2013-10-20

A policy-based architecture for virtual network embedding (PhD thesis)

https://hdl.handle.net/2144/11420

Boston University
BOSTON UNIVERSITY
GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

A POLICY-BASED ARCHITECTURE
FOR VIRTUAL NETWORK EMBEDDING

by

FLAVIO ESPOSITO
M.S., Universitá di Firenze, 2005

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

2014
Approved by

First Reader

Abraham Matta, PhD
Professor of Computer Science

Second Reader

Azer Bestavros, PhD
Professor of Computer Science

Third Reader

John Byers, PhD
Associate Professor of Computer Science
ABSTRACT

Network virtualization is a technology that enables multiple virtual instances to co-exist on a common physical network infrastructure. This paradigm fostered new business models, allowing infrastructure providers to lease or share their physical resources. Each virtual network is isolated and can be customized to support a new class of customers and applications.

To this end, infrastructure providers need to embed virtual networks on their infrastructure. The virtual network embedding is the (NP-hard) problem of matching constrained virtual networks onto a physical network. Heuristics to solve the embedding problem have exploited several policies under different settings. For example, centralized solutions have been devised for small enterprise physical networks, while distributed solutions have been proposed over larger federated wide-area networks.

In this thesis we present a policy-based architecture for the virtual network embedding problem. By policy, we mean a variant aspect of any of the three (invariant) embedding mechanisms: physical resource discovery, virtual network mapping, and allocation on the physical infrastructure. Our architecture adapts to different scenarios by instantiating appropriate policies, and has bounds on embedding efficiency, and on convergence embedding time, over a single provider, or across multiple federated providers. The performance of representative novel and existing policy configurations...
tions are compared via extensive simulations, and over a prototype implementation. We also present an object model as a foundation for a protocol specification, and we release a testbed to enable users to test their own embedding policies, and to run applications within their virtual networks. The testbed uses a Linux system architecture to reserve virtual node and link capacities.
Contents

1 Introduction 1
   1.1 Motivation .................................................. 1
   1.2 Thesis Contributions and Organization ...................... 4

2 Slice Embedding Solutions for Distributed Service Architectures 7
   2.1 Background and Area definition .............................. 9
      2.1.1 The Slice Embedding Problem ......................... 10
      2.1.2 Interactions in the Slice Embedding Problem .......... 11
      2.1.3 Solutions to the Virtual Network (Slice) Embedding Problem . 11
      2.1.4 The novelty of the slice embedding problem .......... 12
   2.2 On Modeling the Slice Embedding Problem .................... 14
      2.2.1 Discovery ............................................. 15
      2.2.2 Virtual Network Mapping .............................. 16
      2.2.3 Allocation ............................................ 18
      2.2.4 Complete Slice Embedding ............................. 18
   2.3 Taxonomy .................................................. 21
      2.3.1 Constraint type ....................................... 21
      2.3.2 Dynamics .............................................. 21
      2.3.3 Admission Control ..................................... 22
   2.4 Resource Discovery ......................................... 23
      2.4.1 Resource Description ................................... 24
      2.4.2 Discovery and virtual network mapping .................. 24
   2.5 Virtual Network Mapping ................................... 26
2.5.1 Network mapping without constraints ........................................ 26
2.5.2 Network mapping with constraints ........................................... 28
2.5.3 Network mapping + allocation .................................................. 29
2.5.4 Dynamic approaches to network mapping and allocation ............. 32
2.5.5 Distributed Virtual Network Mapping Solutions ......................... 35
2.6 Allocation ............................................................................. 37
2.6.1 Game-theory based allocation .................................................. 38
2.6.2 Reservation-based allocation ..................................................... 39
2.6.3 Market-based allocation and guarantees .................................... 41
2.7 Distributed Embedding Solutions ................................................. 44

3 Virtual Network Embedding Applications ....................................... 46
3.1 Virtual Network Embedding Workflow ......................................... 46
3.2 Testbed Embedding Services ....................................................... 48
3.3 Cloud Computing Services for Enterprise Networks ....................... 49

4 A General Distributed Approach to Slice Embedding with Guarantees ........................................................................... 52
4.1 Consensus-based Auctions for
Distributed Slice Embedding ............................................................ 53
4.1.1 CADE mechanism ................................................................ 55
4.1.2 CADE Policies ...................................................................... 56
4.1.3 Phase 1: CADE Election Phase .............................................. 58
4.1.4 Phase 2: CADE Agreement Phase .......................................... 61
4.1.5 Conflicts resolution ............................................................... 63
4.1.6 Pseudo sub-modular utility functions ..................................... 64
4.1.7 Phase 3: Virtual Link Embedding .......................................... 66
4.2 Convergence and Performance Guarantees .................................. 67
4.2.1 Convergence Analysis ................................. 67
4.2.2 Performance Guarantees ............................... 68
4.3 Performance Evaluation ................................. 69
4.3.1 Simulation results .................................. 72
4.3.2 Path Allocations .................................. 74

5 Embedding Protocol and
Object Model Design ................................. 79
  5.1 Broker (SIB Daemon) ................................. 80
  5.2 SIB Daemon API .................................. 81
  5.3 CADE Protocol Messages
     (Operations on Objects) .......................... 83
  5.4 Virtual Network Embedding Objects .................. 87

6 VINEA Prototype Implementation ......................... 94
  6.1 Common VINEA Node Capabilities .................... 95
  6.2 Service Provider Capabilities ....................... 98
  6.3 Infrastructure Provider Capabilities ................. 99
      6.3.1 Resource Directory Service ..................... 99
      6.3.2 Virtual Network Mapping Service ............... 101
  6.4 VINEA Testbed ................................ 106
  6.5 VINEA Prototype Evaluation ......................... 107
      6.5.1 Embedding Success Rate ....................... 110
      6.5.2 Overhead ................................ 112

7 Conclusions .................................. 114
  7.1 Summary of Contributions ......................... 114
  7.2 Future Work ................................ 115

A CADE Synchronous Agreement Rules ...................... 117
B VINEA Asynchronous Agreement Rules 119

C Physical Node Configuration File 122
   C.1 Physical Node Configuration File 122
   C.2 Service Provider Configuration File 124

Bibliography 125

Curriculum Vitae 141
List of Tables

2.1 The different combinations of slice embedding constraints classify the related work. None of the existing solutions holistically consider discovery, mapping and allocation. For clarity of representation, we omit the binary constraints $(4k)$. ........................................... 20

A.1 Rules table for cade synchronous conflict resolution. The sender physical node is denoted with $k$, and the receiver physical node with $i$. The time vector $s$ represents the time stamp of the last information update from each of the other agents. ........................................... 118

B.1 Rules table for vinea asynchronous conflict resolution. The sender physical node is denoted with $k$, and the receiver physical node with $i$ (Table 1 of 2). ........................................... 120

B.2 Rules table for vinea asynchronous conflict resolution. The sender physical node is denoted with $k$, and the receiver physical node with $i$ (Table 2 of 2). ........................................... 121
List of Figures

2.1 Interactions and data exchanges in the slice embedding problem. . . . 12
2.2 Interactions and data exchanges in the slice embedding problem with
the notation used in our model. . . . . . . . . . . . . . . . . . . . . . . 15
2.3 Overview of the slice embedding taxonomy with classification of rep-
resentative references. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22
2.4 Path splitting and migration mapping algorithm [YYRC08]. . . . . . . 31
2.5 Architecture and allocation phases in SHARP [FCC+03]. . . . . . . . 40
2.6 Different values of Oversubscription Degree tune allocation guaran-
tees [FCC+03]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

3.1 Virtual network embedding workflow: a service provider and many
infrastructure providers cooperate or compete to embed a constrained
virtual network request (Figure inspired by [SWP+09]). . . . . . . . . 47

4.1 Virtual network embedding architecture: mechanisms and interfaces. 54
4.2 (a) Slice with capacity constraints to be embedded. (b) Each physical
node (PN) can be owned by a different InP, and can have a different
capacity. (c) CADE workflow: a virtual link embedding phase follows
the virtual node embedding phase. . . . . . . . . . . . . . . . . . . . . . 54
4.3 (left) CDF of the size and lifetime of 8 years of Emulab slice requests.
(right) SAD better balances the load on physical nodes (k = 3). S,
M, H and P indicate SAD, MAD, Hub and Spoke and PolyViNE,
respectively. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
4.4 (left) SAD allocates more slices when a single shortest path is available. (right) MAD allocates more slices when a $k$-shortest path link allocation policy (where $k = 3$) is used. 70

4.5 (left) MAD has shorter convergence time. (right) SAD has shorter response time. 70

4.6 (left) MAD allocates more slices consecutively ($k = 3$). (right) Considering simultaneously node and link stress in the utility improves the slice allocation ratio. 71

4.7 Slice allocation ratio: physical network of 100 nodes following Barabasi-Albert connectivity model. 76

4.8 Link Allocation Ratio: physical network of 100 nodes following Barabasi-Albert connectivity model. 77

4.9 Node allocation ratio: physical network of 100 nodes following Barabasi-Albert connectivity model. 78

5.1 Object Model interfaces: the SIB daemon is responsible for managing the information stored in the SIB and its veracity, updating and making states available to service and infrastructure provider processes. 81

6.1 VINEA prototype architecture. Each node can be instantiated as a service provider or an infrastructure provider. 95

6.2 (left) InP overlay with three embedded virtual networks. (right) VINEA resulting configuration: for each InP process hosting a virtual node there is a virtual switch and at least a virtual host attached to it. An OVS Openflow controller is also attached to each virtual switch. . . . 105
6.3 Testbed Architecture: A physical machine running Linux Ubuntu (version 12.04) hosts the VINEA prototype. Physical wires are emulated with loopback TCP connections on well-known ports. After the virtual networks are embedded, we can run Linux applications between virtual nodes, e.g. ping, traceroute, or we can send data traffic, and measure the reserved bandwidth performance with iperf.

6.4 Five InP processes connected with a linear topology hosting: (left) VNs with linear virtual topology and (right) VNs with full virtual topology. The SAD policy balances the load on the InP processes, and permits an higher acceptance rate as long as only a single loop-free physical path is available.

6.5 Five fully connected InP processes hosting: (left) VNs with a linear virtual topology and (right) VNs with full virtual topology. MAD is outperformed as collocating more virtual nodes on the same InP process quickly exhausts the available physical link capacity: multiple outgoing virtual links have one end hosted on the same InP process. If the link embedding policy allows multiple physical loop-free paths, the MAD policy performs better.

6.6 The MAD policy has less overhead than SAD as the embedding does not require partitioning the virtual network.

6.7 InP processes receiving the request have higher overhead as they propagate the virtual network objects, including their bids, and then respond to the SP.
List of Abbreviations

CADE  Consensus-based Algorithm for Distributed slice Embedding
CFS   Completely Fair Scheduler
DNS   Domain Name System
GENI  Global Environment for Network Innovations
InP   Infrastructure Provider
ISO   International Standard Organization
ISD   Inter-Slice Directory Service
MAD   Multiple Allocation for Distributed slice embedding
NMS   Network Management System
NP    Non-deterministic Polynomial
OSI   Open Systems Interconnection
PAD   Path Allocation for Distributed slice embedding
RINA  Recursive InterNetwork Architecture
RT    Real Time
SAD   Single Allocation for Distributed slice embedding
SP    Service Provider
SLA   Service Level Agreement
SLO   Service Level Objective
SIB   Slice Information Base
TCP/IP Transport Control Protocol/Internet Protocol
VPN   Virtual Private Network
VINI  VIrtual Network Infrastructure
VINEA VIrtual Network Embedding Architecture
XIA   Expressive Internet Architecture
Chapter 1

Introduction

1.1 Motivation

Network virtualization is a technology that enables hardware network resources to be shared among multiple concurrent software instances, \textit{i.e.}, virtual networks. Each virtual network is a collection of virtual nodes and virtual links that connect a subset of the underlying physical network infrastructure. Such infrastructure can be owned by a single provider, or by multiple providers.

Virtual networks are customizable, \textit{i.e.}, they can be configured and operated according to different protocols or architectures, each tailored for a particular service or application \cite{TT05}. Examples of virtual network applications are within the business community \cite{FGR07, CJ09, Rac13a, WGR+09}, \textit{e.g.}, an enterprise leasing an Akamai \cite{NSS10a} content delivery network, or within the research community \cite{Glo09, PACR03, WLS+06, BFH+06, EM09}, \textit{e.g.}, a researcher sharing a testbed infrastructure to experiment with a new protocol on a large-scale virtual network.

Virtual network embedding mechanisms. In this thesis, we focus on the virtual network embedding, the NP-hard \cite{CV03} problem of matching constrained virtual networks onto a physical infrastructure. The problem involves a Service
Provider (SP), providing a virtual network service, and a set of physical Infrastructure Providers (InPs), that own the physical resources. There are three interacting mechanisms that operate when a virtual network needs to be embedded onto a physical network: resource discovery, virtual network mapping, and resource allocation. 

*Resource discovery* is the process of monitoring the state of the substrate resources using sensors and other measurement processes. The monitored states include processor loads, memory usage, network performance data, etc.

*Virtual network mapping* is the step that matches requests for virtual networks, with the available physical resources, and selects some set of physical resources to host the virtual network. Due to the combination of node and link constraints, this is the most complex step in the embedding problem. These constraints include intra-node requirements, e.g., desired physical location, processor speed, storage capacity, type of network connectivity, as well as inter-node constraints, e.g. network topology.

*Allocation* involves assigning the physical resources that match the virtual network (VN) requests to each VN, considering additional constraints, e.g. the physical limits of the infrastructure. A VN is also known as slice, as the physical network infrastructure is sliced into isolated virtual networks.

**Embedding policies and design tradeoffs.** Prior work on the virtual network embedding problem explore different policies for both service providers and infrastructure providers. By policy, we mean a variant aspect of any of the three (invariant) embedding mechanisms. In Chapter 2 we survey the prior solutions across several design dimensions. For example, some solutions aim to maximize the revenue of a single provider, by maximizing the number of virtual networks that can be embedded on their physical infrastructure [ZA06, YYRC08], under the assumption that the virtual network requester pays a monetary price proportional to the amount of physical capacity needed to embed its request. Other so-
olutions aim to find an embedding that satisfies the constraints of experimenters requesting a virtual network from a testbed [CBMP04, Opp05, JA08]. In order to adapt the embedding to dynamic physical infrastructure conditions with the least possible state changes (migrations), some solutions embed virtual nodes first, and then virtual links [YYRC08]. Other approaches have shown that the physical network utilization increases when a virtual network is embedded simultaneously considering virtual nodes and links [CRB09, HLZ08]. Some heuristics have assumed a centralized virtual network embedding onto a small (enterprise) physical network [CRB09, ZA06, YYRC08], where the service provider and infrastructure provider are one; other solutions distribute the embedding decision of a larger-scale virtual network among different (federated) infrastructure providers [HLZ08, CSB10].

Few existing distributed solutions outsource the embedding to a centralized SP that coordinates the process by either splitting the VN request and sending it to a subset of Infrastructure Provider (InP)s [HLBAZ11], or by collecting resource availability from InPs and later offering an embedding [ZZSRR08]. Outsourcing has the advantage of relieving InPs from the entire management complexity, but a single centralized authority [ZZSRR08] could be untrusted, a single point of failure, or both.

Distributed virtual network mapping solutions that allow different InPs to collectively embed a slice already exist [HLZ08, ZXB10, CSB10]: some of them focus on the desirable property of letting InPs use their own (embedding) policies [CSB10], while others rely on truthfulness of a virtual resource auction [ZXB10]. Although they have systematic logic behind their design, such distributed solutions are still restricted to a subset of the three slice embedding tasks, they have performance (e.g. convergence speed or resource utilization) tightly determined by the chosen heuristic, and they are limited to a single distribution model — the type and amount of information propagated to embed a slice.
Existing embedding solutions are also restrictive with respect to slice’s arrival rate and duration: the lifetime of a slice can range from few seconds or minutes (in the case of cluster-on-demand services) to several months and years (in the case of a slice hosting a content distribution service similar to Akamai [NSS10b], or a Global Environment for Network Innovations (GENI) [Glo09] slice hosting a novel architecture looking for new adopters to opt-in.) For instance, in wide-area testbed applications, slices are provided in a best-effort manner, and the inter-arrival time between slice requests and the lifetime of slices are typically much longer than the slice embedding time, so existing solutions assume complete knowledge of the network state, and ignore the overhead of resource discovery and the slice embedding time. In applications with higher churns, e.g., cluster-on-demand for applications such as financial modeling, anomaly analysis, or heavy image processing, where slice providers have rigid Service Level Objective (SLO) — the technical requirements within a Service Level Agreement (SLA) — or where slices have short lifetime and request short response time, it is desirable that solutions attempt to reduce the slice embedding time, and employ limited resource discovery to reduce overhead.

1.2 Thesis Contributions and Organization

Due to the wide range of providers’ goals and allocation models (e.g., best effort or SLO), a flexible solution that is adaptable to different provider goals and tackles the distributed virtual network embedding with its three phases does not yet exist. Moreover, none of the previously proposed solutions studying the virtual network embedding problem provides a complete architecture framework, a realistic prototype implementation, or gives guarantees on either the convergence of the slice embedding process, or the allocation performance — ratio of the number of slices successfully allocated on the physical infrastructure to the total requested. To this end, in this thesis we make the following contributions:
• **Modeling Existing Solutions:** we leverage optimization theory to provide a survey of recent virtual network embedding solutions for distributed service architectures. To dissect the space of solutions, we introduce a taxonomy with three main classification criteria, namely, (1) the type of constraints imposed by the user, *i.e.*, the person or the machine requesting a virtual network through a service provider, (2) the type of dynamics considered in the embedding process, and (3) the allocation strategy adopted (Chapter 2).

• **Motivating Applications:** we outline few key motivating applications for our architecture. For example, in the testbed embedding service community our architecture can be used to solve the GENI virtual network stitching problem, *i.e.*, the problem of merging two separate slices whose physical resources belong to different (federated) infrastructure providers [GEN11]. Moreover, we show how a virtual network can be instantiated to provide some popular cloud computing services, such as load balancing as a service [Rac13b], and Virtual Private Network (VPN) as a service [WGR+09, Amaa], when the physical resources belong to multiple cloud providers, or to geographically dispersed data centers of the same cloud provider (Chapter 3).

• **Protocol Design, Theoretical Bounds and Simulation Results:** we propose a novel policy-based distributed virtual network embedding protocol that we called Consensus-based Algorithm for Distributed slice Embedding (CADE), and compare different instances (policies) of the common embedding mechanisms. Leveraging the consensus literature and the properties of submodular functions, we show how our solutions provides theoretical bounds on embedding efficiency and convergence time. Moreover, via extensive simulations, we show how our embedding protocol outperforms existing distributed virtual network embedding approaches (Chapter 4).
• **Definition of a Slice Embedding Object Model:** We define a management object model, as a foundation for a virtual network embedding protocol specification. An object model consists of: (i) a set of object attributes, that we define using the Google Protocol Buffer abstract syntax notation [Goo13], to utilize as a slice specification language descriptor, and during the embedding process; (ii) an interface to such object attributes, and (iii) a set of operations (protocol messages) to share and modify such object attributes (Chapter 5).

• **Prototype Implementation:** to establish the practicality of our architecture, we complete the evaluation with VIrtual Network Embedding Architecture (VINEA), a prototype implementation that includes the mechanisms for both service and infrastructure providers. The prototype resulted in about 35K lines of Java code, excluding comments and test classes (Chapter 6).

