar to the two-phase commit approach, treating a policy as either unchanged or completely updated. They find that certain scenarios are unsolvable without causing congestion. They propose to instead reduce the rate of the corresponding flows of the policies, to create a solution. The dynamic update approach used in this work might find other solutions. It works on a per switch granularity which might allow a policy update without rate limiting any flows.

Experimental Performance Evaluation: The dynamic update approach arguably provides several benefits compared to the two-phase commit approach. It is independent of tags and has faster first effects on the network. This work provides several theoretical results in terms of complexity and update speed for dynamic network updates. Yet, large-scale experiments comparing the two-phase commit approach with the dynamic approach appear interesting.

Acknowledgements

Looking back on the last four years, I can honestly say that it was a great time. Even during some work intensive deadline times, I was always enjoying the work due to my fabulous colleagues. I will not find the place to acknowledge every single person but I always felt welcome in every room of our office!

This is, of course, thanks to my advisor Anja, who was able to form this outstanding group. She gave me the chance to be part of that team and to find my place in the research community. I am grateful for the amount of freedom she gave me in terms of research while still giving me valuable feedback and guidance. My co-supervisor Stefan was also an integral part, as he was never shy of discussing the progress. I am still impressed by his ability to have multiple meetings concurrently in his calendar and sometimes even managing to attend all of them. He is a large reason that I was able to finish this thesis and it was always fun to work with him.

In my first days in the group, I was lucky to have Nadi and Oliver as my direct neighbors, who helped me to understand the research world and made my first steps very convenient, even though I inherited their teaching organization duties. These duties are now being handed over to Niklas and Damien, whose unbiased view towards research and fresh enthusiasm are fascinating. Gregor showed me, besides guiding me through my diploma thesis, the value of *secure* passwords and Ben taught me about procrastination. I will also miss Srivatsan wandering through the hallways with a salad in his hands and trying to convince me that his favorite football player is not just a middle of the pack striker.

Special thanks go to Felix and Matthias with whom I spend a lot of my time. I enjoyed the many discussions with Felix, starting from topics like football and ending up in the most absurd philosophical topics. He amazed me with his incredible knowledge and I was always interested in his opinion. Matthias will probably never learn how to play SET properly, or how not to work more than his own body can handle, but he was one of the main reasons why I enjoyed my time, especially during the very entertaining deadlines.

An important person missing here is my office mate and old friend Carlo. He brought me into the group for my diploma thesis and went with me through all the struggles of a researcher's life. Even though we are always competing in anything we do, I will (only once in my lifetime) admit that he is a genius (as a sidenote in case he is reading this: I understood FLP first though!) and I am proud that I can call me one of his friends.

I would like to thank everyone in the group whom I did not name personally and who was part of this great experience. You all made INET a superb place.

Finally, there is my family who was always there for me and helped me through all the ups and downs. Thank you Gerd, Brigitte, Lea and Nori. You are the best!

List of Figures

List of Tables

Bibliography

[1] Amazon network outage (amazon web site), April 2011. http://aws.amazon.com/message/65648/.

[2] Github network outage (github web site), December 2012. https://github.com/blog/1346-network-problems-last-friday.

[3] MPLS pricing (networkworld web site), April 2012. http://www.networkworld.com/article/2222196/cisco-subnet/why-does-mpls-cost-so-much-more-than-internet-connectivity-.htm l.

[4] Amazon AWS cloud pricing principles (amazon web site), October 2015. http://aws.amazon.com/pricing/.

[5] Aws case study: Spotify (amazon aws web site), October 2015. https://aws.amazon.com/solutions/case-studies/spotify/.

[6] Microsoft azure pricing (microsoft web site), October 2015. https://azure.microsoft.com/en-us/pricing/.

[7] Open networking foundation (web site), October 2015. https://www.opennetworking.org/.

[8] Roundup of cloud computing forecasts and market estimates, 2015 (forbes web site), January 2015. http://www.forbes.com/sites/louiscolumbus/2015/01/24/roundup-of-cloud-computing-forecasts-and-market-estimates-2015/.