• **Testbed Evaluation:** to demonstrate our implementation, we realized a virtual network embedding testbed. Our base system is a host running an Ubuntu distribution of Linux (version 12.04). The physical network is emulated via TCP connections on the host loopback interface. Each physical node includes the VINEA modules with a Mininet-based [LHM10] implementation of the virtual network allocation mechanism. Each emulated virtual node is a user-level process that has its own virtual Ethernet interface(s), created and installed with `ip link add/set`, and is attached to an Open vSwitch [OVS13] running in kernel mode to switch packets across virtual interfaces. The data rate of each virtual link is enforced by Linux Traffic Control (tc), which has a packet scheduler to shape traffic to the configured rate (Chapter 6).
Chapter 2

Slice Embedding Solutions for Distributed Service Architectures

In the well-known layered International Standard Organization (ISO) Open Systems Interconnection (OSI) and Transport Control Protocol/Internet Protocol (TCP/IP) reference models [KR09], a layer is said to provide a service to the layer immediately above it. For example, the transport layer provides services (logical end-to-end channels) to the application layer, and the internetworking layer provides services (packet delivery across individual networks) to the transport layer.

The notion of distributed service architecture extends this service paradigm to many other (large scale) distributed systems. Aside from the Internet itself, including its future architecture design, e.g., Expressive Internet Architecture (XIA) [HAD+12], NetServ [SSK09] or Recursive InterNetwork Architecture (RINA) [DMM08], with the term distributed service architecture we refer to a large scale distributed system whose architecture is based on a service paradigm.

Some examples are datacenter-based systems [HB09], Cloud Computing [Hay08] (including high performance computing systems such as cluster-on-demand services), where the rentable resources can scale both up and down as needed, Grid Computing [gri03], overlay networks (e.g., content delivery networks [BLBS06] or [BCMR04]),
large scale distributed testbed platforms (e.g., PlanetLab [PACR03], Emulab [WLS+], VrIrtual Network Infrastructure (VINI) [BFH+06], GENI [GEN07]), or Service-oriented Architecture (SoA), where web applications are the result of the composition of services that need to be instantiated across a collection of distributed resources [YL05].

A common characteristic of all the above distributed systems is that they all provide a service to a set of users or, recursively, to another service. In this chapter, we restrict our focus on a particular type of service: a slice. We define a slice to be a set of virtual instances spanning a set of physical resources.

The lifetime span of a slice ranges from few seconds (in the case of cluster-on-demand services) to several years (in the case of a virtual network hosting a content distribution service similar to Akamai [NSS10b], or a GENI experiment hosting a novel architecture looking for new adopters to opt-in [GEN07]). Therefore, the methods to acquire, configure, and manage such slices could be different across different service architectures. In particular, the problem of discovering, mapping and allocating physical resources (slice embedding) has different time constraints in each service architecture.\(^1\)

In some distributed service architecture applications, e.g. virtual network testbed, the slice creation and embedding time is often negligible relative to the running time of the service they are providing. In many other applications, e.g. financial modeling, anomaly analysis, or heavy image processing, the time to solution — instant from when the user, application or service requests a slice till of task completion — is dominated by or highly dependent on the slice creation and embedding time.

Therefore, to be profitable, most of those service architectures require agility—the ability to allocate or deallocate any physical resource (node or link) to / from any service at any time.\(^2\) Those stringent requirements, combined with the imper-

\(^1\)By resources we mean processes, storage capacity, and physical links, as well as computational resources such as processors.

\(^2\)We extend the definition of agility as “ability to assign any server to any service” given by
fect design of today’s data center networks [GHJ+09] and with the lack of an ideal virtualization technology [WST11], have recently re-motivated research on resource allocation [EMI13, CBK10, ZA10, LBCP09, GHJ+09, ABC+09, SNP+05].

In this chapter, we define in Section 2.1 the slice embedding problem — a subarea of the resource allocation for service architectures. With the help of optimization theory, we model the three phases of the slice embedding problem as well as its tasks’ interactions (Section 2.2). We point out how all the proposed approaches either have not considered the slice creation and embedding time at all, or did not model some of the slice embedding tasks. The goal of our unifying model is to capture the “interactions” among the three slice embedding tasks. For example, the model captures resource discovery which can run the gamut from limited to full discovery. How much resources to discover affects metrics such as the overhead of discovery, response time, and the quality of the other two tasks of mapping and allocation. Similarly, the quality of the mapping affects the success of the final allocation given resource constraints, and accounting for resource constraints in the mapping process can yield better candidates for resource allocation.

We then give a taxonomy (Section 2.3), and we survey some of the recent solutions for each of the slice embedding tasks (Sections 2.4, 2.5 and 2.6). Finally, we discuss existing distributed virtual network embedding approaches (Section 2.7).

### 2.1 Background and Area definition

A recent survey on network virtualization can be found in [CB10]. The authors compare with a broad perspective, approaches related to network virtualization, e.g. virtual private networks and overlay networks. The paper also discusses economic aspects of service providers, analyzes their design goals (such as manageability or scalability), and overviews recent projects that use this technology (e.g. Planet-Greenberg et al. [GHJ+09] by including links and, other resources along with deallocation.)
lab [PACR03] and GENI [Glo09]). We narrow our focus on a more specific subarea of network virtualization (i.e. slice embedding), introducing a new taxonomy inspired by optimization theory for the three phases of the slice embedding problem. We leave our utility functions and model constraints as general as possible, so they can be instantiated, refined or augmented based on policies that would lead to efficient slice embedding solutions.

2.1.1 The Slice Embedding Problem

A slice is defined as a set of virtual instances spanning a set of physical resources of the network infrastructure. The slice embedding problem comprises the following three steps: resource discovery, virtual network mapping, and allocation.

Resource discovery is the process of monitoring the state of the substrate (physical) resources using sensors and other measurement processes. The monitored states include processor loads, memory usage, network performance data, etc. We discuss the resource discovery problem in Section 2.4.

Virtual network mapping is the step that matches users’ requests with the available resources, and selects some subset of the resources that can potentially host the slice. Due to the combination of node and link constraints, this is by far the most complex step in the slice embedding problem. In fact this problem is NP-hard [CV03]. These constraints include intra-node (e.g., desired physical location, processor speed, storage capacity, type of network connectivity), as well as internode constraints (e.g., network topology). We define the virtual network mapping problem in Section 2.5.

Allocation (Section 2.2.3) involves assigning the resources that match the user’s request to the appropriate slice. The allocation step can be a single shot process, or it can be repeated periodically to either reassign or acquire additional resources for a slice that has already been embedded.
2.1.2 Interactions in the Slice Embedding Problem

Before presenting existing solutions to the tasks encompassing the slice embedding problem, it is important to highlight the existence of interactions among these tasks, the nature of these interactions, how they impact performance, as well as the open issues in addressing these interactions.

In Figure 2·1, a user is requesting a set of resources using a resource description language. The arrow (1) going from the “Requests” to the “Discovery” block, represents user requests (queries) that could potentially have multiple levels of expressiveness and a variety of constraints. 3

The resource discoverer (2) returns a subset of the available resources (3) to the principle in charge of running the virtual network mapping algorithm (4). Subsequently, the slice embedding proceeds with the allocation task. A list of candidate mappings (5) are passed to the allocator (6), that decides which physical resources are going to be assigned to each user. The allocator then communicates the list of winners (7)—users that won the allocation—to the discoverer, so that future discovery operations can take into account resources that have already been allocated. It is important to note that the slice embedding problem is essentially a closed feedback system, where the three tasks are solved repeatedly—the solution in any given iteration affects the space of feasible solutions in the next iteration.

2.1.3 Solutions to the Virtual Network (Slice) Embedding Problem

Solutions in the current literature either solve a specific task of the slice embedding problem, or are hybrids of two tasks. Some solutions jointly consider resource discovery and network mapping [HS03, AOVP08], or discovery and allocation [AL12] (mapping single virtual machines), others only focus on the mapping

3The connection between the resource discovery and description is tight. In Section 2.4 we discuss how resources are described and advertised in both centralized and distributed models.
phase [ZA06, JA08, CBMP04], or on the interaction between virtual network mapping and allocation [YYRC08, LK09], while others consider solely the allocation step [ACSV04, BNCV05, LRAM05, FCC+03, CNA+04]. Moreover, there are solutions that assume the virtual network mapping task is solved, and only consider the interaction between the resource discovery and allocation [ROLV06]. We do not discuss solutions that address the resource discovery task in isolation, since it is not different from classical resource discovery in the distributed system literature (see [MRPM08] for an excellent survey on the topic). In addition to considering one [ZA06, ACSV04] or more [OAPV05, YYRC08] tasks, solutions also depend on whether their objective is to maximize the utility of users or providers.

### 2.1.4 The novelty of the slice embedding problem

The slice embedding problem, or more specifically its constituent tasks, and network virtualization in general, may seem identical to problems in classical distributed systems. Network virtualization, however, is different in several ways, namely: (a) it enables novel business models, (b) it enables the co-existence of novel network ap-
proaches, and (c) it creates new embedding challenges that must be addressed.

**Business models:** network virtualization lays the foundations for new business models [CW09]. Network resources are now considered commodities to be leased on demand. The leaser could be an infrastructure or service provider, and the lessee could be another service provider, an enterprise, or a single user (e.g. a researcher in the case of virtual network testbed as in [Glo09, BFH+06, HRS+08, PACR03, EM09]). In those cases where the infrastructure is a public virtualizable network testbed (e.g. GENI [Glo09]), the physical resources may not have any significant market value, since they are made available at almost no cost to research institutions.

**Coexisting network approaches:** the concept of multiple coexisting logical networks appeared in the networking literature several times in the past. The most closely related attempts are virtual private networks and overlay networks. A virtual private network (VPN) is a dedicated network connecting multiple sites using private and secured tunnels over a shared communication network. Most of the time, VPNs are used to connect geographically distributed sites of a single enterprise: each VPN site contains one or more customer edge devices attached to one or more provider edge routers [RR99].

An overlay network, on the other hand, is a logical network built on top of one or more existing physical networks. One substantial difference between overlays and network virtualization is that overlays in the existing Internet are typically implemented at the application layer, and therefore they may have limited applicability.

For example, they falter as a deployment path for radical architectural innovations in at least two ways: first, overlays have largely been in use as means to deploy narrow fixes to specific problems without any holistic view; second, most overlays have been designed in the application layer on top of the IP protocol, hence, they cannot go
beyond the inherent limitations of the existing Internet [APST05].

In the case of VPNs, the virtualization level is limited to the physical network layer while in the case of overlays, virtualization is limited to the end hosts. Network virtualization introduces the ability to access, manage and control each layer of the current Internet architecture in the end hosts, as well as providing dedicated virtual networks.

**Embedding challenges:** although the research community has explored the embedding of VPNs in a shared provider topology, *e.g.*, [DGG+02], usually VPNs have standard topologies, such as a full mesh. A virtual network in the slice embedding problem, however, may have any topology. Moreover, resource constraints in a VPN or overlays are limited to either bandwidth requirements or node constraints, while in network virtualization, both link and node constraints may need to be present simultaneously. Thus, the slice embedding problem differs from the standard VPN embedding because it must deal with both node and link constraints for arbitrary topologies.

### 2.2 On Modeling the Slice Embedding Problem

In this section we use optimization theory to model the interaction between the three tasks of the slice embedding problem. We first model each standalone task — resource discovery, virtual network mapping, and allocation — and subsequently model the slice embedding problem as a whole by merging the three phases into a centralized optimization problem. First, we start by providing the following definition:

**Definition 1.** *(Network)* A Network is defined as an undirected graph $G = (N, L, C)$ where $N$ is a set of nodes, $L$ is a set of links, and each node or link $e \in N \cup L$ is associated with a set of constraints $C(e)$. A physical network will be denoted as $G^P = (N^P, L^P, C^P)$, while a virtual network will be denoted as $G^V = (N^V, L^V, C^V)$. 
Consider the ellipsoid in Figure 2.2, augmented from Figure 2.1 (we explain the rest of the notation throughout this section): user $j$ requests a virtual network composed of $\gamma_j \in \mathbb{N}$ virtual nodes, $\psi_j \in \mathbb{N}$ virtual links and a vector of constraints $C_j(e)$ where $e$ is a vector of $c = \gamma_j + \psi_j$ elements — nodes and links — of the virtual network.

### 2.2.1 Discovery

To model the resource discovery we introduce two binary variables, $n_{ik}^P$ and $p_{kj}$ that are equal to 1 if the $i^{th}$ physical node and the $k^{th}$ loop-free physical path, respectively, are available, and zero otherwise. An element is available if a discovery operation is able to find it, given a set of protocol parameters, e.g., find all loop-free paths within a given deadline, or find as many available physical nodes as possible within a given number of hops.

If the system does not return at least $\gamma_j$ physical nodes and $\psi_j$ available loop-free physical paths among all the possible $N$ nodes and $P$ loop-free paths of the physical network $G^P$, then the user’s request should be immediately discarded. Among all
possible resources, the system may choose to return a set that maximizes a given notion of utility. Those utilities may have the role of selecting the resources that are closer — with respect to some notion of distance — to the given set of constraints $C_j(e)$. If we denote as $u_{ij} \in \mathbb{R}$ and $\omega_{kj} \in \mathbb{R}$ the utility of physical nodes and paths respectively, then the discovery phase of the slice embedding problem can be modeled as follows:

$$\begin{align*}
\text{maximize} & \quad f(n_{ij}^P, p_{kj}) = \sum_{i \in N} u_{ij} n_{ij}^P + \sum_{k \in P} \omega_{kj} p_{kj} \\
\text{subject to} & \quad \sum_{i \in N} n_{ij}^P \geq \gamma_j \\
& \quad \sum_{k \in P} p_{kj} \geq \psi_j \\
& \quad n_{ij}^P, p_{kj} \in \{0, 1\} \quad \forall i, j, k
\end{align*}$$  

(2.1)

After the discovery phase is completed, the set of available physical resources $\{n_{ij}^P, p_{kj}\}$ are passed to the virtual network mapper.

### 2.2.2 Virtual Network Mapping

The virtual network mapping problem is defined as follows [LK09]:

**Definition 2.** (Virtual Network Mapping) Given a virtual network $G^V = (N^V, L^V, C^V)$ and a physical network $G^P = (N^P, L^P, C^P)$, a virtual network mapping is a mapping of $G^V$ to a subset of $G^P$, such that each virtual node is mapped onto exactly one physical node, and each virtual link is mapped onto a loop-free path $p$ in the physical network. The mapping is called valid if all the constraints $C_j(e)$ of the virtual network are satisfied and do not violate the constraints of the physical network. Formally, the mapping is a function

$$M : G^V \rightarrow (N^P, \mathcal{P}).$$

$M$ is called a valid mapping if all constraints\(^4\) of $G^V$ are satisfied, and for each $l^v = (s^V, t^V) \in L^V$, $\exists$ a physical loop-free path $p : (s^P, \ldots, t^P) \in \mathcal{P}$ where $s^V$ is mapped to $s^P$ and $t^V$ is mapped to $t^P$.

Due to the combination of node and link constraints, the virtual network mapping problem is NP-hard. For example, assigning virtual nodes to the substrate (physical) network without violating link bandwidth constraints can be reduced to the multiway

\(^4\)Examples of node constraints include CPU, memory, physical location, whereas link constraints may be delay, jitter, or bandwidth.
separator problem which is NP-hard [Dav02].

To reduce the overall complexity, several heuristics were introduced, including backtracking algorithms [JA08, LK09], simulated annealing as in Emulab [RAL03], as well as heuristics that solve the node and link mapping independently.

The virtual network mapping task takes as input all the available resources (sub-
set of all the existing resources) \( P' \subseteq P \) and \( N' \subseteq N \), maps virtual nodes to physical nodes, virtual links to loop-free physical paths, and returns a list of candidates — virtual nodes and virtual links — to the allocator. To model this task, we define two sets of binary variables \( n_{ij}^V \forall i \in N', \) and \( l_{kj} \forall k \in P', \forall j \in J \), where \( J \) is the set of users requesting a slice. \( n_{ij}^V = 1 \) if a virtual instance of node \( i \) could possibly be mapped to user \( j \) and zero otherwise, while \( l_{kj} = 1 \) if a virtual instance of the loop-free physical path \( k \) could possibly be mapped to user \( j \), and zero otherwise.

The virtual network mapping phase of the slice embedding problem can hence be modeled by the following optimization problem:

\[
\begin{align*}
\text{maximize} & \quad g(n_{ij}^V, l_{kj}) = \sum_{j \in J}(\sum_{i \in N'} \Theta_{ij} n_{ij}^V + \sum_{k \in P'} \Phi_{kj} l_{kj}) \\
\text{subject to} & \quad \sum_{i \in N'} n_{ij}^V = \gamma_j \quad \forall j \in J \\
& \quad \sum_{k \in P'} l_{kj} = \psi_j \quad \forall j \in J \\
& \quad n_{ij}^V, n_{ij}, p_{kj}, l_{kj} \in \{0, 1\} \quad \forall i \forall j \forall k,
\end{align*}
\]

(2.2)

where \( \Theta_{ij} \) is the revenue that the system would get if user \( j \) gets assigned to virtual node \( i \), and \( \Phi_{kj} \) is the system’s revenue if the user \( j \) gets the virtual link \( k \). The first two constraints enforce that all the virtual resources requested by each user are mapped, the third constraint ensures that the one-to-one mapping between virtual and physical nodes is satisfied, and the fourth constraint ensures that at least one loop-free physical path is going to be assigned to each virtual link of the requested slice.
2.2.3 Allocation

As soon as the virtual mapping candidates have been identified, a packing problem needs to be run, considering both user priorities and physical constraints. Enhancing the level of details from the standard set packing problem [Ski97] to virtual nodes and links, we model the allocation phase of the slice embedding problem as follows:

maximize $h(y_j) = \sum_{j \in J} w_j y_j$
subject to

$\sum_{j \in J} n_i^j y_j \leq C_i^n \ \forall i \in N'$

$\sum_{j \in J} l_k^j y_j \leq C_k^l \ \forall k \in \mathcal{P}'$

$y_j \in \{0, 1\} \ \forall j$ \hspace{1cm} (2.3)

where $C_i^m$ and $C_k^l$ are the number of virtual nodes and links respectively, that can be simultaneously hosted on the physical node $i$ and physical path $k$, respectively, and $y_j$ is a binary variable equal to 1 if user $j$ has been allocated and zero otherwise. A weight $w_j$ is assigned to each user $j$, and it depends on the allocation policy used (e.g. in first-come first-serve, $w_j = w \ \forall j$, or in a priority based allocation $w_j$ represents the importance of allocating user $j$’s request). As multiple resources are typically required for an individual slice, the slice embedding needs to invoke the appropriate resource allocation methods on individual resources, and it does so throughout this last phase. Each resource type may in fact have its own allocation policy (e.g., either guaranteed or best-effort resource allocation models), and this phase only ensures that users will not be able to exceed physical limits or their authorized resource usage. For example, the system may assign a weight $w_j = 0$ to a user that has not yet been authorized, even though her virtual network could be physically mapped.

2.2.4 Complete Slice Embedding

Building on previous optimization problems, we formulate a unified centralized framework that considers the various facets of the slice embedding problem as a whole.
The framework also provides insights on understanding the interactions among such phases, and how they may impact efficiency in network virtualization. In particular, we model the three phases as follows:

\[
\begin{align*}
\text{maximize} \quad & \alpha \cdot f(n_{ij}^P, p_{kj}) + \beta \cdot g(n_{ij}^V, l_{kj}) + \delta \cdot h(y_j) \\
\text{subject to} \quad & \sum_{i \in N} n_{ij}^P \geq \gamma_j \quad \forall j \\
& \sum_{k \in P} p_{kj} \geq \psi_j \quad \forall j \\
& \sum_i n_{ij}^V = \gamma_j \quad \forall j \\
& \sum_k l_{kj} = \psi_j \quad \forall j \\
& n_{ij}^V \leq n_{ij}^P \quad \forall i \quad \forall j \\
& l_{kj} \leq p_{kj} \quad \forall k \quad \forall j \\
& \sum_{j \in J} n_{ij}^V y_j \leq C_i^n \quad \forall i \\
& \sum_{j \in J} l_{kj} y_j \leq C_k^l \quad \forall k \\
& y_j \leq n_{ij}^V \quad \forall i \quad \forall j \\
& y_j \leq l_{kj} \quad \forall k \quad \forall j \\
& y_j, n_{ij}^P, p_{kj}, n_{ij}^V, l_{kj}, \in \{0, 1\} \quad \forall \ i, j, k
\end{align*}
\]

where constraints \(a, b\) and \(e - h\) are the same as in problems (2.1), (2.2) and (2.3), constraints \(i\) and \(j\) bind the mapping and allocation phases while \(e, f\) act as binding constraints between discovery and mapping, and \(\alpha, \beta\) and \(\delta\) are normalization factors. All the above constraints have never been simultaneously considered before in related literature as we discuss in the following sections.

Table 2.1 highlights how the combination of constraints in (4) alone can classify the existing literature on slice embedding. In [YYRC08] for example, the first two as well as the last two constraints are omitted (plus \(\alpha = \delta = 0\)), and a global knowledge
Table 2.1: The different combinations of slice embedding constraints classify the related work. None of the existing solutions holistically consider discovery, mapping and allocation. For clarity of representation, we omit the binary constraints \((4k)\).

<table>
<thead>
<tr>
<th>Constraints in ((4))</th>
<th>Considered Tasks</th>
<th>Representative References</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a, b)</td>
<td>Discovery</td>
<td>MRPM08</td>
</tr>
<tr>
<td>(a - f)</td>
<td>Discovery and mapping</td>
<td>AOVVP08</td>
</tr>
<tr>
<td>(c - d)</td>
<td>Mapping</td>
<td>[ZA06, FA06] [CBMP04] [LT06, RAL03] [LK09]</td>
</tr>
<tr>
<td>(c, d, g - j)</td>
<td>Mapping and allocation</td>
<td>[CSZ+11, BFH+06] [CRB09, LK09] [YYRC08, HLBAZ11]</td>
</tr>
<tr>
<td>(g, h)</td>
<td>Allocation</td>
<td>[CBK10, ZAI0] [ABC+09, ACSV04] [BNCV05, FCC+03] [SNP+05, ZXB10]</td>
</tr>
</tbody>
</table>

of the resource availability is assumed. Other solutions that focus only on the virtual network mapping phase (for example [ZA06]), omit the capacity constraints \((g\) and \(h)\).

From an optimization theory point of view, constraint omissions in general may result in infeasible solutions while constraint additions may lead to sub-optimal solutions. For example, the resource discovery constraints impact the other phases of the slice embedding, since a physical resource not found certainly cannot be mapped or allocated. Moreover, it is useless to run the virtual network mapping phase on resources that can never be allocated because they will exceed the physical capacity constraints. As a consequence, we believe that centralized or distributed solutions for the slice embedding problem are a valuable research subarea of network virtualization. In what follows, we classify existing solutions and highlight which aspect(s) of the above abstract formulation they do address (or ignore.)
2.3 Taxonomy

To dissect the space of existing solutions spanning the slice embedding tasks, as well as interactions among them, we consider three dimensions as shown in Figure 2·3: the type of constraint, the type of dynamics, and the resource allocation approach.

2.3.1 Constraint type

Users need to express their requests (queries) efficiently. Some constraints are on the nodes and/or links (e.g., minimum CPU requirement, average bandwidth, maximum allowed latency) while others consider inter-group [AOVP08] or geo-location constraints [CRB09].

Based on this dimension, research work in this area assumes no constraints [ZA06], considers constraints on nodes only [PACR03], links only [LT06, HZsL+08], or on both nodes and links [ACSV04, YYRC08, RAL03, NMCS11]. In addition, the order in which the constraints are satisfied is important as pointed out in [LK09]: satisfy the node constraints and then the link constraints [ZA06, YYRC08], or satisfy both constraints simultaneously [JA08, LK09].

2.3.2 Dynamics

Each task in the slice embedding problem may differ in terms of its dynamics. In the resource discovery task, the status updates of each physical resource may be collected periodically [HS03], or on demand [AOVP08].

In the virtual network mapping task, virtual resources may be statically mapped to each physical resource [ZA06], or they can move (e.g., using path migrations [YYRC08] or by re-running the mapping algorithm [FA06]) to maximize some notion of utility [HZsL+08]. Also, the mapping can focus only on one single phase at a time where each phase considers only nodes or links [ZA06, HLZ08], or simultaneously both nodes and links [LK09, CRB09].
Finally, the allocation task may be dynamic as well: users may be swapped in or out to achieve some Quality of Service (QoS) or Service Level Agreement (SLA) performance guarantees, or they can statically remain assigned to the same slice. An example of static assignment of a slice may be an infrastructure hosting a content distribution service similar to Akamai, whereas an example of dynamic reallocation could be a researcher’s experiment being swapped out from/into the Emulab testbed [WLS^+].

2.3.3 Admission Control

As the substrate—physical infrastructure—resources are limited, some requests must be rejected or postponed to avoid violating the resource guarantees for existing vir-
tual networks, or to maximize profit of the leased network resources. Some research work, however, does not consider any resource allocation [HS03, JA08, CBMP04, ZA06, LT06, LK09]. Others consider the resource allocation task, with [FCC+03] or without [LRA+05, ACSV04, YYRC08] guarantees to the user, i.e., the resource allocation mechanism enforces admission to the users, or it only implements a tentative admission, respectively. An example of tentative admission is a system that issues tickets, without guarantee that those tickets can be exchanged with a resource later in time. The literature defines those tentative admission mechanisms that do not provide hard guarantees as soft reservation [FCC+03].

2.4 Resource Discovery

Although researchers have developed, and in some cases deployed a number of resource discovery solutions for wide-area distributed systems, the research in this area still has many open problems. Some of the existing distributed systems provide resource discovery through a centralized architecture, see, e.g., Condor [LLM88] or Assign [RAL03]; others use a hierarchical architecture such as Ganglia [MCC03], while XenoSearch [SH03], SWORD [OAPV05] and iPlane Nano [MKbA+09] employ a decentralized architecture.

All of these systems allow users to find nodes that meet per-node constraints, except iPlane Nano that considers path metrics, while SWORD, and Assign also consider network topologies. Unfortunately, none of these solutions analyze the resource discovery problem when the queried resources belong to multiple infrastructure or service providers. To obtain an efficient slice embedding, such cases would in fact require some level of cooperation (e.g., by sharing some state), and such incentives to cooperate may be scarce.

As mentioned previously, we do not discuss solutions that address the resource discovery task in isolation, since it is not different from classical resource discovery
in the distributed systems literature. Instead, we consider the resource discovery problem in combination with either the allocation or the network mapping task.

2.4.1 Resource Description

A related problem to resource discovery is the resource description problem (see e.g. [vdHGvdP+07, RSp, BXM+10, CBMP04] and references therein), where expressive languages (interfaces) to publish and search for resources as well as data structures for organizing such information are defined. These solutions focus either on how to specify the desired functionalities, e.g. what operating system should be installed on a virtual machine [RSp, Wor06, BNCV05, LLM88, FCC+03], or on how to define the requested performance of a service (slice) [Org06, OAPV05].

RSpec [RSp] for example is an XML-based language that allows ProtoGENI [Glo09] users to describe the resource topology and their constraints. Other (centralized [ACSV04] or distributed [OAPV05, LLM88, FCC+03]) systems include a resource description language, whether they focus on discovery and mapping [OAPV05, LLM88] or on the allocation [ACSV04], phases.

Regardless of the type of description language, its purpose is to describe the input to a subsequent slice embedding task, whether it is a resource discovery [OAPV05], or (in case e.g. [ACSV04, BNCV05]) allocation. A complete overview of description languages is outside the scope of this chapter, but few existing solutions applicable to network virtualization as well as their limitations are described in [DL07].

2.4.2 Discovery and virtual network mapping

We present SWORD [AOVP08], a system that considers the interaction between the resource discovery and the virtual network mapping tasks. SWORD is a resource discovery infrastructure for shared wide-area platforms such as PlanetLab [PACR03]. We choose to describe SWORD as it is a well known network discovery system whose
source code is available [SWO05]. The system has been running on PlanetLab for several years. Some of the functionalities described in the original paper, however, are currently disabled. For example, the current implementation of SWORD runs in centralized mode, and inter-node and group requirements (i.e., constraints on links and set of nodes, respectively), are not supported because no latency or bandwidth estimates are available.

Users wishing to find physical nodes for their application submit a resource request expressed as a topology of interconnected groups. A group is an equivalence class of nodes with the same per-node requirements (e.g., free physical memory) and the same inter-node requirements (e.g., inter-node latency). Supported topological constraints within and among groups include the required bandwidth and latency.

In addition to specifying absolute requirements, users can supply SWORD with per-attribute penalty functions, that map the value of an attribute (feature of a resource, such as load or delay) within the required range but outside an ideal range, to an abstract penalty value. This capability allows SWORD to rank the quality of the configurations that meet the applications’ requirements, according to the relative importance of each attribute. Notice that these penalty values would be passed to the allocation together with the list of candidates.

Architecturally, SWORD consists of a distributed query processor and an optimizer which can be viewed as a virtual network mapper. The distributed query processor uses multi-attribute range search built on top of a peer-to-peer network to retrieve the names and attribute values of the nodes that meet the requirements specified in the user’s query. SWORD’s optimizer then attempts to find the lowest-penalty assignment of platform nodes (that were retrieved by the distributed query processor) to groups in the user’s query—that is, the lowest-penalty embedding of the requested topology in the PlanetLab node topology, where the penalty of an embedding is defined as the sum of the per-node, inter-node, and inter-group penalties.
associated with that selection of nodes.

Due to the interaction between the distributed query processor (resource discovery task) and the optimizer (mapping task), SWORD is more than a pure resource discoverer. SWORD provides resource discovery, solves the network mapping task, but does not provide resource allocation. Formally, this means that only the first six constraints \((a - f)\) in (4) are considered, and the final allocation task is left to the user \((\delta = 0)\).

In particular, since PlanetLab does not currently support resource guarantees, a set of resources that SWORD returns to a user may no longer meet the resource request at some future point in time. In light of this fact, SWORD supports a continuous query mechanism where a user’s resource request is continually re-matched to the characteristics of the available resources, and in turn a new set of nodes are returned to the user. The user can then choose to migrate one or more instances of their application. This process is all part of the general feedback system outlined in Figure 2.1.

2.5 Virtual Network Mapping

The virtual network mapping is the central phase of the slice embedding problem. In this section we survey solutions that focus only on this task, as well as solutions that cover interactions with the other two tasks of the slice embedding problem.

2.5.1 Network mapping without constraints

The problem of static assignments of resources to a virtual network has been investigated in [ZA06]. Since it is NP-hard, the authors proposed a heuristic to select physical nodes with lower stress \((i.e.,\) with the lower number of virtual nodes already assigned to a given physical node), in an attempt to balance the load. The algorithm consists of two separate phases: node mapping and link mapping. The
node mapping phase consists of an initialization step —cluster center localization— and an iterative subroutine —substrate node selection— that progressively selects the next physical node $u'$ to which the next virtual node is mapped, i.e. the physical node with the least stress.

In particular, the center cluster is selected as follows:

$$u' = \arg \max_v \left\{ [S_{n\text{max}} - S_N(v)] \sum_{l \in L(v)} [S_{l\text{max}} - S_L(l)] \right\}$$

where $S_{n\text{max}}$ and $S_{l\text{max}}$ are the maximum node and link stress seen so far in the physical network, respectively. $S_N(v)$ is the stress on the physical node $v$, while $S_L(l)$ is the stress on the physical link $l$. $[S_{n\text{max}} - S_N(v)]$ captures the availability of node $v$, while the availability on the links adjacent to $v$ is captured by $\sum_{l \in L(v)} [S_{l\text{max}} - S_L(l)]$.

The substrate node selection subroutine maps the remaining virtual nodes by minimizing a potential function proportional to both node and link stress on the physical network, i.e.:

$$u' = \arg \min_v \frac{\sum_{u \in V_A} D(v, u)}{S_{n\text{max}} - S_N(v) + \epsilon}$$

where $V_A$ is the set of already selected substrate nodes, $v$ is an index over all physical nodes (so $v$ could be the same as some $u$), $\epsilon$ is a small constant to avoid division by zero, and $D$ is the distance between any two physical nodes $v$ and $u$ and it is defined as:

$$D(v, u) = \min_{p \in \mathcal{P}(u, v)} \frac{1}{\sum_{l \in p} S_{l\text{max}} - S_L(l) + \epsilon}$$

where $p$ is an element of all loop-free paths $\mathcal{P}(u, v)$ on the physical network that connects nodes $u$ and $v$. The node mapping phase successfully terminates when all the virtual nodes are mapped.

The link mapping invokes a shortest path algorithm to find a minimum hop
(loop-free) physical path connecting any pair of virtual nodes.

In the same paper, the authors modify this algorithm by subdividing the complete topology of a virtual network into smaller star topologies. These sub-topologies can more readily fit into regions of low stress in the physical network.

Note how the two functions $u'$ and $D(u, v)$ correspond in (4) to $g(\cdot)$ for virtual nodes and virtual link mapping, respectively, and so $\alpha = \delta = 0$, and that the heuristic only considers constraints $c - d$.

### 2.5.2 Network mapping with constraints

Many of the solutions to the virtual network mapping problem consider some constraints in the query specification. Lu and Turner [LT06] for example, introduce flow constraints in the mapping of a single virtual network. The NP-hard mapping problem is solved by greedily finding a backbone-star topology of physical nodes (if it exists, otherwise the slice cannot be embedded), and the choice is refined iteratively by minimizing a notion of cost associated with the candidate topologies. The cost metric of a virtual link is proportional to the product of its capacity and its physical length. No guarantees on the convergence to an optimal topology mapping are provided, and only bandwidth constraints are imposed.

A novel outlook on the virtual network mapping problem for virtual network testbeds is considered in [CBMP04]. A topology and a set of (upper and lower bound) constraints on the physical resources are given, and a feasible mapping is sought. In order to reduce the search space of the NP-hard problem, a depth-first search with pruning as soon as a mapping becomes infeasible is used.

Another solution that considers embedding with constraints is presented in [LK09]. The authors propose a backtracking algorithm based on a subgraph isomorphism search method [LPCV01], that maps nodes and links simultaneously. The advantage of a single step node-link approach is that link constraints are taken into account
at each step of the node mapping, therefore when a bad decision is detected, it can be adjusted by backtracking to the last valid mapping. With a two-stage approach instead, the remapping would have to be done for all the nodes, which is computationally expensive.

Note that even though all these mapping solutions add (for example flow) constraints with respect to the solution described in Section 2.5.1, their constraints can all be captured within $c - d$ of our problem (4).

2.5.3 Network mapping + allocation

In all the solutions that focus only on the virtual network mapping task, only a single virtual network is considered (with or without constraints), and no resource allocation mechanism is provided. In case the mapping algorithm is designed for virtual network testbeds such as Emulab [WLS+] or Planetlab [PACR03], this may not be an issue except in rare cases, *e.g.*, during conference deadlines (see *e.g.*, Figure 1 in [ACSV04]). The lack of resource allocation is rather detrimental to an efficient slice embedding when the system aims to embed virtual networks (slices) that are profitable to the leasing infrastructure.

We discuss the case study of [YYRC08], that adds resource allocation to the virtual network mapping task, and hence introduces cooperation between the last two tasks of the slice embedding problem. The solution proposed in [YYRC08] is targeted specifically for infrastructure providers, as the physical resources considered—bandwidth and CPU—are assumed to be rentable. The authors define a revenue function $\Pi$ for each requested virtual network $G^V = (N^V, L^V)$ as:

$$\Pi(G^V) = \sum_{l^V \in L^V} bw_r(l^V) + \Omega \sum_{n^V \in N^V} CPU_r(n^V),$$

(2.5)

where $bw_r(l^V)$ and $CPU_r(n^V)$ are the bandwidth and the CPU requirements for the virtual link $l^V$ and the virtual node $n^V$, respectively. $L^V$ and $N^V$ are the sets
of requested virtual links and nodes, and $\Omega$ captures the price difference that the infrastructure provider may charge for CPU versus bandwidth.

The algorithm is depicted in Figure 2.4: after collecting a set of requests, a greedy node mapping algorithm with the objective of maximizing the (long term) revenue $\Pi$ is run. In particular, the algorithm consists of the following three steps:

1. First the requests are sorted by revenue $\Pi(G^V)$ so that the most profitable mapping is sought with highest priority.

2. Then the physical nodes with insufficient available CPU capacity are discarded to reduce the complexity of the search.

3. Similarly to [ZA06] (see Section 2.5.1), a virtual node is mapped on the physical node $n^P$ (if it exists) that maximizes the available resources $H$, where:

   $$H(n^P) = CPU_a(n^P) \sum_{l^P \in L(n^P)} bw_a(l^P)$$

   $CPU_a(n^P)$ and $bw_a(l^P)$ are the CPU and bandwidth available on the physical node $n^P$ and link $l^P$, respectively, and $L(n^P)$ is the set of links adjacent to $n^P$.

After the node mapping, different link mapping algorithms are presented. First, the authors propose to use a $k$-shortest path algorithm [Epp99]. The originality of this paper though, lies in the improvement of such a link assignment algorithm through two techniques: path splitting and path migration. In path splitting the virtual routers forward a fraction of the traffic through different physical paths to avoid congestion of critical physical links useful to host other virtual networks. By adopting fractional path splitting, the authors are able to make their problem tractable; the mapper is in fact able to solve a fractional multi-commodity flow problem, rather than the integer counterpart, making the problem polynomial rather than NP-complete. Path migration instead is adopted to further improve the resource utilization as it
Figure 2-4: Path splitting and migration mapping algorithm [YYRC08].

consists of a periodic link mapping re-computation with a larger set of pre-mapped virtual networks, leaving unchanged both node mapping—virtual node cannot migrate on another physical node—and the path splitting ratios—fraction of the total virtual links requested to which at least two physical loop-free paths are assigned. After the link mapping algorithm, the slice requests that could not be embedded are queued for a re-allocation attempt, and they are definitively discarded if they fail a given number of attempts.

Note how $\Pi(G^V)$ corresponds to $g(\cdot)$ of problem (4), and even though the objective function ignores both the discovery and allocation tasks ($\alpha = \delta = 0$), constraints that bind mapping and allocation ($c, d, g - j$) are considered, updating the available bandwidth and CPU capacities before the greedy node mapping algorithm attempts to embed a new request.

Inspired by [YYRC08] and by the PageRank algorithm [PBMW99], two topology-aware virtual network mapping and allocation algorithms (Random Walk MaxMatch and Random Walk Breadth First Search) have been recently proposed [CSZ+11]. The novelty, and common underlying idea of the two algorithms, is to use the same
Markov chain model used in PageRank [PBMW99] to sort both physical and virtual nodes (instead of web pages), and map the most important virtual nodes to the most important physical nodes. A physical (virtual) node is highly ranked not only if it has available (required) CPU, and its adjacent links have available (required) bandwidth (as in [YYRC08]), but also if its neighbors (recursively) have high rank.

After sorting both physical and virtual nodes, highly ranked virtual nodes are mapped to highly ranked physical nodes.