[9] Unify (web site), October 2015. http://www.fp7-unify.eu.

[10] V. Abhishek, I. A. Kash, and P. Key. Fixed and market pricing for cloud services. In *Proc. NetEcon Workshop*, 2012.

[11] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proc. SIGCOMM*, 2008.

[12] J. Altmann, C. Courcoubetis, G. D. Stamoulis, M. Dramitinos, T. Rayna, M. Risch, and C. Bannink. GridEcon: A market place for computing resources. In *Proc. 5th international workshop on Grid Economics and Business Models (GECON)*, pages 185–196, 2008.

[13] T. Anderson, L. Peterson, S. Shenker, and J. Turner. Overcoming the internet impasse through virtualization. *Computer*, 38(4), 2005.

[14] P. Antoniadis, S. Fdida, T. Friedman, and V. Misra. Federation of virtualized infrastructures: sharing the value of diversity. In *Proc. 6th CoNEXT*, 2010.

[15] M. Armbrust, A. Fox, R. Grifith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. In *UC Berkeley Technical Report EECS-2009-28*, 2009.

[16] B. Awerbuch, Y. Azar, and S. Plotkin. Throughput-competitive on-line routing. In *Proc. IEEE FOCS*, 1993.

[17] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. The price is right: towards location-independent costs in datacenters. In *HotNets*, 2011.

[18] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proc. ACM SIGCOMM*, pages 242–253, 2011.

[19] N. Bansal, K.-W. Lee, V. Nagarajan, and M. Zafer. Minimum congestion mapping in a cloud. In *Proc. ACM PODC*, pages 267–276, 2011.

[20] A. Belbekkouche, M. Hasan, and A. Karmouch. Resource discovery and allocation in network virtualization. *IEEE Communications Surveys Tutorials*, (99):1–15, 2012.

[21] T. Benson, A. Akella, and D. A. Maltz. Unraveling the complexity of network management. In *NSDI*, pages 335–348, 2009.

[22] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. ACM IMC*, 2010.

[23] M. Bienkowski, A. Feldmann, D. Jurca, W. Kellerer, G. Schaffrath, S. Schmid, and J. Widmer. Competitive analysis for service migration in VNets. In *Proc. ACM VISA*, pages 17–24, 2010.

[24] A. Blenk, A. Basta, M. Reisslein, and W. Kellerer. Survey on network virtualization hypervisors for software defined networking. 2015.

[25] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[26] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Elsevier FGCS*, 25(6), 2009.

[27] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. A distributed and robust sdn control plane for transactional network updates. In *Proc. IEEE INFOCOM*, 2015.

[28] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proc. ACM SIGCOMM*, 2007.

[29] M. Charikar, K. Makarychev, and Y. Makarychev. A divide and conquer algorithm for d-dimensional arrangement. In *Proc. ACM SODA*, pages 541–546, 2007.

[30] S. Chen and K. Nahrsted. An overview of quality of service routing for next-generation high-speed networks: problems and solutions. *Network, IEEE*, 12(6):64–79, 1998.

[31] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: a cross-industry study of mapreduce workloads. *Proc. of the VLDB Endowment*, 5(12), 2012.

[32] M. K. Chowdhury and R. Boutaba. A survey of network virtualization. *Elsevier Computer Networks*, 54(5), 2010.

[33] M. K. Chowdhury, M. R. Rahman, and R. Boutaba. Virtual network embedding with coordinated node and link mapping. In *Proc. IEEE INFOCOM*, 2009.

[34] M. K. Chowdhury, F. Samuel, and R. Boutaba. PolyViNE: Policy-based virtual network embedding across multiple domains. In *Proc. ACM VISA*, 2010.

[35] C. Courcoubetis and R. R. Weber. Economic issues in shared infrastructures. In *Proc. ACM VISA*, pages 89–96, 2009.

[36] D. Dash, V. Kantere, and A. Ailamaki. An economic model for self-tuned cloud caching. In *Proc. IEEE ICDE*, pages 1687–1693, 2009.