Also in [CSZ+11], before running the node sorting algorithms, virtual and physical capacity constraints are considered, that translates into considering constraints $c, d, g - j$, and with $\alpha = \delta = 0$ in problem (4).

### 2.5.4 Dynamic approaches to network mapping and allocation

As mentioned in Section 2.3.2, in the virtual network mapping task, virtual resources may be statically assigned to each physical resource, or they can be dynamically reassigned to maximize some notion of utility during the lifetime of a slice [HZsL+08, FBCB10, RAB10]. In general these mechanisms are dynamic in the sense that their policy changes over time within the same slice embedding attempt or across subsequent attempts. Such policies may refer to available resources, current load on the system, or the type of requests (e.g. delay-sensitive versus bandwidth-sensitive slices.)

Many algorithms whose task is simply to discover feasible mappings are considered static, whether they use simulated annealing [RAL03], genetic algorithms [WLS+], or backtrack heuristics [JA08, LK09]. A static resource assignment for multiple virtual networks though, especially when each virtual network needs to be customized to a particular application, can lead to lower performance and under utilization of the physical resources. Being aware of such inefficiencies, adaptive mechanisms to re-allocate physical resources, on demand or periodically, have been
Zan and Ammar [ZA06] have proposed (in the same paper) also a dynamic version of their mapping algorithm, in which critical nodes and links in the physical network are periodically identified. Their algorithm is dynamic in the sense that, in order to balance the load on physical nodes, the metric to select the physical nodes to embed the slice alternates between node and link stress, as the available capacity changes with the arrival and life-time of the requested slices. In particular, to evaluate the current stress levels $S_N$ and $S_L$ for physical nodes and links, two metrics are defined: the node and link stress ratio ($R_N$ and $R_L$). The former is the ratio between the maximum node stress and the average node stress across the whole physical network, while the latter is the ratio between the maximum link stress and the average link stress. Formally:

$$R_N = \frac{\max_{v \in N^P} S_N(v)}{\sum_{v \in N^P} S_N(v) / |N^P|}$$

$$R_L = \frac{\max_{l \in L^P} S_L(l)}{\sum_{v \in L^P} S_L(l) / |L^P|}$$

where $N^P$ and $L^P$ are the set of physical nodes and edges of the hosting infrastructure, respectively. $R_N$ and $R_L$ are periodically compared, and new requests are mapped optimizing the node stress if $R_N > R_L$, or the link stress if $R_N < R_L$. This process is iterated with the aim of minimizing the stress across the entire physical network.

Dynamic mapping approaches also include the solutions proposed in [LT06], since virtual links are iteratively reassigned, and in [YYRC08], due to the migration operations. Although without any considerations to the node constraints, also in [FA06] the authors consider a dynamic topology mapping for virtual networks.

A solution to the dynamic network mapping problem that uses optimization theory was presented in the DaVinci architecture—Dynamically Adaptive Virtual
Networks for a Customized Internet [HZsL+08]. A physical network with $n_0$ virtual mapped networks is considered. Each virtual network $k = 1, \ldots, n_0$ runs a distributed protocol to maximize its own performance objective function $U^k(\cdot)$, assumed to be convex with respect to network parameters, efficiently utilizing the resources assigned to it. These objective functions, assumed to be known to a centralized authority, may vary with the traffic class (e.g., delay-sensitive traffic may wish to choose paths with low propagation-delay and keep the queues small to reduce queuing delay, while throughput-sensitive traffic may wish to maximize aggregate user utility, as a function of rate), and may depend on both virtual path (flow) rates $z^{(k)}$ and the bandwidth share $y^{(k)}$ of virtual network $k$ over every physical link $l$.

The traffic-management protocols running in each virtual network are envisioned as the solution to the following optimization problem:

\[
\begin{align*}
\text{maximize} & \quad U^{(k)}(z^{(k)}, y^{(k)}) \\
\text{subject to} & \quad C^{(k)} z^{(k)} \leq y^{(k)} \\
& \quad g^{(k)}(z^{(k)}) \leq 0 \\
& \quad z^{(k)} \geq 0 \\
\text{variables} & \quad z^{(k)}, y^{(k)} \quad \forall k
\end{align*}
\]

(2.6)

where $z^{(k)}$ are the variables (virtual path rates), $g^{(k)}(z^{(k)})$ are general convex constraints, and $C^{(k)}$ defines the mapping of virtual paths over physical links. This means that there could be many flows on a single virtual network, i.e., a virtual network $k$ may host (allocate) multiple services. In particular, $c_{lj}^{(k)} = 1$ if virtual path $j$ in virtual network $k$ uses the physical link $l$ and 0 otherwise. \(^5\)

The dynamism of this approach lies in the periodic bandwidth reassignment among the $n_0$ hosted virtual networks. The physical network in fact runs another (convex) optimization problem, whose objective is to maximize the aggregate utility of all the virtual networks, subject to some convex constraints:

---

\(^5\)As in [ISLA10], a system may in fact be hosted on a physical infrastructure by leasing a slice, and then provide other services by (recursively) hosting other slices.
\[
\text{maximize } \sum_k w^{(k)} U^{(k)}(z^{(k)}, y^{(k)}) \\
\text{subject to } C^{(k)} z^{(k)} \leq y^{(k)} \forall k \\
\sum_k y^{(k)} \leq D \\
g^{(k)}(z^{(k)}) \leq 0 \forall k \\
\sum_k y^{(k)} \leq \mathcal{D} \\
z^{(k)} \geq 0 \forall k \\
y^{(k)} \geq 0 \forall k
\] (2.7)

where \( w^{(k)} \) is a weight (or priority) that a centralized authority in charge of embedding the slices assigns to each virtual network, and \( \mathcal{D} \) represents the physical capacities. Note how there are two levels of resource allocation in this model: each slice maximizes its utility by assigning capacity to each service hosted, and the physical network maximizes its utility by assigning resources to some slices.

As in [YYRC08], the DaVinci architecture allows (virtual) path splitting, causing packet reordering problems, and assumes the node mapping to be given. A more significant assumption is that physical links are aware of the performance objectives of all the virtual networks, which may not be possible in existing systems.

### 2.5.5 Distributed Virtual Network Mapping Solutions

All the previously discussed solutions assumed a centralized entity that would coordinate the mapping assignment. In other words, their solutions are limited to the intra-domain virtual network mapping. These solutions are well suited for enterprises serving slices to their customers by using only their private resources. However, when a service must be provisioned using resources across multiple provider domains, the assumption of a complete knowledge of the substrate network becomes invalid, and another set of interesting research challenges arises.

It is well known that providers are not happy to share traffic matrices or topology information, useful for accomplishing an efficient distributed virtual network mapping. As a result, existing embedding algorithms that assume complete knowledge of the substrate network are not applicable in this scenario.
Some solutions on how to stitch different physical nodes and links belonging to different providers into a single slice, rely on a centralized authority that partitions the slice and orchestrates the mapping [ZZSRR08, HLBAZ11, XBM+11], while others do not require such orchestration and hence we classify them as fully distributed.

To the best of our knowledge, the first fully distributed virtual network mapping problem was devised by Houidi et al. [HLZ08]. The protocol assumes that all the requests are hub-spoke topologies, and runs concurrently three distributed algorithms at each substrate node: a capacity-node-sorting algorithm, a shortest path tree algorithm, and a main mapping algorithm. The first two are periodically executed to provide up-to-date information on physical node and link capacities to the main mapping.

For every element mapped, there has to be a trigger and a synchronization phase across all the physical nodes. The algorithm is composed of two phases: when all virtual nodes (hubs) are mapped, a shortest path algorithm is run to map the virtual links (spokes). The authors propose the use of an external signalling/control network to alleviate the problem of the heavy overhead.

In [CRB09], the authors proposed a simultaneous node and link distributed mapping algorithms. In order to coordinate the node and the link mapping phases, the distributed mapping algorithm is run on the physical topology augmented with some additional logical elements (meta node and meta links) associated with the location of the physical resource.

In [CSB10], the same authors describe a similar distributed (policy-based) inter-domain mapping protocol, based on the geographic location of the physical resources: PolyViNE. Each network provider keeps track of the location information of their own substrate nodes by employing a hierarchical addressing scheme, and advertising availability and price information to its neighbors via a Location Awareness Protocol (LAP) — a hybrid gossiping and publish/subscribe protocol. Gossiping is used
to disseminate information in a neighborhood of a network provider and pub/sub is employed so a provider could subscribe to other providers which are not in its neighborhood. PolyViNE also propose a reputation metric to cope with the lack of truthfulness in disseminating the information with the LAP protocol.

Other distributed solutions rely on auctions [ZXB10, EDM13] among physical resource owners: in V-Mart [ZXB10], infrastructure providers submit their bids on a subset of the slice to the auctioneer that repeats the auctions for a second round to a selected set of infrastructure providers. V-Mart ensures a fair market but does not guarantee performance, in terms of providers’ utilities, of the NP-hard auction winner determination algorithm. In Chapter 4 we describe our consensus-based algorithm (that can be instantiated as an auction) that guarantees convergence and approximation bounds on the optimality of the embedding.

2.6 Allocation

Different strategies have been proposed when allocating physical resources to independent parties. Some solutions prefer practicality to efficiency, and adopt best effort approaches, (see, e.g., PlanetLab [PACR03]), while others let the (selfish) users decide the allocation outcome with a game [JLT09, ISLA10, ISBA12]. When instead it is the system that enforces the allocation, it can do it with [FCC+03] or without [ACSV04] providing guarantees. In the remainder of this section we focus first on the game theoretic solutions to resource allocation, and then on the latter case, describing first a set of solutions dealing with market-based mechanisms [ACSV04, LRA+05, BNCV05], and then a reservation-based approach [FCC+03]. All those solutions focus solely on the standalone allocation task of the slice embedding problem.
2.6.1 Game-theory based allocation

Londoño et al. [JLT09] defined a general pure-strategies colocation game which allows users to decide on the allocation of their requests. In their setting, customer interactions are driven by the rational behavior of users, who are free to relocate and choose whatever is best for their own interests. Under their model, a slice consists of a single node in a graph that needs to be assigned to a single resource. They define a cost function for user $i$ when mapped to resource $j$ to be $\frac{\omega_{ij}}{U_j}$, where $\omega_{ij}$ is the weight (or utilization) imposed on resource $j$ by user $i$, $U_j$ is the overall utilization of resource $j$, which must satisfy the resources capacity constraint.

They define a rational “move” of user $i$ from resource $a$ to resource $b$ if its cost decreases as she moves from $a$ to $b$. The game terminates when no user has a move that minimizes her cost. Note that the utility of a user (player) is higher if she can move to a more “loaded” resource, as she will share the cost with the other players hosted on the same resource.

The model has two interesting properties. First, the interaction among customers competing for resources leads to a Nash Equilibrium (NE), i.e. a state where no customer in the system has incentive to relocate. Second, it has been shown that the Price of Anarchy—the ratio between the overall cost of all customers under the worst-case NE and that cost under a socially optimal solution— is bounded by $3/2$ and by 2 for homogeneous and heterogeneous resources, respectively. The authors also provide a generalized version of this game (General Colocation Game), in which resources to be allocated are graphs representing the set of virtual resources and underlying relationships that are necessary to support a specific user application or task. In this general case however, the equilibrium results no longer hold as the existence of a NE is not always guaranteed.

The work by Chen and Roughgarden [CR09] also introduces a game theoretical
approach to link allocation in the form of source-destination flows on a shared network. Each flow has a weight and the cost of the link is split in proportion to the ratio between the weight of a flow and the total weights of all the flows sharing the physical link.

As shown by Chowdhury [CRB09] in a centralized solution, the virtual network mapping problem can be thought of as a flow allocation problem where the virtual network is a flow to be allocated on a physical network.

These two game theoretic approaches may serve as inspiring example for new allocation strategies involving different selfish principles for virtual service provisioning / competition. A system may in fact let the users play a game in which the set of strategies represent the set of different virtual networks to collocate with, in order to share the infrastructure provider costs.

2.6.2 Reservation-based allocation

As the last piece of this section on allocation approaches, we discuss a reservation-based system, SHARP [FCC+03] whose architecture is depicted in Figure 2.5. The system introduces a level of indirection between the user and the centralized authority responsible for authentication and for building the slice: the broker or agent. The authority issues a number of tickets to a number of brokers (usually many brokers responsible for a subset of resources are connected). Users then ask and eventually get tickets, and later in time, they redeem their tickets to the authority that does the final slice assignment (Figure 2.5).

This approach has many interesting properties but it may lead to undesirable effects. For example, coexisting brokers are allowed to split the resources: whoever has more requests should be responsible for a bigger fraction of them. This sharing of responsibilities may bring fragmentation problems as resources become divided into many small pieces over time. Fragmentation of the resources is a weakness, as the
resources become effectively unusable being divided into pieces that are too small to satisfy the current demands.

One of the most relevant contributions of SHARP [FCC+03] in the context of the slice embedding problem, is the rule of the *Oversubscription Degree (OD)*. The OD is defined as the ratio between the amount of issued tickets and the number of available resources. When OD is greater than one, i.e., there are more tickets than actual available resources, the user has a probability less than one to be allocated even though she owns a ticket. When instead OD is less or equal than one, users with tickets have guaranteed allocation (Figure 2-6).

Note how the level of guarantees changes with OD. In particular, when the number of tickets issued by the authority increases, the level of guarantees decreases. The authors say that the allocation policy tends to a first come first serve for OD that tends to infinity. In other words, if there are infinite tickets, there is no reservation at
The oversubscription degree is not only useful to control the level of guarantees (by issuing less tickets than available resources the damage from resource loss if an agent fails or becomes unreachable is limited), but it can be used also to improve resource utilization by means of statistically multiplexing the available resources. Note that all the above described allocation mechanisms can be mapped to problem (4) where all the constraints are ignored with the exception of 4g and 4h, and that $\alpha = \beta = 0$ in the utility function to maximize.

### 2.6.3 Market-based allocation and guarantees

When demand exceeds supply and so not all needs can be met, virtualization systems’ goals can no longer be related to maximizing utilization, and different policies to guide resource allocation decisions have to be designed. A natural policy is to seek efficiency, namely, to allocate resources to the set of users that bring to the system the highest utility. To such an extent, the research community has frequently proposed market-based mechanisms to allocate resources among competing interests while maximizing the overall utility of the users. A subclass of solutions dealing with this type of allocation is represented by auction-based systems. An auction is the process of buying and selling goods or services by offering them up for bid, taking bids, and then selling them to the highest bidder.
Few examples where auctions have been adopted in virtualization-oriented systems are Bellagio [ACSV04], Tycoon [LRA+05], and Mirage [BNCV05]. They use a combinatorial auction mechanism with the goal of maximizing a social utility (the sum of the utilities for the users who get their resources allocated).

A Combinatorial Auction Problem (CAP) is equivalent to a Set Packing Problem (SPP), a well studied integer program: given a set \( O \) of elements and a collection \( Q \) of subsets of these elements, with non-negative weights, SPP is the problem of finding the largest weight collection of subsets that are pairwise disjoint. This problem can be formulated as an integer program as follows: we let \( y_j = 1 \) if the \( j^{th} \) set in \( Q \) with weight \( w_j \) is selected and \( y_j = 0 \), otherwise. Then we let \( a_{ij} = 1 \) if the \( j^{th} \) set in \( Q \) contains element \( i \in O \) and zero otherwise. If we assume also that there are \( b_i \) copies of the same element \( i \), then we have:

\[
\begin{align*}
\text{maximize} & \quad \sum_{j \in Q} w_j y_j \\
\text{subject to} & \quad \sum_{j \in Q} a_{ij} y_j \leq b_i \quad \forall i \in O \\
& \quad y_j = \{0, 1\} \quad \forall j \in Q
\end{align*}
\]

(2.8)

SPP is equivalent to a CAP if we think of the \( y_j \)'s as the users to be possibly allocated and requesting a subset of resources in \( O \), and \( w_j \) as the values of their bids. Note that solving a set packing problem is NP-hard [dVV03]. This means that optimal algorithms to determine the winner in an auction are also NP-hard. To deal with this complexity, many heuristics have been proposed. In [ACSV04] for example, the authors rely on a thresholding auction mechanism called SHARE [CNA+04], which uses a first-fit packing heuristic.

Another example of a system that handles the allocation for multiple users with an auction is Tycoon [LRA+05]. In Tycoon, users place bids on the different resources they need. The fraction of resource allocated to one user is her proportional share of the total bids in the system. For this reason, Tycoon’s allocation mechanism can also be considered best-effort: there are no guarantees that users will receive the
desired fraction of the resources. The bidding process is continuous in the sense that any user may modify or withdraw their bid at any point in time, and the allocation for all the users can be adjusted according to the new bid-to-total ratio.

As pointed out in [ABC+09], although market-based allocation systems can improve user satisfaction on large-scale federated infrastructures, and may lead to a social optimal resource allocation, there are few issues that should be taken into account when designing such mechanisms. In fact, the system may be exploited by users in many ways. Current auction-based resource allocation systems often employ very simple mechanisms, and there are known problems that may impact efficiency or fairness (see [ABC+09], Section 6). We report three of them here:

- **underbidding**: users know that the overall demand is low and they can drive the prices down.

- **iterative bidding**: often one shot auctions are not enough to reach optimal resource allocation but the iterations may not end by the time the allocations are needed.

- **auction sandwich attack**: occurs when users bid for resources in several time intervals. This attack gives the opportunity to deprive other users of resources they need, lowering the overall system utility.

Auction algorithms and their optimality performance have also been theoretically studied in several application domains [Ber01]. In the electronic commerce for example [DK09], truthful auction strategies are sought when multiple items are released by a centralized auctioneer, and guarantees on an equilibrium are proven to exist [LST12]. The virtual network embedding architecture described in this thesis does not need a centralized authority that clears the allocation (as an auctioneer would do), and we also prove bounds on the number of iterations to reach an equilibrium (convergence to an embedding), as a function of the physical network diameter,
and the size of the slice to allocate. Moreover, in our settings truthful strategies may not work as there is uncertainty on whether more slices, or even more virtual nodes in the same slice, are to be assigned in the future; bidders may have incentives to preserve resources for stronger future bids.

In different settings, Choi et al. [CBH09] present a decentralized auction that greedily assigns tasks to a fleet of robots. Our problem formulation allocates virtual nodes and links, and physical nodes do not move as robots do.

2.7 Distributed Embedding Solutions

To avoid restricting services within a limited single provider’s domain, distributed solutions to the slice embedding have been proposed. Some solutions rely on a centralized authority that partitions the slice and orchestrates the mapping [ZZSRR08, HLBAZ11], while others do not require such orchestration and hence we classify them as fully distributed [HLZ08].

The only (to the best of our knowledge) fully distributed embedding solution existing today [HLZ08] has discouraging discovery overhead as each mapping information is flooded to all physical nodes.

The resource discovery phase is different in PolyViNE [CSB10], where an SP sends the entire slice to a subset of trusted InPs, which can eventually map the slice partially, and forwards the residual virtual subgraph to another set of trusted InPs. The process continues and the slice is rejected if a threshold number of hops is reached before its mapping is complete. The SP does the final allocation, based on the best price among the multiple candidate mapping solutions returned by different sets of InPs. The mapping and the allocation depend on the discovery, that is, on the sequence of visited InPs and therefore the proposed heuristic in practice lead to heavy sub-optimalities or to significant overhead (in case the residual virtual network is flooded to all remaining InPs.)
Our mechanism also supports slice splitting and centralized embedding orchestration, but its bidding mechanism (thanks to the max-consensus strategy) provides a complete resource discovery relying on low overhead nearest-neighbor communications, and furthermore allocation is concurrently done.
Chapter 3

Virtual Network Embedding Applications

Applications for the virtual network embedding problem can be broadly divided into two main categories: applications for testbed embedding services, and applications for cloud computing services. In this chapter, we first describe the workflow and the entities involved in embedding a constrained virtual network (Section 3.1). We then outline few key representative applications that our policy-based virtual network embedding architecture could support if deployed, for both the research community (Section 3.2) and the business community (Section 3.3).

3.1 Virtual Network Embedding Workflow

There are several entities involved in the embedding of a constrained virtual network (Figure 3.1). The embedding process begins when a user (or an application) sends a constrained virtual network request expressed using a resource specification language. The service provider that received such request forwards such request to a single or to multiple infrastructure providers, possibly adding its own constraints. Existing solutions have also envisioned a connectivity provider [ZZSRR08], sometimes called virtual network provider [HLBAZ11], that coordinates the incoming embedding requests from multiple service providers. Such intermediate providers act as brokers
between service and infrastructure providers. The infrastructure providers then apply a centralized or distributed embedding protocol to decide which physical node hosts each requested virtual node, and which loop-free physical path(s) hosts each virtual link. After the mapping decision is made, the physical resources are to be reserved. Sometimes the service provider is in charge of deciding the final resource allocation decision, by choosing among a set of potential valid candidate mappings. The infrastructure provider merely bind the chosen resources. Regardless from the entity making the final allocation decision, such information is returned to the service provider. The service provider can then access (e.g., via SSH) the reserved virtual nodes and customize the virtual network with its own policies, installing the required software in support of the user or the virtual network application.

Examples of service providers are an enterprise renting cloud resources to offer an intrusion detection system as a service to other enterprises, or a networking researcher reserving the resources of a virtual network testbed to test her clean-slate Internet architecture. Examples of users can be another enterprise that wish to pro-
tect its own private network, or another researcher that “opts-in” a GENI virtual network experiment. Examples of infrastructure providers are the GENI aggregate managers [Glo09], or the data centers of a cloud provider, e.g. Rackspace [Rac13a].

3.2 Testbed Embedding Services

Research results founded solely on experimental simulations are often not enough to convince the networking and the distributed system communities of the validity of a novel approach. Recent virtual network testbed initiatives [Glo09, REN11] have enabled researchers to experiment with their protocols and architectures over isolated wide-area virtual networks, evolving from the PlanetLab testbed [PACR03] where slices of virtual nodes were guaranteed without virtual link isolation. Although the progress has been impressive over the past few years, obtaining realistic and reproducible experiments in a wide-area virtual network still poses several challenges.

**Virtual Network Stitching:** among these challenges, the problem of how to connect the physical resources provided by multiple infrastructure providers is still an open key architectural question for wide-area virtual network testbeds.

Within the GENI community, ethernet VLANs have been identified as the initial network technology to provide slice level inter-aggregate connections and isolation [GEN11]. However, how to select the VLAN IDs to use and inform all necessary infrastructure providers along such connection is still an open question.

Another open problem within the GENI community that could be solved by deploying our virtual network embedding architecture is the connectivity management of two virtual networks managed by different GENI aggregates via an external network. Some policies of our architecture (e.g. SAD and MAD) enable embeddings in which virtual links are hosted by loop-free physical paths where the source and destination virtual nodes are hosted by different infrastructure providers and they are only connected through an external network (e.g., Internet).
3.3 Cloud Computing Services for Enterprise Networks

The problem of virtual network stitching is not unique to the GENI community. The desire of enterprises to spread their resources across multiple cloud providers is increasing as the need to provide services that are scalable, resilient to failures, or that have geographic dispersion have increased [FFK+06, DGG+99, WGR+09, CYB+ed, NLR10, REN11].

Our current architecture enables wide-area virtual network services to be built as applications after a customized virtual network embedding. Our current testbed implementation (Chapter 6) allocates virtual network resources assigning private (IP) addresses to Linux virtual hosts and virtual switches using the Mininet API [LHM10]. By leveraging Openstack [Ope13] and OpenFlow [MAB+08] technologies, our prototype could be used to build more advanced virtual network services that require binding together virtual machines across different data centers.

In the rest of this chapter we list few common examples of the virtual network services that would be possible within a virtual network embedded using our policy-based virtual network embedding architecture.

Intrusion Detection System as a service (IDS-aaS). In a public cloud computing environment, “tenants” (cloud consumers) may not trust the security infrastructure of their cloud providers. Leveraging our embedding architecture, tenants would be able to easily protect their virtual networks by implementing their own intrusion detection systems: a monitoring application that would log suspicious network activities can be in fact implemented with a simple subscribe message to each physical hosting node. Virtual nodes where an intrusion, a misbehaving, or a misconfigured physical node has been detected can immediately migrate to another physical node by requesting a new virtual network with different (geo-location) constraints with
our *slice specification* language (see Chapter 5).

**Data Center Interconnect as a Service (DCI-aaS).** Equivalently to the virtual network stitching case for virtual network testbed services, our architecture enables the creation of a virtual network using physical resources across different data centers, that can be owned by a single, or by multiple infrastructure providers. The NP-hard virtual network embedding problem of interconnecting data centers across different domains is still an active area of research, and few distributed (single or multi-infrastructure provider) virtual network embedding solutions have been proposed [HLZ08, CYB+10, CSB10, HLBAZ11]. Our policy-based embedding architecture improves the embedding success rate of existing solutions, and provides guarantees on convergence time and bounds on optimality [EDM13].

**Virtual Private Network as a service (VPN-aaS).** To offer Virtual Private Clouds [WGR+09], cloud providers have to cope with many network service management challenges. Some of these challenges include network isolation, custom addressing, and dynamic acquisition and release of virtual resources, with different Service Level Objectives (SLOs) and policies [BASS11]. After receiving from an enterprise a virtual network request with some SLO resource constraints (*e.g.*, geographical location, bandwidth or delay), multiple cloud providers running our architecture can cooperate to set up a virtual network spanning physical resources across all their domains. After the embedding is completed, credentials for the enterprise’ external access to the VPN can be released, *e.g.*, a temporary user identifier and password to enable login into each virtual node.
Load balancing as a service (LB-aaS). Cloud providers, *e.g.* Rackspace [Rac13b], offer a load balancing virtual network service for about $11 per month per virtual instance. By dynamically adopting the CADE embedding policies, our architecture can also be configured to be a load balancer. In particular, existing virtual networks can be re-embedded using a different policy (*e.g.* SAD), having the same effect of migrating virtual nodes towards less loaded physical nodes and links. A virtual network migration can be seen as a deallocation followed by an allocation of a new virtual network, with some book-keeping to ensure no loss of ongoing data traffic [WKB+08].

Firewall as a service (Firewall-aaS). Our current architecture prototype implementation attaches an Open Virtual Switch (OVS) controller [OVS13] to each embedded virtual network. By merely inserting the appropriate OpenFlow forwarding rules in all virtual switches, our architecture can provide a virtual network with firewall rules (Firewall as a service). Examples of such rules are: drop all datagrams whose source network address belongs to a given black list.
Chapter 4

A General Distributed Approach to Slice Embedding with Guarantees

Leveraging properties from the consensus literature [Lyn96], in this chapter we propose a general Consensus-based Allocation mechanism for Distributed slice Embedding (CADE). The mechanism is general as it supports a large spectrum of applications and providers’ objectives along with their distribution models by tuning its policies. CADE iterates over an election (or voting) and an agreement (or consensus) phase to embed virtual nodes, before a third phase embeds virtual links. By only exchanging votes (or bids) and few other policy-driven information with their neighbors, physical nodes discover available resources, find a mapping solution and agree on a slice assignment.

To demonstrate its flexibility, we compare and analyze the tradeoffs between two different policy configurations of CADE (Section 4.1): the first, that we call Single Allocation for Distributed slice embedding (SAD), allows voting on a single virtual node per election round. The second, called Multiple Allocation for Distributed slice embedding (MAD), allows physical nodes to win multiple virtual resources simultaneously and therefore leads to faster slice embedding (convergence) time. Using extensive trace-driven simulations, we show the counter-intuitive result that having
full knowledge of the entire slice to be allocated before the election phase, MAD may yield lower allocation efficiency. Moreover, we show that SAD better balances the load and often has shorter response time — time to identify whether a slice can be embedded — independently from the slice (virtual) topology (Section 4.3).

It is known that distributed allocation algorithms converge to a solution if the utility function is sub-modular [KST09, CBH09]. We obtain the same convergence result relaxing the sub-modularity assumption and using the notion of pseudo-submodularity of the utility function that physical nodes use in the election phase, that is, each physical node is free to use any private function for each allocation round, and communicates its votes in a way so that they appear to be obtained using a sub-modular function. We show that independently from the utility (policy) that InPs decide to adopt, CADE has a worst-case convergence time of $D \cdot |V_H|$, where $D$ is the diameter of the physical network and $|V_H|$ the size of the slice $H$ to be embedded (Section 4.2). Under the same assumptions, we also show that CADE has a minimum performance guarantee of $(1 - e^{-1})$ relative to the optimal solution.

4.1 Consensus-based Auctions for Distributed Slice Embedding

**Problem statement.** As defined in Chapter 2, we are given a virtual network $H = (V_H, E_H, C_H)$ and a physical network $G = (V_G, E_G, C_G)$, where $V$ is a set of nodes, $E$ is a set of links, and each node or link $e \in V \cup E$ is associated with a capacity constraint $C(e)$, a virtual network (slice) mapping (or embedding) is a mapping of $H$ onto a subset of $G$, such that each virtual node is mapped onto exactly one physical node, and each virtual link is mapped onto a loop-free physical path $p$. Formally, the mapping is a function $\mathcal{M} : H \rightarrow (V_G, \mathcal{P})$ where $\mathcal{P}$ denotes the

---

1Each $C(e)$ could be a vector $\{C_1(e), \ldots, C_\gamma(e)\}$ containing different types of constraints, e.g. physical geo-location, delay or jitter.
Figure 4.1: Virtual network embedding architecture: mechanisms and interfaces.

Figure 4.2: (a) Slice with capacity constraints to be embedded. (b) Each physical node (PN) can be owned by a different InP, and can have a different capacity. (c) CADE workflow: a virtual link embedding phase follows the virtual node embedding phase.

set of all loop-free paths in $G$. $\mathcal{M}$ is called a valid mapping if all constraints of $H$ are satisfied, and for each $i^H = (s^H, r^H) \in E_H$, $\exists$ at least one physical loop-free path $p: (s^G, \ldots, r^G) \in \mathcal{P}$ where $s^H$ is mapped to $s^G$ and $r^H$ is mapped to $r^G$.

**Objective:** multiple valid mappings of $H$ over $G$ may exist; each physical node $i$ has a utility function $U_i$. We are interested in finding in a distributed fashion the embedding solution that maximizes the sum of the utilities of all providers $\sum_{i \in V_G} U_i$, e.g., by letting InPs instantiate policies according to their goals and run the auction. A natural objective for an embedding algorithm is to maximize revenue. The revenue can be defined in various ways according to economic models. As in [YYRC08], we use the notion of a higher economic benefit (reward) from accepting a slice or virtual request that requires more resources (e.g., bandwidth, CPU) from the physical network.
4.1.1 CADE mechanism

Consider a slice embedding request by an SP (Figure 4-2a) on a physical network (Figure 4-2b) where each physical node (PN) belongs to a different InP. The SP sends to (a subset of) all physical nodes a request with (a subset of) the virtual elements (nodes and links), e.g. virtual nodes VN1 and VN2 connected by virtual link VL1. Each physical node $i$, where $i \in V_G$, uses a private utility function $U_i \in \mathbb{R}_{+}^{|V_H|}$ to bid on the virtual nodes, knowing that it could be the winner of a subset (for example VN1 or VN2 or both), and stores its bids in a vector $b_i \in \mathbb{R}_{+}^{V_H}$. Each entry $b_{ij} \in b_i$ is a positive real number representing the highest vote (or bid) known so far on virtual node $j \in V_H$. Also, physical nodes store the identifiers of the virtual nodes that they are attempting to host in a list (bundle vector) $m_i \in V_H^T$, where $T_i$ is a target number of virtual nodes mappable on $i$. After the private voting phase, each physical node exchanges the votes with its neighbors, updating an assignment vector $a_i \in V_G^{V_H}$ with the latest information on the current assignment of all virtual nodes, for a distributed winner determination.

The winner physical nodes communicate the mapping to the SP which, if possible, releases the next slice(s) or the next slice partition if any (e.g. VN3, VN4, VL3 in Figure 4-2a). Once the physical nodes have reached consensus on who is the winner for all the virtual nodes of the (partial or full) slice released, a distributed link embedding phase is run to embed each virtual link on a set of (one or many) loop-free physical paths (Figure 4-2c). The mechanism iterates over multiple node voting and agreement (consensus) phases synchronously, that is, the second voting phase does not start until the first agreement phase terminates. Physical nodes act upon messages received at different times during each bidding phase and each consensus phase; therefore, each individual phase is asynchronous. In the rest of the

---

2The slice partitioning problem has been shown to be NP-hard, e.g. in [HLBAZ11] and it is outside the scope of this paper.
paper, we denote such rounds or iterations of node bidding followed by consensus with the letter $t$ and we omit $t$ when it is clear from the context.

Adapting the definition of max-consensus from the consensus literature [Lyn96] to the slice embedding problem we have:

**Definition 3.** (Max-consensus) Given a physical network $G$, an initial bid vector of physical nodes $b(0) \triangleq (b_1(0), \ldots, b_{|V_G|}(0))^T$, a set of neighbors $N_i \forall i \in V_G$, and the consensus algorithm for the communication instance $t+1$:

$$b_i(t+1) = \max_{j \in N_i \cup \{i\}} \{b_j(t)\} \ \forall i \in V_G,$$

(4.1)

Max-consensus on the bids among the physical nodes is said to be achieved with convergence time $l$, if $\exists l \in \mathbb{N}$ such that $\forall t \geq l$ and $\forall i, i' \in V_G$,

$$b_i(t) = b_{i'}(t) = \max\{b_1(0), \ldots, b_{|V_G|}(0)\},$$

(4.2)

where $\max\{\cdot\}$ is the component-wise maximum.

**Assumptions:** we assume that physical nodes are aware of the physical outgoing link capacity to reach each of its first-hop neighbors to propagate the highest bids, the routing table for the path embedding phase, and the diameter $D$ of the physical network, useful as a termination condition: if a physical node has received more than $D$ messages the voting phase terminates. \footnote{Algorithms to compute the diameter of a network in a distributed way are well known [Lyn96], and they are outside the scope of this paper.}

### 4.1.2 CADE Policies

one of the design goals of CADE is its flexibility — ability to create customizable slice embedding algorithms to satisfy desired policies, rules, and conditions. We describe here such policies, and later in this section we show few examples of how they can be instantiated to satisfy other goals. A straightforward example of policy is the (normalized) utility function $U$ that InPs use to allocate virtual resources (node,
links, or both). In our evaluation (Section 4.3), the bid value of physical node $i$ on virtual node $j$ is equivalent to $U_{ij}$, where:

$$
T_i = C_i + \sum_{k \in N_i} C_{ik}, \quad U_{ij} = \frac{T_i - S_{ij}}{T_i}
$$

where $T_i$ is the target virtual (node and links) capacity that is allocatable on $i$, and $S_{ij}$ the stress on physical node $i$, namely, the sum of the virtual node capacity already allocated on $i$, including virtual node $j$ on which $i$ is bidding, plus the capacity of the virtual links allocated on the adjacent physical links. Note that, due to the max consensus definition, $b_{ij}$ at physical node $i$ on virtual node $j$ is the maximum value seen so far. The normalization factor $\frac{1}{T_i}$ ensures that the utility values are comparable across physical nodes.

We have seen from related work, e.g. [ZZSRR08, HLZ08], how embedding protocols may require SPs to split the slice. CADE is able to express this requirement by enforcing a limit on the length of the vector $b_i$, so that physical nodes attempt to host only virtual nodes within the released slice partition. Each InP can also enforce a load target on its resources by limiting its target allocatable capacity $T_i$, which, in turn, limits its bundle size $T_i$.

Another policy is the assignment vector $a_i$, that is, a vector that keeps track of the current assignment of virtual nodes. $a_i$ may assume two forms: least and most informative. In its least informative form, $a_i \equiv x_i$ is a binary vector where $x_{ij}$ is equal to one if physical node $i$ hosts virtual node $j$ and 0 otherwise. In its most informative form, $a_i \equiv w_i$ is a vector of physical nodes that are far winning the hosting of virtual nodes; $w_{ij}$ represents the identifier of the physical node that had the highest utility so far to host virtual node $j$. Note that when $a_i \equiv w_i$ the assignment vector reveals information on which physical nodes are so far the winners of the mapping phase, whereas if $a_i \equiv x_i$ physical node $i$ only knows if it is winning
Procedure 1 CADE electionPhase for physical node $i$ at iteration $t$

1: **Input:** $a_i(t - 1), b_i(t - 1)$
2: **Output:** $a_i(t), b_i(t), m_i(t)$
3: $a_i(t) = a_i(t - 1), b_i(t) = b_i(t - 1), m_i(t) = \emptyset$
4: if utilityIsNeeded($a_i(t), T_i$) then
5: if $\exists j: h_{ij} = \mathbb{I}(U_{ij}(t) > b_{ij}(t)) == 1$ then
6: $\eta = \arg\max_{j \in V_H} \{h_{ij} \cdot U_{ij}\}$
7: $m_i(t) = m_i(t) \oplus \eta$ // append $\eta$ to bundle
8: $b_{i\eta}(t) = U_{i\eta}(t)$
9: update($\eta, a_i(t)$)
10: Send / Receive $b_i$ to / from $k \ \forall k \in \mathcal{N}_i$
11: if $a_i \equiv w_i$ then
12: Send / Receive $w_i$ to / from $k \ \forall k \in \mathcal{N}_i$
13: end if
14: end if
15: end if

each virtual node or not. As a direct consequence of the max-consensus, this implies that when the assignment (allocation) vector is in its least informative form, each physical node only knows the value of the maximum utility value so far without knowing the identity of the physical node whom produced such value. We also leave as a policy whether the assignment vector is exchanged with the neighbors or not. In case all physical nodes know about the assignment vector of the virtual nodes, such information may be used to allocate virtual links in a distributed fashion. Instead, if $a_i \equiv x_i$, to avoid physical nodes flooding their assignment information, $i$ asks the SP about the identity of the physical node hosting the other end of the virtual link and attempts to allocate at least one loop-free physical path.

4.1.3 Phase 1: CADE Election Phase

Consider procedure 1: after the initialization of the assignment vector $a_i$, the utility vector $b_i$ and the bundle vector $m_i$ for the current iteration $t$ (line 3) \footnote{We elaborate on the need to reset $m_i$ at the end of Remark 2, Section 4.1.5.}, each physical node checks if another election phase is needed (line 4), for example because...
there is enough capacity or because the embedding policy allows another vote, or else terminates. If a physical node has a positive utility value, but cannot overcome any utility, the election phase terminates. If instead there is at least one virtual node $j$ such that $U_{ij}(t) > b_{ij}$ (line 5), physical node $i$ registers in its utility vector the value with the highest reward $\eta = \arg\max_{j \in V_H} \{ h_{ij} \cdot U_{ij}\}$ (line 6) and updates the state vectors (lines 7–9). At the end of the election phase, the current winning utility vector (line 10) and if the embedding policy allows it (lines 11–13), the assignment vector are exchanged with each neighbor. Depending on the configured policies, the functions utilityIsNeeded() and update() of Procedure 1 may behave differently.