[37] D. Drutskoy, E. Keller, and J. Rexford. Scalable network virtualization in software-defined networks. *Internet Computing, IEEE*, PP(99):1, 2012.

[38] N. G. Duffield, P. Goyal, A. Greenberg, P. Mishra, K. K. Ramakrishnan, and J. E. van der Merive. A flexible model for resource management in virtual private networks. In *ACM SIGCOMM CCR*, volume 29, pages 95–108. ACM, 1999.

[39] R.-c. Duh and M. Fürer. Approximation of k-set cover by semi-local optimization. In *Proc. ACM STOC*, STOC '97, pages 256–264, New York, NY, USA, 1997. ACM.

[40] N. Economides. The economics of the internet backbone. In *Handbook of Telecommunications Economics*, 2006.

[41] ETSI. Network Functions Virtualisation – Introductory White Paper. 2012.

[42] G. Even, M. Medina, G. Schaffrath, and S. Schmid. Competitive and deterministic embeddings of virtual networks. In *Proc. ICDCN*, 2012.

[43] J. Fan and M. H. Ammar. Dynamic topology configuration in service overlay networks: A study of reconfiguration policies. In *Proc. IEEE INFOCOM*, 2006.

[44] N. Feamster and H. Balakrishnan. Detecting bgp configuration faults with static analysis. In *Proc. NSDI*, pages 43–56. USENIX Association, 2005.

[45] N. Feamster, J. Rexford, and E. Zegura. The road to sdn. *Queue*, 11(12):20:20–20:40, 2013.

[46] A. Fischer, J. F. Botero, M. Till Beck, H. De Meer, and X. Hesselbach. Virtual network embedding: A survey. *Communications Surveys & Tutorials, IEEE*, 15(4):1888–1906, 2013.

[47] R. Fleischer. On the bahncard problem. *Theor. Comput. Sci.*, 268(1):161–174, 2001.

[48] C. Fuerst, M. Pacut, P. Costa, and S. Schmid. How hard can it be? understanding the complexity of replica aware virtual cluster embeddings. In *Proc. IEEE ICNP*, 2015.

[49] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: fair allocation of multiple resource types. In *Proc. 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2011.

[50] S. Ghorbani and B. Godfrey. Towards correct network virtualization. In *Proc. ACM HotSDN*, pages 109–114, 2014.

[51] S. Goyal and B. Giri. Recent trends in modeling of deteriorating inventory. *Elsevier EJOR*, 134(1), 2001.

[52] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM CCR*, volume 39, pages 51–62. ACM, 2009.

[53] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4d approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, 2005.

[54] S. Guha and K. Munagala. Improved algorithms for the data placement problem. In *Proc.ACM SODA*, pages 106–107, 2002.

[55] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. SecondNet: A data center network virtualization architecture with bandwidth guarantees. In *Proc. ACM CoNEXT*, 2010.

[56] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett. Sdx: A software defined internet exchange. In *Proc. ACM SIGCOMM*, pages 551–562, 2014.

[57] A. Haider, R. Potter, and A. Nakao. Challenges in resource allocation in network virtualization. In *Proc. ITC Specialist Seminar*, volume 18, page 20, 2009.

[58] P. Hande, M. Chiang, R. Calderbank, and S. Rangan. Network pricing and rate allocation with content-provider participation. In *Proc. IEEE INFOCOM*, 2010.

[59] T. A. Henzinger, A. V. Singh, V. Singh, T. Wies, and D. Zufferey. A marketplace for cloud resources. In *Proc. ACM EMSOFT*, 2010.

[60] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center. In *Proc. USENIX NSDI*, 2011.

[61] X. Hu, S. Schmid, A. Richa, and A. Feldmann. Optimal migration contracts in virtual networks: Pay-as-you-come vs pay-as-you-go pricing. In *Proc. ICDCN*, 2013.

[62] G. Iannaccone, C.-N. Chuah, S. Bhattacharyya, and C. Diot. Feasibility of ip restoration in a tier 1 backbone, 2004.