**SAD configuration:** in particular, let us consider a scenario in which InPs (1) wish to reveal the least possible information to other (competitor) InPs, and (2) they are interested in the quickest possible response time on a slice request. To accommodate these goals, we set the assignment vector policy to its least informative form, the partition size to two (so that a slice is rejected as soon as one of the two virtual nodes or their adjacent virtual link is not allocatable), and the bundle vector size to one, so that the auction is on a single item. As we are forcing physical nodes to bid on a single virtual node per auction round, we refer in the rest of the paper to this policy configuration as Single Allocation for Distributed slice embedding (SAD).

**SAD election:** given such policy configuration, the utilityIsNeeded() function can be implemented by only verifying if $A(t) = \sum_{j \in V_H} x_{ij}(t) = 0$, knowing that physical nodes are only allowed to win one virtual node per round “$t$”, that is, $A(t) \leq 1$. Given the SAD policy configuration, the update() function implementation simply changes the assignment vector from $x_{i\eta}(t) = 0$ to $x_{i\eta}(t) = 1$.

---

$^{5}$\(\Pi(\cdot)\) is an indicator function, unitary if the argument is true and 0 otherwise.
Example 1. (SAD election). Consider Figure 4.2: virtual nodes VN1 and VN2 are released by the SP. Assuming that all nodes use as utility their residual node capacity, PN1, PN3 and PN5’s initial utility vectors are $b_{PN1}(0) = (8, 0)$, $b_{PN3}(0) = (0, 20)$, and $b_{PN5}(0) = (0, 40)$. Note that the first utility value of each physical node is its initial capacity, and PN1 cannot have a positive utility to host VN2 since VN2 requires 9 capacity units whereas PN1’s capacity is 8. Also we assume that a physical node, whenever feasible, may have a positive utility on the virtual node with highest residual capacity as this brings higher reward (revenue). In their first election phase, physical nodes assign themselves as winners for the virtual nodes as they do not know yet each other’s bids, and so $x_{PN1} = (1, 0)$ and $x_{PN3} = x_{PN5} = (0, 1)$.

MAD configuration: let us now consider a scenario in which embedding slices with the least possible iterations (convergence time) is more desirable than hiding information from other physical nodes. To this end, we remove the limit on the number of mappable virtual nodes within the same election round, and we do not partition the slice so that each physical node has an offline knowledge of the entire slice (as opposed to SAD that releases the slice components in an online fashion, i.e. the slice embedding algorithm runs without a complete knowledge of the input). Moreover, we set the assignment vector policy to its most informative form, so that the consensus is run simultaneously on both the utility vector and on the assignment vector.

MAD election: under these settings, the function utilityIsNeeded() is implemented so that it returns true while there is still room for additional virtual resources. The amount of virtual resources that physical node $i$ is attempting to host can be expressed either in terms of the total number of virtual nodes in its current bundle $m_i(t)$, i.e. $|m_i(t)|$, or in terms of the resulting virtual capacity stress on physical node $i$ as in (4.3). Also under these settings, the update() function implementation updates the allocation vector with $w_{i,j}(t) = i$ (not just with 1 or 0 but with the identifier of the winning physical node).

Example 2. (MAD election). Let us consider Figure 4.2 and let us assume that the
target allocated capacity of PN3 is 16 units, and that the requested virtual capacity is equivalent to the reward that a physical node gets if it wins the hosting of that virtual node. In this example, let us also assume that physical node utilities are equivalent to their residual physical capacity, e.g., a physical node with residual capacity 10 units bids 10 to attempt the hosting of a virtual node whose requested capacity is no higher than 10 units. Let us apply MAD to construct the bundle of PN3. First PN3 bids on VN2, as it is the virtual node with the highest requested capacity (reward) and so $b_{PN3} = (0, 20, 0, 0)$. After filling its bundle with VN2, PN3 updates its residual capacity from 20 to 11 (as VN2 requested capacity is 9). The next virtual node to be inserted in the bundle is hence VN1, as it has the highest requested capacity among the remaining virtual nodes. PN3 election phase terminates with $b_{PN3} = (11, 20, 0, 0)$, $w_{PN3} = (PN3, PN3, -, -)$ and bundle $m_{PN3} = (VN2, VN1)$, as embedding more virtual nodes would increase the allocated capacity beyond the target.

4.1.4 Phase 2: CADE Agreement Phase

In this phase, physical nodes make use of a maximum consensus strategy to converge on the winning bids $\bar{b}$, and to compute the allocation vector $\bar{a}$ (Procedure 2).

The consensus, for example on the utility vector $b_i$ after receiving the utilities from each physical node $k$ in $i$’s neighborhood $N_i$, is performed by comparing the utility value $b_{ij}$ with $b_{kj}$ for all $k$ members of $N_i$. This evaluation is performed by the function $\text{IsUpdated}()$ (line 5). In case the algorithm requires consensus only on a single virtual node at a time, i.e. $|m_i| = 1$ as in SAD, the function $\text{IsUpdated}()$ merely checks if there is a higher utility value, that is, if $\exists k, j : b_{kj} > b_{ij}$. This means that when a physical node $i$ receives from a neighboring physical node $k$ a higher utility value for a virtual node $j$, the receiver $i$ always updates its utility vector $b_i$ ($b_{ij} \leftarrow b_{kj}$), no matter when the higher utility value was generated. In general, i.e., when $|m_i| > 1$, physical nodes may receive higher utilities that are out of date. We discuss the conflict resolution of CADE in Section 4.1.5.

Example 3. (SAD consensus). We have assumed that hosting higher capacity virtual nodes bring higher revenue, and so continuing Example 1, after exchanging its utility
vector with PN5, PN3 updates $b_{PN3}$ from $(0,20)$ to $(0,40)$, and $x_{PN3}$ from $(0,1)$ to $(0,0)$. Having lost the election for node VN2 (the most profitable virtual node) to PN5, PN3 bids on VN1, and so updates again its utility vector from $b_{PN3} = (0,40)$ to $(20,40)$, as all PN3’s capacity can now be used for VN1 and PN5’s bid on VN2 is recorded. PN3 also changes its allocation vector again from $x_{PN3} = (0,0)$ to $(1,0)$. Eventually, all physical nodes agree that PN5’s utility is the highest for the most profitable virtual node VN2, while PN4 wins VN1 as it has the highest residual capacity after VN2 assignment.

When instead physical nodes are allowed to elect multiple virtual nodes in the same round ($|m_i| > 1$) as in MAD, even if the received utility value for a virtual node is higher than what is currently known, the information received may not be up-to-date. In other words, the standard max-consensus strategy may not work. Each physical node is required to evaluate the function $\text{IsUpdated}()$. In particular, $\text{IsUpdated}()$ compares the time-stamps of the received bid vector, and updates the bundle, the utility and the assignment vector accordingly (Procedure 2, line 6). Intuitively, a physical node loses its assignment on a virtual node $j$ if it gets outvoted by another physical node that has a more recent bid, or after realizing that its utility for $j$ was subsequent to another previous utility value that it had lost more recently.

More precisely, in CADE utilities on a physical node for the same virtual node are required to be lower if more virtual nodes are previously allocated. This is obvious in our examples, as a physical node uses its residual capacity as utility, that decreases as more virtual nodes are added to the bundle — as we show later, this monotonically non-increasing condition must hold for any other utility function. This means that if a physical node $i$ is outvoted on a virtual node $j$, all the subsequent nodes $m_{ij'}$, $\forall j' > j$ were computed using an invalid value and therefore need to be released, that is, $b_{ij'} = 0$ $\forall j' > j$. 
Procedure 2 CADE agreementPhase for physical node $i$ at iteration $t$

1: Input: $a_i(t)$, $b_i(t)$, $m_i(t)$
2: Output: $a_i(t)$, $b_i(t)$, $m_i(t)$
3: for all $k \in N_i$ do
4: for all $j \in V_H$ do
5: if $\text{IsUpdated}(b_{kj})$ then
6: update($b_i(t), a_i(t), m_i(t)$)
7: end if
8: end for
9: end for

4.1.5 Conflicts resolution

When it receives a bid update, physical node $i$ has three options: (i) ignore the received bid leaving its bid vector and its allocation vector as they are, (ii) update according to the information received, i.e. $w_{ij} = w_{kj}$ and $b_{ij} = b_{kj}$, or (iii) reset, i.e. $w_{ij} = \emptyset$ and $b_{ij} = 0$. When $|m_i| > 1$, the bids alone are not enough to determine the auction winner as virtual nodes can be released, and a physical node $i$ does not know if the bid received has been released or is outdated.

We conclude this subsection with two remarks that explore how such conflicts are resolved. In particular, we illustrate how bids should be ignored or reset if they are outdated, and how subsequent bids to a more recently lost bid should be released.

Remark 1. (utility values may be ignored or reset). There are cases in which the utility values are not enough to resolve conflicts, and so the time-stamps at which the utility value was generated are used to resolve conflicts. In particular, (1) if a sender physical node $i$ thinks that a receiver $k$ is the winner and $k$ thinks the winner is $n \neq \{i, k\}$, or (2) when $i$ thinks $n$ is the winner and $k$ thinks the winner is $m \neq \{n, i, k\}$, or when (3) both $i$ and $k$ think $m$ is winning but with a different bid. In all these cases, knowing which utility value is most recent allows $k$ to either ignore or update its value based on the utility value received from $i$. In other cases, even the time-stamps are not enough and $i$ and $k$ need to reset their utility values. In particular, (4) when $i$ thinks the winner is $k$ and $k$ thinks the winner is $i$. In this case, even if $i$’s utilities were more recently generated, it might have been generated before $k$’s utility value was received by $i$. The complete synchronous conflict resolution table
that we used in our simulations is reported in Appendix A.

**Remark 2.** (releasing subsequent bids). Given PN3’s bidding phase in Example 2, and computing PN5’s vectors we have: \( m_{PN5} = (VN2, VN1, VN3, VN4) \), \( b_{PN5} = (31, 40, 25, 20) \) and \( w_{PN5} = (PN5, PN5, PN5, PN5) \). After receiving the bids from PN5, PN3 realizes that its first bundle’s entry is outbid \( (20 < 40) \) and so it must release VN2. Therefore PN3 needs to also release the other subsequent node in its bundle VN1, as its bid value was a function of the bid on VN2, i.e. the bid on VN1 assumed the residual capacity after VN2 is allocated on PN3.

Since CADE allows physical nodes to generate utility values using their most updated residual capacity, releasing subsequent items from a bundle intuitively improves the sum of the utilities of the physical nodes and hence, when physical nodes cooperate, this improves the number of slices allocated. Moreover, as we show in Section 4.2.1, such residual capacity utility guarantees convergence to a slice embedding. Note also that, due to the slice topology constraints, a change of assignment of any virtual node not present in a bundle may invalidate all its utility values. Assume, for example (Figure 4·2), that PN5 is winning VN2 when PN3 bids on VN1. The utility for hosting VN1 may change if the connected VN2 is later hosted by another physical node, e.g. PN4, as the residual physical link capacity to connect physical nodes PN3 and PN4 may be smaller than the residual capacity of the physical link connecting PN3 and PN5. In extreme cases, the residual capacity of the physical link PN3-PN4 can be even null, not allowing the embedding of the slice at all. To avoid storing utility values computed with an out-of-date utility, physical nodes simply reset their own bundle at the beginning of every election phase (procedure 1, line 3).

### 4.1.6 Pseudo sub-modular utility functions

As we will see in Section 4.2, our CADE mechanism guarantees convergence allowing InPs to use their own embedding policies, as long as the function appears to be sub-modular to other bidders \([JCPH12]\). Sub-modularity is a well studied concept
in mathematics [NWF78], and applied to the distributed slice embedding problem, can be defined as follows:

**Definition 4.** (*Sub-modular function.*) The marginal utility function $U(j, m)$ obtained by adding a virtual resource $j$ to an existing bundle $m$, is sub-modular if and only if

\[ U(j, m') \geq U(j, m) \forall m' | m' \subset m. \tag{4.4} \]

This means that if a physical node uses a sub-modular utility function, a value of a particular virtual resource $j$ cannot increase because of the presence of other resources in the bundle.

Although having sub-modular utility functions may be realistic in many resource allocation problems [KST09], in the distributed slice embedding problem this assumption may be too restrictive, as the value of a virtual node may increase as new resources are added to the bundle, e.g. the cost of mapping a virtual link between two virtual nodes decreases if a physical node hosts both virtual source and destination. To guarantee convergence without using a sub-modular score function, as in [JCPH12], we let each physical node communicate its bid on virtual node $j$ obtained from a bid warping function:

\[ W_{ij}(U_{ij}, b_i) = \min_{k \in \{1, \ldots, |b_i|\}} \{U_{ij}, W_{ik}\} \tag{4.5} \]

where $W_{ik}$ is the value of the warping function for the $k^{th}$ element of $b_i$. Note how by definition, applying the function $W$ to the bid before sending it is equivalent to communicating a bid that is never higher than any previously communicated bids. In other words, bids *appear* to other physical nodes to be obtained from a sub-modular utility function.
4.1.7 Phase 3: Virtual Link Embedding

Similar to the bidding and agreement phases for virtual nodes, in the virtual link embedding phase, our CADE mechanisms allow applications and provider’s goals to tune the slice embedding protocol behavior through policy instantiation.

This last phase is based on the observation that all virtual link embedding schemes have two commonalities: information known at each physical node about physical paths, and the algorithm for determining the best physical path(s) to allocate a virtual link. We hence define three CADE policies for virtual link embedding: (i) the type of information known at each physical node, for example the routing table or the available paths for any source-destination, (ii) the update frequency of such information, for example every hour or every time a new slice is requested, and (iii) the selection of physical path(s) over which a virtual link is mapped. One example of such virtual link embedding scheme is a simple SP assisted auction, where, similarly to [ZXB10] and [IKNS05], an SP elicits bids from each InP, computes the “cheapest” loop-free physical path according to the bids, and then allocates the virtual link on that path. As shown in [YYRC08], another effective example is a k-shortest path algorithm with path splitting [Epp99].

In our experiments we let physical nodes know the routing table, computed only once at the beginning of our experiments using Dijkstra’s algorithm, and we also use the k-shortest (hop distance) path algorithm with $k = 3$. This virtual link (path) embedding policy has the limitation of forcing intermediate physical nodes on a path to accept the allocation of a virtual link if they have capacity. We leave for future work the exploration of other strategies (for example path bidding [IKNS05]).
4.2 Convergence and Performance Guarantees

In this section we show results on the convergence properties of CADE. By convergence we mean that a valid mapping (Section 4.1) is found in a finite number of steps (Definition 3). Moreover, leveraging well-known results on sub-modular functions [NWF78, Fei98], we show that under the assumption of pseudo sub-modularity (Section 4.1.6) of the utility function, CADE guarantees a \((1 - \frac{1}{e})\) optimal approximation. \(^6\)

4.2.1 Convergence Analysis

All physical nodes need to be aware of the mapping, by exchanging their bids with only their first-hop neighbors, therefore a change of a maximum utility needs to traverse all the physical network, which we assume has diameter \(D\). The following proposition (Proposition 1) states that a propagation time of \(D\) hops is also a necessary and sufficient condition to reach max-consensus on a single virtual node allocation. Another interesting observation that follows from the result is that the number of steps for CADE to converge on the embedding of a slice of \(|V_H|\) virtual nodes is always \(D \cdot |V_H|\) in the worst case, regardless of the size of the bundle vector. This means that the same worst-case convergence bound is achieved if CADE runs on a single or on multiple virtual nodes simultaneously. These claims are a corollary of Theorem 1 in [CBH09], which deals with a distributed task allocation problem for a fleet of robots.

Let the tasks allocated by a robot represent the virtual nodes to be hosted by a physical node. Therefore, by induction on the size of the bundle the following result holds as a corollary of Theorem 1 in Choi et al. [CBH09]:

---

\(^6\)Note that in our descriptive examples we have used utility functions that optimize the allocation of virtual nodes and their first-hop links, but not virtual path allocations (discusses in Section 4.3.2).
Proposition 1. (Convergence of CADE). Given a virtual network \( H \) with \( |V_H| \) virtual nodes to be embedded on a physical network with diameter \( D \), the utility function of each physical node is pseudo sub-modular, and the communications occur over reliable channels, then the CADE mechanism converges in a number of iterations bounded above by \( D \cdot |V_H| \).

**Proof (Sketch).** We use \( W_{ij}(U_{ij}, b_i) \) as a utility function (sub-modular by definition). From [CBH09] we know that a consensus-based auction run by a fleet of \( N_u \) agents, each assigned at most \( L_t \) tasks, so as to allocate \( N_t \) tasks, converges in at most \( N_{\min} \cdot D \) where \( N_{\min} = \min\{N_t, N_u \cdot L_t\} \). Note that the proof of Theorem 1 in [CBH09] is independent of the utility function used by the agents as long as they are sub-modular, and of the constraints that need to be enforced on the tasks. Since for CADE to converge, every virtual node needs to be assigned, in the distributed slice embedding problem, \( N_{\min} \) is always equal to \( N_t \equiv |V_H| \), and therefore we prove the claim.

Proposition 2. (Message complexity) The number of messages exchanged to reach an agreement on the node assignment using the CADE mechanisms is at most \( D \cdot |E_G| \), where \( D \) is the diameter of the physical network and \( |E_G| \) is the number of directed edges in the physical network.

4.2.2 Performance Guarantees

We assume that each physical node \( i \) does not bid on a virtual node \( j \) unless it brings a positive utility, therefore \( U_{ij} \) and so \( W_{ij} \) are positive. Moreover, if we append the bundle \( m_i \) to bid on an additional set of virtual nodes \( v \) resulting in bid vector \( b'_i \), we have:

\[
W_{ij}(U_{ij}, b'_i) \leq W_{ij}(U_{ij}, m_i) \forall v \neq \emptyset \tag{4.6}
\]

which means that \( W_{ij} \) is monotonically non-increasing.

Since the sum of the utilities of each single physical node, and since the bid warping function \( W_{ij}(U_{ij}, b_i) \) of CADE is a positive, monotone (non-increasing) and
sub-modular function, all the axioms of Theorem 3.1 in Nemhauser et al. [NWF78] on sub-modular functions are satisfied. We hence obtain the following result:

**Proposition 3.** (*CADE Approximation*). The *CADE node consensus strategy yields an* \((1 - \frac{1}{e})\)-**approximation** w.r.t. the optimal node assignment solution.

### 4.3 Performance Evaluation

To test the proposed distributed embedding algorithms, we developed our own trace-driven simulator, whose code is publicly available at [CAD].

**Physical Network Model**: Using the BRITE topology generator [MLMB01], we obtain a physical topology. We use the generation model of BRITE to build a flat topology using either the Waxman model, or the Barabasi-Albert model with incremental growth and preferential connectivity. We tested our algorithms with physical network sizes varying \(n\) physical nodes with about \(5n\) physical links (as in [YYRC08]).

Our simulations do not consider delay constraints, while link capacity constraints are discussed later in this section. The results are similar regardless of the topology generation model and the physical network size. We only show the results obtained for
Figure 4.4: (left) SAD allocates more slices when a single shortest path is available. (right) MAD allocates more slices when a $k$-shortest path link allocation policy (where $k = 3$) is used.

Figure 4.5: (left) MAD has shorter convergence time. (right) SAD has shorter response time.
Figure 4.6: (left) MAD allocates more slices consecutively ($k = 3$). (right) Considering simultaneously node and link stress in the utility improves the slice allocation ratio.

$n = 50$ and a Barabasi-Albert physical topology.

**Virtual Network Model:** we use a real dataset of 8 years of Emulab [WLS$^+$] slice requests [R.R11]. For each simulation run we process 61968 requests; the average size of a request is 14 with standard deviation of 36 virtual nodes; 99% of the requests have less than 100 virtual nodes, and 85% have at most 20 virtual nodes. Excluding the 10% long-lived requests that cause the standard deviation of slice lifetime to exceed 4-million seconds, the duration of the requests is on average 561 with 414 seconds of standard deviation (Figure 4.3 left). As the dataset does not contain neither the number of virtual links nor the virtual network topology, we connect each pair of virtual nodes at random with different average node degree (Figures 4.4, 4.5, 4.6). Moreover, we extend our evaluation comparing linear, star, tree, and fully connected virtual topologies (Figure 4.3 right). All our simulation results show 95% confidence intervals; the randomness comes from both the virtual network topology to be embedded, and the virtual constraints, that is, virtual node and link capacity requirements. Similarly to previous work [YYRC08], we randomly assign physical
link capacities between 1 and 100, then we assign the physical node capacity to be the sum of its outgoing physical link capacities. Then we assume the virtual link capacity to be randomly chosen between $1/R$ and $100/R$, where $R = \{50, 100, 500\}$, and the virtual node capacity is then assigned to be the sum of its outgoing virtual links. The results are similar and we only show plots for $R = 100$.

**Comparison Method:** we compare our CADE mechanism, instantiated with the SAD and MAD configuration, with another policy based distributed virtual network embedding algorithm, PolyViNE [CSB10], and with the first published distributed virtual network embedding algorithm [HLZ08], that we call Hub and Spoke due to the adopted heuristic.

**Evaluation metrics:** our evaluation results quantify the benefits of our approach along two metrics: embedding efficiency and time to find a solution. In particular, we evaluate the response time — number of steps measured in one-hop communications needed to realize a VN can or cannot be embedded — and the convergence time — number of steps until a *valid* embedding is found. The efficiency of an embedding is evaluated with the VN allocation ratio — ratio between the number of virtual networks successfully embedded and requested, and with the resource utilization — physical node and link capacity utilized to embed the VN requests, as well as with the *endurance* of the algorithm, *i.e.* the number of successfully allocated requests before the first VN request is rejected. We also evaluate the effect of different utility functions.

### 4.3.1 Simulation results

We present here our trace-driven simulation results summarizing the key observations.

1. **MAD leads to larger VN allocation ratio, as long as multiple physical paths are available for each virtual link.** When the virtual link allocation policy allows a virtual link to be allocated only on a single physical shortest path, SAD has a higher VN
allocation ratio (Figure 4.4 left). This is because SAD, allowing a single virtual node allocation for each node allocation round, balances the load over physical resources more efficiently. When instead a physical node $i$ is allowed to simultaneously win a bundle of virtual nodes $m_i$ as in MAD, the physical links adjacent to $i$ quickly exhaust their capacity due to the VN topology; all the outgoing virtual links adjacent to the virtual nodes in $m_i$ that are not mapped on $i$ are in fact mapped onto a small set of physical paths starting from physical node $i$. However, if the virtual link embedding policy uses a $k$-shortest path (with $k \geq 3$), MAD is able to allocate more VNs (Figure 4.4 right). From this result we conclude that when fewer physical paths are available, InPs should consider (switching to) a SAD setting, otherwise MAD is more efficient. In the considered physical topologies, there are no more than 3 physical paths between any pair of physical nodes, and the confidence intervals overlap for SAD and MAD with $k = 2$.

(2) MAD has faster convergence time. Although we showed that MAD has the same worst-case convergence bound as SAD, simulation results show how MAD can in practice be faster (Figure 4.5 left). In the best case, a single physical node has highest bids for all virtual nodes, and all the other bidders will converge on a VN allocation in a single auction round.

(3) SAD has faster response time. Due to the VN partitioning policy, that is, due to the fact that the SP releases only two virtual nodes at a time, SAD has a quicker response time as physical nodes immediately know if a virtual node or a link (and so the entire VN) cannot be allocated (Figure 4.5 right). We do not show the response time for the other algorithms in Figure 4.5 (right) as they are similar to their convergence time.

(4) SAD better balances the load independent of the VN topology. To verify our findings, we average over time the variance of the utilization across all nodes with 25% and 75% percentiles for each of the algorithms, and we repeat the experiment
for linear, star, tree, and full virtual network topologies (Figure 4·3 right). Note how SAD better balances the load, independent of the VN topology. One exception is PolyViNE, that has lowest load variance for tree topologies, but at the expense of lowest VN allocation ratio.

(5) **SAD allocates more VNs before the first one is rejected.** As a direct consequence of a better VN allocation ratio, we verify that SAD yields a larger number of VNs allocated before the first one gets rejected in case the virtual link allocation policy allows only a single physical shortest path, while MAD allocates more requests if multiple physical loop-free paths are available (Figure 4·6 left).

(6) **Considering link stress in the utility function improves the VN allocation ratio.** In this last experiment we show how different utility functions may lead to different VN allocation efficiency. In particular, by comparing two different utilities, *i.e.* $U'_{ij} = (T_i - S'_{ij})$ where $S'$ is only the stress on the physical nodes, and $U_{ij}$ where the stress also includes adjacent physical links, we confirm the premise that considering nodes and links simultaneously in the slice embedding problem leads to higher VN allocation rate (Figure 4·6 right). We leave the investigation of the best utility function given the goals of providers as an interesting research direction.

### 4.3.2 Path Allocations

In this subsection we analyze the performance of a Path Allocation for Distributed slice embedding (PAD), another CADE policy in which physical nodes attempt to host contiguous virtual paths. By contiguous virtual path we mean that neighboring virtual nodes are allocated to neighboring physical nodes. In other words, each virtual link is allocated on a single physical link.

Both SAD and MAD node allocation policies may result in assignments in which virtual links might be established between non neighboring physical nodes, forcing intermediate physical nodes to relay data traffic. During the bidding phase, physical
nodes applying the PAD embedding policy are allowed to attempt hosting a virtual node $j$ only if the virtual nodes adjacent to $j$ are currently won by the node itself, or by an adjacent physical node. By forcing the PAD policy, virtual path of length $L > 0$ will be embedded on loop-free physical path of length at most $L$, hence avoiding physical node relays.

The PAD policy performs better than the MAD policy but worse than the SAD policy. This advantage over MAD vanishes as the number of links grows, i.e. we move from a linear to a full virtual network topology (Figures 4·7). We can explain this effect by further delving into the partial embedding of virtual nodes and link allocation (Figures 4·8 and 4·9). PAD limits the space of possible physical nodes that can participate in an embedding; during the CADE election phase of a full virtual network topology, there are less physical nodes available and less adjacent link capacity to host the slice, hence the slice allocation ratio decreases. Note also how PAD falls under the category of the multiple-item embedding policies, and so it has the same convergence time of MAD.
Figure 4.7: Slice allocation ratio: physical network of 100 nodes following Barabasi-Albert connectivity model.
Figure 4.8: Link Allocation Ratio: physical network of 100 nodes following Barabasi-Albert connectivity model.
Figure 4.9: Node allocation ratio: physical network of 100 nodes following Barabasi-Albert connectivity model.
Chapter 5

Embedding Protocol and Object Model Design

Transparency is the ability of hiding the complexity of the implementation of mechanisms of a (distributed) system from both users (of a service or applications) and application programmers. To provide transparency, a distributed system architecture should offer interfaces to the (physical, virtual or logical) resources, so that such resources appear to be locally available. An object model is the means by which such transparency is provided, and consists of (i) a set of object definitions, (ii) a set of interfaces to the object attributes (resources), (iii) a set of operations on the objects, and (iv) a broker to handle such operations.

The design and implementation of our virtual network embedding object model arose from years of design and implementation experience on network management and architectures [Day08, DMM08, EWMD, EWMD12, IAEM12, EWMD13, WEMD13b, WEMD13a].

In this section we define, a VIrtual Network Object model (VINO), in support of our policy-based architecture. The objects are updated by the physical nodes participating in the virtual network embedding process, and stored into a distributed
data structure called *Slice Information Base (SIB).* ¹ As the Management Information Base (MIB) defined in [K. 90] or the Network Information Base (NIB) defined in Onix [KCG⁺10], our SIB is a partially replicated distributed object-oriented database that contains the union of all managed objects within a slice (virtual network) to embed, together with their attributes. In the NIB, attributes are element of the forwarding table; in our SIB, an example of attribute is a list of virtual neighbors for a given virtual network, or the list of physical nodes currently mapping a given virtual link. The role and responsibilities of the SIB daemon are similar to those of memory management in an operating system: to manage the information stored in the SIB and its veracity, updating and making states available to physical nodes participating in the virtual network embedding.

Based on a publish/subscribe model, a distributed set of SIBs and SIB daemons enable infrastructure and service providers to specify different styles of embedding management architectures, ranging from fully decentralized, *i.e.* autonomic, to centralized, *i.e.* manager-agents style, to hybrid approaches, *e.g.* hierarchical.

In the rest of this section we describe the broker architecture (Section 5.1), the objects (Section 5.4), as well as the interface (Section 5.2) and the operations on such objects (Section 5.3), that is, the CADE protocol used to modify the object attributes *i.e.* the physical network states during a virtual network embedding.

### 5.1 Broker (SIB Daemon)

Similar to traditional existing network service management object models [Int92, Obj92], our architecture has a broker (part of the SIB management) responsible for allowing physical nodes participating to a virtual network embedding to transparently make requests and receive responses. The broker handles such communication

---

¹Slice is an alternative term for virtual network. The physical network is in fact “sliced” into multiple virtual networks.
Figure 5.1: Object Model interfaces: the SIB daemon is responsible for managing the information stored in the SIB and its veracity, updating and making states available to service and infrastructure provider processes.

with subscription events. A subscription represents the quantity and type of information, to propagate in predefined situations by an embedding instance on objects when specific situations occur. In our architecture, example of publishers are service providers or infrastructure providers, and example of object attributes being published are the constraints of a virtual network to be embedded, or routing updates among physical node participating in a distributed virtual network embedding.

Subscription events are mapped by the broker that recognizes the objects from their type, and acts upon different requests with a set of operations on the objects stored in a local or a remote SIB, on behalf of an embedding application instance. Our subscriptions have equivalent design goals as the notification events defined by the OSI model [ISO91], or traps in the Simple Network Management Protocol (SNMP) [Cas90], though they specifically support virtual network embedding operations.

5.2 SIB Daemon API

We provide an API that simplifies the design and development of sophisticated virtual network embedding solutions, leveraging the separation between mechanisms and
policies, thus allowing physical nodes to read and write any state — set of object attributes — of any physical node participating in a virtual network embedding. Object attributes can be read or written through a general subscription mechanism that includes registration for passive (subscribe) or active (publish) notifications of local or remote state changes.

Every physical node within a service or infrastructure provider has a copy of its embedding management states stored in the SIB distributed data structure. Every physical node involved in the virtual network embedding runs a Broker (or SIB daemon), responsible for managing the subscriptions and updating the SIB distributed object database. The SIB represents a generalized case of the Routing Information Base stored in IP routers. Rather than only storing prefixes to destinations, our SIB stores all the states accessible by each component of our virtual network embedding architecture.

**Subscription structure:** when a physical node subscribes to a service, for example a virtual network request from the service provider, a subscription event is created. A subscription requires a set of parameters: *(i)* a mandatory unique subscription ID, limited to the scope of the physical network; *(ii)* the object attribute to be exchanged, *(iii)* a list of physical nodes and/or service providers among which the attributes are exchanged, *(iv)* a frequency at which those attribute have to be exchanged and *(v)* a bit to identify on whether the event needs to read (0) or write (1) *i.e.* subscribe or publish the attributes.

**Publish/Subscribe:** Our SIB subscription mechanism is a generalized case of a publish-subscribe paradigm. Standard publish-subscribe systems are usually asymmetric: a given event will be delivered to potentially many subscribers, *i.e.* the publish-subscribe paradigm is a one-to-many communications paradigm. Our SIB
subscription mechanism supports both the symmetric and asymmetric paradigms, and a query-based mechanism. A symmetric subscription mechanism is a process in which the publisher node is capable of selecting the subscriber nodes. For example, a virtual network embedding application process sending capacity updates may prefer to be (temporarily) silent with subscribers along a given path, because of congestion, or because it has detected a misconfiguration or a suspicious behavior. Moreover, our mechanism supports the traditional query-based paradigm, where a physical node may send a message to another physical node and waits for its response.

5.3 CADE Protocol Messages
(Operations on Objects)

To share or modify states such as routing updates, or virtual to physical mappings, we define a set of operations executable on (remote) objects. Such operations are the CADE protocol messages. The CADE messages are supported by encapsulation in the payload of a Common Distributed Application Protocol (CDAP) message [DMM08, EWMD13]. CDAP is our object-based protocol, whose design is based on modular, object-oriented principles as in HEMS [PT87], CMIP [ISO91] or ACSE [ISO95]. The CDAP protocol is based on three logical modules: a Common Application Connection Establishment (CACE) module, required as in [ISO95] for application protocol and syntax agreement within the application connection, an (optional) authentication module, and a set of CDAP messages for operating on objects as in CMIP [ISO91] or SNMP [Cas90].

Why CDAP? Object-based protocols like HEMS, CDAP, CMIP or SNMP were designed to support any management application protocol; CADE is a virtual network
management protocol. To disseminate the information necessary for the coordina-
tion of any virtual network management operation, we used CDAP as it has less
overhead than the more complex CMIP or HEMS [PT87], and it is more expressive
than SNMP.

The architecture frees (virtual) network programmers to leverage the existing
virtual network embedding architecture and define new types of objects to serve any
virtual network management purpose. However, we impose that all the physical node
instances agree on the same object data representation for the embedding protocol,
and we impose restrictions on specific uses of such protocol messages, in forming and
managing a centralized or distributed virtual network embedding. In the rest of this
section we describe such restrictions.

CADE is an application protocol, and hence it relies on the reliability of the trans-
port service provided by the layer below; the messages are sent and received asyn-
chronously between the physical nodes and a service provider during the embedding
process. We now describe the protocol messages, and the actions to perform upon
their reception.

**Slice Request:** this primitive is invoked when a service provider wants to issue an
virtual network embedding request. This primitive is sent to at least a physical node
belonging to some infrastructure provider and an *embedding timeout* is started. The
message contains a slice identifier, and the temporal and non temporal constraints
*e.g.* virtual node capacity, location, time at which the slice is needed and the lifetime
(virtual network entry and exit time).

Upon receipt of the Slice Request message, if the receiver physical node \( k \) is ca-
ppable of hosting at least a virtual node, a **First Vote** message is sent to all the
logical neighbors, *i.e.* all physical nodes that have subscribed to \( k \), *e.g.*, all its first
hop neighbor physical nodes. If the receiver physical node \( k \) can allocate the entire
virtual network, a **success** message is sent to the service provider directly. When instead the receiver physical node \( k \) cannot allocate any virtual node, a **First Bid** message message is sent to all the logical neighbors to forward the request.

**First Vote:** this primitive is invoked by a physical node that has received an embedding request from a service provider. The message contains the utility value (vote) of the sender physical node (it might be empty), as well as the virtual network constraints, received by the slice provider.\(^2\)

Upon receipt of this message, a physical node applies the CADE algorithm, attempting to overcome the utility of the requested virtual nodes, and when necessary, the physical node propagates its utility values. After sending the first vote message, a **vote timeout** is set. This is necessary to understand when the asynchronous agreement phase terminates. When the virtual nodes have all been assigned, this receiver physical node replies with an **Embed Success** message to the service provider. If the **vote timeout** expires before reaching an agreement, the physical node replies the service provider with an **Embed Failure** message.\(^3\)

**Vote:** this primitive is invoked when a physical node has terminated a virtual network mapping agreement phase, and as a result the utility values data structures need to be propagated for a distributed virtual network embedding. The payload of the message contains the utility value objects of the sender physical node customized according to the CADE policies.

---

\(^2\)The mandatory sliceID field can be cryptographically signed (encrypted using public-key cryptography) to avoid malicious physical nodes changing the slice request.

\(^3\)Note how an embedding agreement may be reached before the vote timeout expires. To speed up the embedding process we could include a system of tokens [BT89]. Each physical node would have to update an additional bitfield data structure, of size equal to the number of physical nodes participate to the embedding, and propagate with its neighbors the bitwise OR of the received bitfields. When the zeros disappear from the bitfield, the physical node may respond with a positive or negative response.
Upon receipt of this message, if the receiver physical node \( k \) is able to overbid, the utility data structures are updated and another vote message is propagated; the propagation occurs if at least a physical node have subscribed to \( k \). Physical nodes that subscribe for votes after a slice request has been issued may not participate to the ongoing embeddings.

**Embed Success:** this primitive can be invoked only by the physical nodes that have received the **Slice Request** from a service provider. The message is created and sent to the slice provider after the **vote timeout** has expired, and an embedding agreement has been reached. Upon receipt of this message, the slice provider releases the next slice partition, if any, or else starts the link embedding phase invoking the **Link Embedding** primitive.

**Embed Failure:** this primitive is invoked by the physical node that received the **Slice Request** message, after its **vote timeout** has expired, and there is still no agreement on the requested virtual network. Upon Receipt of this message, the service provider logs the embedding failure, and either releases the next slice, if any, else returns in a listening state for new embedding requests.

**Link Embedding:** this primitive is invoked by the service provider after receiving an **Embed Success** message from a physical node. Upon Receipt of this message, the slice provider sends to the winner physical nodes of each virtual node the identity of the other end, in order for them to establish a virtual link.

**Connect request:** this primitive is invoked by a physical node after receiving a Link Embedding message from the service provider. Upon receipt, a physical node requests a flow by sending a **Connect Response** message to the other end of the
virtual link to embed, and updates his states with the final biding of physical to virtual resource. The requested virtual link bandwidth is reserved using a traffic shaper system call.

**Connect Response:** this primitive is invoked by a physical node after receiving a Connect request from a service provider or from a physical node. Upon receipt of this message the bandwidth is reserved using the virtual network allocation service interface.

**Slice Release Request:** this primitive is invoked by a service provider to terminate the virtual network. Upon receipt, the receiver physical node sends back a Slice Release Response message after releasing the reserved resources, and terminating all the applications running on each hosted virtual node.

**Slice Release Response:** this primitive is invoked by a physical node in response to a Slice Release Request message. Upon receipt of this message, the receiver node releases its reserved virtual resources after terminating all the applications running on each hosted virtual node.

### 5.4 Virtual Network Embedding Objects

We define each object using the recently released Google Protocol Buffer (GPB) abstract syntax [Goo13]. One of the main advantages of using an abstract syntax is the implementation independence of the framework —the object model can be serialized and deserialized using any programming language. Many object serialization languages have been proposed. We can classify them into binary serialization e.g. Binary JavaScript Object Notation (BSON) [Int13] and Google Protocol Buffer (GPB) [Goo13], and character-based, e.g. the Extensible Markup Language
(XML) [Wor13] or the Abstract Syntax Notation 1 (ASN.1) [Lar99]. Character based representations as XML are more powerful than what we need, which leads both to unnecessary complexity and size in implementation. Binary serializations like BSON and GPB are order-of-magnitude more efficient to parse than XML for example, depending on how rich is the XML parser. BSON was a good candidate, but we choose GPB as we were already familiar with the compiler to serialize and deserialized the data structures (objects). In the rest of this section we report the format object definitions.

Format of a CADE Object

We called the main object (message) of the CADE protocol **CADE**. This object is used to exchange policies and embedding requests among physical nodes belonging to both service and infrastructure providers. A CADE object has the following attributes:

```plaintext
message CADE {
    required int32 version
    required int32 sliceID
    optional Slice sliceRequest
    optional string allocationPolicy
    repeated assignment a
    repeated utility b
    repeated int32 m
    repeated voteForgingTime timeStamp
}
```

The required attribute **version** specifies the version of the CADE protocol (only one version exists today). The only other required attribute is the **sliceID**. The attribute is needed to support simultaneous virtual network embeddings.