[63] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM CCR*, volume 43, pages 3–14. ACM, 2013.

[64] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Bridging the tenant-provider gap in cloud services. In *Proc. ACM SoCC*, 2012.

[65] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, J. Rexford, R. Wattenhofer, and M. Zhang. Dionysus: Dynamic scheduling of network updates. In *Proc. ACM SIGCOMM*, 2014.

[66] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. USENIX NSDI*, pages 99–112, 2013.

[67] A. Khan, A. Zugenmaier, D. Jurca, and W. Kellerer. Network virtualization: a hypervisor for the internet? *Communications Magazine, IEEE*, 50(1):136–143, 2012.

[68] H. Kim and N. Feamster. Improving network management with software defined networking. *Communications Magazine, IEEE*, 51(2):114–119, 2013.

[69] S. Kniesburges, C. Markarian, F. M. auf der Heide, and C. Scheideler. Algorithmic aspects of resource management in the cloud. In *Structural Information and Communication Complexity*, pages 1–13. Springer, 2014.

[70] D. Kreutz, F. M. Ramos, P. Esteves Verissimo, C. Esteve Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proc. of the IEEE*, 103(1):14–76, 2015.

[71] A. Kumar, A. Gupta, and T. Roughgarden. A constant-factor approximation algorithm for the multicommodity rent-or-buy problem. In *Proc. IEEE FOCS*, 2002.

[72] M. Kuzniar, P. Peresini, and D. Kostic. What you need to know about sdn flow tables. In *Proc. PAM*, 2015.

[73] D. Levin, M. Canini, S. Schmid, F. Schaffert, and A. Feldmann. Panopticon: Reaping the benefits of incremental sdn deployment in enterprise networks. In *Proc. USENIX ATC*, 2014.

[74] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *Proc. ACM IMC*, pages 1–14. ACM, 2010.

[75] S. Li, A. Mäcker, C. Markarian, F. M. auf der Heide, and S. Riechers. Towards flexible demands in online leasing problems. In *Computing and Combinatorics*, pages 277–288. Springer, 2015.

[76] J. Lischka and H. Karl. A virtual network mapping algorithm based on subgraph isomorphism detection. In *Proc. ACM VISA*, pages 81–88, 2009.

[77] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *ACM SIGCOMM*, August 2013.

[78] J. Lu and J. Turner. Efficient mapping of virtual networks onto a shared substrate. In *Technical Report, WUCSE-2006-35, Washington University*, 2006.

[79] M. Armbrust et al. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[80] J. R. M. Yu, Y. Yi and M. Chiang. Rethinking virtual network embedding: Substrate support for path splitting and migration. *ACM SIGCOMM CCR*, 38(2):17–29, Apr 2008.

[81] J. K. MacKie-Mason and H. R. Varian. Pricing congestible network resources. *Selected Areas in Communications, IEEE Journal on*, 13(7):1141–1149, 1995.

[82] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *Proc. ACM HotNets*, 2013.

[83] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient synthesis of network updates. In *Proc. ACM PLDI*, 2015.

[84] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[85] A. M. Meyerson. The parking permit problem. In *Proc. IEEE FOCS*, pages 274–284, 2005.

[86] J. C. Mogul and L. Popa. What we talk about when we talk about cloud network performance. *ACM SIGCOMM CCR*, sep 2012.

[87] B. Monien and H. Sudborough. Embedding one interconnection network in another. In *Computational Graph Theory*, 1990.

[88] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. In *Proc. NSDI*, 2013.

[89] A. Odlyzko. Paris metro pricing for the internet. In *Proc. ACM conference on Electronic commerce*, pages 140–147. ACM, 1999.

[90] P. Skoldstrom et al. Towards unified programmability of cloud and carrier infrastructure. In *Proc. EWSDN*, 2014.

[91] R. Pal and P. Hui. Economic models for cloud service markets. In *Proc. 13th ICDCN*, pages 382–396, 2012.

[92] B. Quoitin, V. V. den Schrieck, P. François, and O. Bonaventure. Igen: Generation of router-level internet topologies through network design heuristics. In *Proc. ITC*, 2009.

[93] P. Reichl, D. Hausheer, and B. Stiller. The cumulus pricing model as an adaptive framework for feasible, efficient, and user-friendly tariffing of internet services. *Computer Networks*, 43(1):3–24, 2003.

[94] P. Reichl, S. Leinen, and B. Stiller. A practical review of pricing and cost recovery for internet services, 1999.

[95] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proc. ACM SIGCOMM*, pages 323–334, 2012.

[96] R. Ricci, C. Alfeld, and J. Lepreau. A solver for the network testbed mapping problem. *ACM SIGCOMM CCR*, 33(2):65–81, 2003.

[97] H. Rodrigues, J. R. Santos, Y. Turner, P. Soares, and D. Guedes. Gatekeeper: Supporting bandwidth guarantees for multi-tenant datacenter networks. In *WIOV*, 2011.

[98] M. Rost, C. Fuerst, and S. Schmid. Beyond the stars: Revisiting virtual cluster embeddings. *ACM SIGCOMM CCR*, 45(3):12–18, July 2015.

[99] F. S. Salman, J. Cheriyan, R. Ravi, and S. Subramanian. Buy-at-bulk network design: Approximating the single-sink edge installation problem. In *Proc. ACM SODA*, pages 619–628, 1997.

[100] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proc. of the VLDB Endowment*, 3(1-2):460–471, 2010.

[101] G. Schaffrath, S. Schmid, and A. Feldmann. Optimizing long-lived cloudnets with migrations. In *Proc. IEEE/ACM UCC*, 2012.

[102] G. Schaffrath, S. Schmid, I. Vaishnavi, A. Khan, and A. Feldmann. A resource description language with vagueness support for multi-provider cloud networks. In *Proc. ICCCN*, 2012.

[103] G. Schaffrath, C. Werle, P. Papadimitriou, A. Feldmann, R. Bless, A. Greenhalgh, A. Wundsam, M. Kind, O. Maennel, and L. Mathy. Network virtualization architecture: Proposal and initial prototype. In *Proc. ACM VISA*, 2009.

[104] V. Sekar, S. Ratnasamy, M. K. Reiter, N. Egi, and G. Shi. The middlebox manifesto: Enabling innovation in middlebox deployment. In *Proc. ACM HotNets*, 2011.

[105] S. Shakkottai and R. Srikant. Economics of network pricing with multiple isps. *IEEE/ACM TON*, 14(6), 2006.

[106] D. Songhurst and F. Kelly. Charging schemes for multiservice networks. In *Proc. 15th International Teletraffic Congress*, 1997.

[107] Z. Wang and J. Crowcroft. Quality-of-service routing for supporting multimedia applications. *Selected Areas in Communications, IEEE Journal on*, 14(7):1228–1234, 1996.

[108] D. P. Williamson, M. X. Goemans, M. Mihail, and V. V. Vazirani. A primal-dual approximation algorithm for generalized Steiner network problems. *Combinatorica*, 15:708–717, 1995.

[109] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: meeting deadlines in datacenter networks. In *Proc. ACM SIGCOMM*, 2011.

[110] Z. Wu, C. Yu, and H. V. Madhyastha. Costlo: Cost-effective redundancy for lower latency variance on cloud storage services. In *Proc. USENIX NSDI*, 2015.

[111] D. Xie, N. Ding, Y. C. Hu, and R. Kompella. The only constant is change: incorporating time-varying network reservations in data centers. In *Proc. ACM SIGCOMM*, pages 199–210, 2012.

[112] F. Zaheer, J. Xiao, and R. Boutaba. Multi-provider service negotiation and contracting in network virtualization. In *Proc. 12th IEEE/IFIP Network Operations and Management Symposium (NOMS 2010)*, 2010.

[113] Y. Zhu and M. H. Ammar. Algorithms for assigning substrate network resources to virtual network components. In *Proc. IEEE INFOCOM*, 2006.