The attribute **sliceID** is an identifier that must remain unique for the entire lifetime of the slice (virtual network), within the scope of both service and infrastructure providers. It is a 32 bit identifier, and it could be any integer, including an
hash value of the string `sliceproviderName.sliceID`. The field `allocationPolicy` allows service providers to specify different virtual network embedding strategies. This attribute is used to specify the form of the `assignment` vector `a`.

**Format of the Assignment Object**

We have an `assignment` object for each virtual node, is defined as follows:

```java
message assignment {
  required int32 vNodeId
  optional string hostingPnodeName
  optional bool assigned
}
```

The assignment object is used to keep track of the current virtual to physical node mappings. The `allocationPolicy` may assume two forms: `least` and `most` informative. If the `allocationPolicy` attribute is set to “least”, the `assignment` attribute `a` is filled out with its boolean `assigned` field — set to true if physical node `i` hosts virtual node `j` and 0 otherwise. When the `allocationPolicy` attribute is set to its most informative form, then the attribute `a` should contain the identities of the physical node currently hosting so far the virtual node identifiers *i.e.*, with integers representing the `vNodeID` attributes.

Note that if the `allocationPolicy` is set to its most informative form, the assignment vector reveals information on which physical nodes are so far hosting `vNodeID`, whereas if the `allocationPolicy` is set to its least informative form, each physical node only knows if `vNodeID` is currently being hosted by a physical node or not.\(^4\)

The remaining attributes of the CADE object (bid vector, bundle vector and the time stamp vector) are needed to resolve the conflicts during the agreement phase of the CADE protocol.

\(^4\)Note how, as a consequence of the max-consensus, when using the least informative assignment policy, each physical node only knows the value of the maximum utility so far without knowing the identity of the bidder.
Format of the Bid Objects

The following two attributes of a CADE object, vote and voteForgingTime are essential to run our distributed consensus embedding asynchronously. In particular, their abstract syntax is specified as follows:

```plaintext
message utility {
  required int32 vNodeId
  optional double utilityValue
}
message voteForgingTime {
  required int32 vNodeId
  optional int64 time
}
```

The voteForgingTime object is essential to solve conflicts in CADE (agreement phase), as the time at which the message is sent or received are not sufficient to guarantee convergence to an embedding agreement. This is because CADE is an asynchronous communication protocol, and messages from different sources may arrive out of order, i.e., messages created earlier than another message could potentially arrive at a later time. 5 Every time a physical node bids on a virtual node identified by the vNodeId attribute, a utilityValue and the time attributes are forged.

Format of the Slice Specification Object

A Slice is attached as an attribute to a CADE object and sent from a service provider in a Slice Request message to at least a physical node to begin the embedding process. The object is also attached in a First Vote message. The Slice object abstract syntax is defined as follows:

---

5Note that CADE is an application protocol, and so it does not perform transport functionalities; this means that message reordering from the same source are not a problem for CADE as they are handled by the reliable transport protocol on which CADE relies on.
message Slice {
  required int32 sliceID
  optional int64 entryTime
  optional int64 exitTime
  optional string topology
  optional string predicate
  repeated vNode virtualnode
  repeated vLink virtuallink
}

The first required attribute is the sliceID, a unique identifier within the scope of the service provider. The two optional attributes entryTime and exitTime define the lifetime of the virtual network. The topology and the predicate attributes enable filtering rules. For example, a service provider may send all virtual network requests whose predicate attribute is set to Partition1 to a given subset (partition) of the physical network, e.g. to proactively balance the load, or to increase the response time of an embedding. Service providers could also use the predicate attribute to manage the virtual network partitions.

Virtual Node and Virtual Link Objects

The fields vNode and vLink define the constraints for each virtual node and link, respectively, and their abstract syntax is defined as follows:

message vNode {
  required int32 vNodeId
  optional int32 vNodeCapacity
  optional int32 vNodeType
  optional string vNodeClass
  optional string vNodeName
}

The attribute vNodeId is the unique virtual node identifier while vNodeCapacity represents the requested capacity. The vNodeType attribute enables additional ex-
pressiveness in the slice constraint specifications. For example, small, large or extra-large virtual node type, as in Amazon EC instance [Amab]. The `vNodeName` and the `vNodeClass` attributes allow the specification of a hierarchical object model for virtual nodes. For example, the `vNodeName` may be used to specify the name (address) or the region (e.g. the country or the subnetwork) of the physical node on which the virtual nodes must be embedded, while the `vNodeClass` attribute might be set to `geolocation` to indicate that this virtual node has a geolocation constraint, specified by the `vNodeName` attribute.

The virtual link object is analogous, except that it also requires the identifier of the source and destination virtual nodes. The abstract syntax notation is denoted as follows:

```plaintext
message vLink {
  required int32 vLinkId
  required int32 vSrcID
  required int32 vDstID
  optional int32 vLinkCapacity
  optional int32 vLinkType
  optional string vLinkClass
  optional string vLinkName
}
```

**Format of the Error Code Object**

The `CADErrorCode` object is needed to specify the particular type of errors that may be encountered. The CADE error code design was inspired by the HEMS protoErrorCode [PT87]. The abstract syntax defines two fields: a required error code integer, and an optional message description.

```plaintext
message CADErrorCode {
  required int32 eCode
  optional string description
}
```
The **description** field gives a more detailed description of the particular error encountered, while the error code integer identifies the error type as follows:

0 - **Reserved**. This error code is not used.

1 - **Syntax format error**: some error has been encountered when parsing the received message. Examples of such an error are an unknown type for an object attribute, for example the use of a different type when for the sliceID attribute, or a violation of the Google Buffer Protocol syntax.

2 - **Wrong version number**: this error should be invoked when the version number of the Google Protocol Buffer abstract syntax or the CADE protocol syntax in the common header is invalid. The error may indicate a possible network intrusion, and should be logged at sites concerned with security.

3 - **Authentication error**: this error appears when a message is received by an unknown node or when a node authentication in the physical network has failed. Note that returning an authentication failure information may inform malicious users attempting to crack the authentication system, but it may be useful to detect misconfigurations.

4 - **CADE node application failed**: this error should be sent when any CADE application node failure (service provider or physical node) made impossible the processing of the received message.
Chapter 6

VINEA Prototype Implementation

To establish the practicality of our virtual network embedding architecture and object model, we tested them on a system implementation. The implementation allowed us to refine the design of our object model, and enables users to write real applications on top of the embedded virtual networks.

VINEA processes join a private overlay before running the CADE protocol to embed the request released by a VINEA node instantiated as service provider. Then InP processes run a physical resource discovery protocol, the asynchronous virtual network mapping phase, and finally, the virtual network is allocated using the Mininet library [LHM10]. Our prototype is implemented in a single host Linux-based testbed (Section 6.4), and its InP overlay resources are simulated, i.e., physical CPU and link available capacity are not measured but set from a configuration file, and updated as virtual networks are being embedded. Also, the InP overlay connectivity is emulated by TCP connections on the Linux loopback interface. We emulate the allocation phase of the embedding problem by reserving CPU on virtual hosts, attached to virtual switches running in kernel mode, and we use the Linux Traffic Control application to reserve link capacity. Once the virtual network allocation phase is complete, we run real applications such as ping, iperf and Openflow [MAB+08].
Our VINEA prototype resulted in about $35K$ lines of Java code, without considering comments, test classes, and the Mininet [LHM10] Python and the C code that VINEA leverages for the virtual link allocation. Logically, the prototype is divided into nine main architecture components (Figure 6-1): a Network Management system (NMS), the three embedding services of an infrastructure provider — resource discovery, virtual network mapping and allocation, a set of service provider functionalities, a Slice Information Base (SIB), a broker (or SIB daemon), a message parser to serialize and deserialize objects, and a publish/subscribe system. In the rest of this section we describe in detail each of these components and their functionalities.

6.1 Common VINEA Node Capabilities

Each VINEA node can be instantiated as a service provider node, or as infrastructure provider node. Each infrastructure provider may act as a service provider, and lease
the acquired virtual resources using (recursively) the same mechanisms. Regardless
of the type of VINEA node instance (SP or InP), a set of common mechanisms are
needed in support of both functionalities. In particular, each VINEA node needs to
manage the consistency and updates of both the shared InP overlay, and the virtual
networks to be embedded. States of the InP overlay include connectivity, bids for the
distributed consensus-based auction on the virtual resources, and enrollment states
such as authentication information (ID and password) of new InP processes that
wish to join the private InP overlay to participate in an embedding. States of a
virtual network include (service level objective) constraints such as requested CPU,
bandwidth, delay, or lifetime of the virtual network.

**Network Management System**

**Network Monitoring:** in the network management literature, a Network Manage-
ment System (NMS) is an architecture component usually responsible for monitor-
ing, control, and repair functionalities of a physical network. The NMS component
of our architecture includes an InP overlay monitoring task, as in an NMS of a typ-
ical telecommunication network, and an identity manager, similar to the *Keystone*
component of the OpenStack architecture [Ope13]. The network monitoring task is
a thread that sends at a configurable rate keep-alive messages to all InP processes
of the monitored network. When an InP process does not respond to a keep-alive
message, the NMS publishes an event to update the members of the InP overlay
about the failure status of such node.

**Identity Manager:** when bootstrapping the InP overlay, or when a new VINEA
node wishes to join an existing InP overlay, the identity manager is responsible for
authenticating such processes, so that each process can be trusted. Our current
VINEA implementation [Esp13] supports two authentication policies: “no authen-
tication” —every InP process requesting to join an existing InP overlay is automatically accepted— and authentication with ID and password. In the latter case, the authentication information are to be specified as a clear text in the private InP process configuration file. We separated the identity manager mechanism from its policies, so that other authentication policies may be easily supported, e.g., a public key encryption scheme such as RSA [RSA78].

**InP Overlay Connectivity and DNS:** the Domain Name System (DNS) component is not part of the VINEA architecture (and is not shown in Figure 6-1), but it is a necessary artifact of our InP overlay connectivity implementation.

The connectivity of a real physical network needs to be set up in advance by plugging (ethernet) cables on well-known network interfaces. In VINEA, each wire providing physical connectivity between its nodes is emulated by a TCP connection on dynamically-assigned ports. By dynamically-assigned we mean that each new VINEA node that joins the InP overlay can choose a port and register with DNS. Each VINEA node, once forked, registers with a (previously forked and listening) DNS server, so that later, a wire (i.e. a TCP connection) can be setup with any other VINEA node. Our DNS implementation is centralized.

**Slice Information Base (SIB)**

As described in Chapter 5, the SIB architecture component is responsible for maintaining the object attributes and managing their veracity. Each VINEA node runs an instance of a SIB daemon, responsible for updating such states within the InP overlay, and for creating new states through our pub/sub mechanism.

We support multiple threads accessing the database efficiently, with a synchronized hash table, and we exposed the SIB interface to enable different implementations. An alternative SIB implementation could use an open source object database
as db4o [Dat13].

**Publish/Subscribe Service**

To share and modify the object attributes, each VINEA node has an interface to a publish/subscribe mechanism. SPs for example, publish virtual network objects to embed, together with their policies and constraints, and the interested InP processes subscribe to such objects to attempt a virtual network embedding. The publish/subscribe system is also used by VINEA nodes to publish and subscribe to management objects of the InP overlay, *e.g.* neighbor discovery or routing update events. Each pub/sub event can be customized with an update frequency; for example, VINEA nodes subject to lossy channels may request higher frequency neighbor updates than others.

### 6.2 Service Provider Capabilities

A VINEA node instantiated as an SP has two main functionalities: (*i*) generating virtual network requests, using our *slice specification objects*, and (*ii*) partitioning the virtual network request, when required by the virtual network embedding policy.

The virtual network generator translates incoming virtual network requests into slice objects, that are later serialized and sent to the InPs. The virtual network partitioning problem is NP-hard [HLBAZ11]. VINEA supports a simple virtual network partitioning heuristic, that merely extracts sequentially the next yet-to-be-embedded virtual link from the virtual network request. The partition being sent is hence formed by a single virtual link, and its two adjacent virtual nodes. Each service provider has an interface to the virtual network partitioning service, enabling support for additional (more complex) virtual network partitioning implementations, for example a “hub-and-spoke” heuristic as proposed in [HLZ08].
6.3 Infrastructure Provider Capabilities

The support for the infrastructure provider (or InP process) is the core of the VINEA prototype and by far the most complex, both in terms of logic and size of code. Each InP process has interfaces to the three main mechanisms: resource discovery, virtual network mapping and allocation.

6.3.1 Resource Directory Service

The resource discovery service is the logical set of mechanisms needed to collect and propagate physical and virtual network states such as neighbor discovery, or physical resource availability. The neighbor discovery is useful for the InP overlay monitoring operation performed by the network management system (Section 6.1), while the knowledge of the available physical resource is used by the virtual network mapping and allocation services to make informed embedding decisions. The resource discovery service can be divided into two architecture components: (i) registration and bootstrap, and (ii) discovery.

DNS Registration and Bootstrap: each VINEA node (uniquely identified by an application name or URL) in its bootstrap phase is required to register its address with our DNS. In order to send embedding messages, each VINEA node only needs to know the address of the DNS, and the names of other VINEA nodes physically connected to it.

Inter-Slice Discovery: after the DNS registration, necessary for InP overlay connectivity, InP processes register also with an Inter-Slice Discovery service (ISD) in order to establish a private InP overlay [TGDB12]. The ISD component of the architecture can be thought of a DNS across all private InP overlays potentially hosting
a virtual network. An InP process may wish to participate in the embedding of a particular virtual network, being unaware of whether there are other InP processes currently bidding on it.

When a physical VINEA node belonging to some InP subscribes to an SP to participate in a distributed embedding, it queries the ISD service to obtain the (IP) address of the network management system in charge of the authentication (Section 6.1). If the authentication is successful, the network manager enrolls the new VINEA node enforcing the policies instantiated on that particular InP overlay. Examples of such policies include node and link embedding policies or a given subset of all InP processes currently in the InP overlay, so that the new VINEA node may subscribe to their message updates.

**Enrollment:** we define by enrollment the procedure of authentication and policy exchange among ISD, NMS and the new VINEA node. Only the VINEA nodes enrolled in a private InP overlay are allowed to later exchange CADE messages to participate in a virtual network embedding. We say that the InP overlay is private as it uses customized private addresses. VINEA nodes do not process incoming CADE messages whose source is not a member of a private InP overlay. \(^1\) The ISD service, when queried with a slice identifier, returns the (IP) address of the manager of the InP overlay that is currently or has previously embedded a given slice. We implemented the ISD service as a single centralized synchronized database. Each VINEA node has an interface to query the ISD service. The modularity of our prototype however enables alternative (distributed) ISD implementations: we envision a more scalable scenario with many peer ISDs, each one containing a partially replicated subset of

---

\(^1\) A private network is merely a collection of application processes that maintain the set of shared states over a certain scope. In our case, such states are those useful to embed a virtual network using our embedding protocol, and the scope is defined by all the VINEA nodes participating in such embedding.
all the objects, communicating to retrieve the queried object. In such distributed cases, a request to the ISD is forwarded across the peer ISDs until the destination application (the ISD process that contain the sought NMS address is found), or until a predefined termination condition is met (a policy to limit the discovery overhead).

**Physical Resource Discovery:** if we set the InP process bidding function to be equivalent to the residual capacity, by only exchanging the bids on virtual nodes with their neighbors, the InP processes automatically discover the available resources.

### 6.3.2 Virtual Network Mapping Service

This service is responsible for deciding which InP process hosts which virtual node, and what physical loop-free path hosts each virtual link using the CADE protocol (Chapter 5). After the InP overlay bootstrapping phase, InP processes subscribe to the slice objects to be released by a service provider, and bid on virtual resources as they receive embedding requests. In our implementation, we have assumed that the service providers’ names are known after reading them from an InP process configuration file, while their addresses are dynamically resolved with DNS. We could however also acquire the name (or address) of the service provider from the NMS during the bootstrapping phase.

**Election Phase.** We implemented the MAD and SAD node embedding policies, using the node residual capacity as a utility function. We have an Utility package that can be enriched with customized node and link embedding policies. Each InP process can load its private bidding function policy from its configuration file.

**Agreement Phase:** in our system implementation of the CADE protocol, we relaxed the assumption of a synchronous conflict resolution phase. By relaxing this
assumption, the synchronous rules in Table A.1, used for our simulations became invalid to correctly update the InP process states on a asynchronous distributed virtual network embedding protocol. In particular, for an asynchronous conflict resolution, we needed to a (i) concept of re-broadcasting a message, and (ii) a new concept of time-stamp $t$, that is, the time at which the bid was generated, as opposed to the time $s$ at which a vote message is received in the synchronous version of CAD.

When a VINEA vote message is sent from InP process $k$ and received by InP process $i$, the receiver follows the rules in Tables B.1 and B.2 to resolve the allocation conflicts asynchronously. If none of the conditions in such conflict resolution tables is met, the receiver InP process $i$ applies the default rule, that is “leave” its states as they are without broadcasting any update. We denote with $b_{ij}$ the value of the utility known by InP process $i$ on virtual node $j$, while with $t_{ij}$ we denote the time at which the utility value on virtual node $j$ was generated by InP process $i$. $\epsilon$ is a small positive number. For a correct asynchronous agreement phase, the receiver InP process may need to rebroadcast (propagate) the sender states, or the receiver states. In particular:

- If rebroadcast is coupled with \texttt{leave} or with \texttt{update}, the receiver broadcasts its own CADE states.

- If rebroadcast is coupled with \texttt{update} or with \texttt{reset}, the receiver broadcasts the sender’s states.

In order to reduce the message overhead when rebroadcasting, \textit{i.e.}, to avoid rebroadcasting redundant information, we have several rebroadcasting cases:

1. **Update and rebroadcast**: the receiver InP process updates its allocation vector $a_{ij}$, the winning utility value $b_{ij}$, and the time $t_{ij}$ at which the highest utility value was generated with the received information from the sender InP
process $k$. Then it rebroadcasts this updates, and, in case the embedding policy dictates it (e.g., in MAD), also the new winner identity $a_{ij}$.

2. **Leave and rebroadcast**: the receiver InP process does not change its information state, but rebroadcast its local copy of the winning node information to look for confirmation from another InP process.

3. **Leave and no rebroadcast**: this is the default option. The receiver InP process does not update any of its states and does not rebroadcast anything. This action is applied when it is clear that the received bid message is identical to the existing information.

4. **Reset and rebroadcast**: due to messages arrived out of order and to the fact that CADE releases bids subsequent to an outbid virtual node, the receiver InP process received some confusing information and resets its states as follows: the allocation vector and the time stamp are set to none and null, respectively, and the bid is set to zero. After that, the original sender information is rebroadcasted so that the confusion can be resolved by another InP process.

5. **Update time and rebroadcast**: the receiver InP process receives a possibly confusing message. The receiver updates the timestamp on its bid to reflect the current time, confirming that it still thinks it is the winner. This helps to resolve situations of vote messages arriving out of order. For example, assume that InP process 1 sends a vote message at time $t_1$, with a utility value $b_1$. Before this message reaches InP process 2, InP process 2 votes on the same virtual node at time $t_2$, with an associated utility, $b_2$; where $t_2 > t_1$ and $b_1 > b_2$. Now assume that the vote message of InP process 1 arrives at InP process 3 first. InP process 3 updates its states with this information. But just after the
update, InP process 3 receives also the utility value from InP process 2, which was lower but forged at a later time. So InP process 3 does not know if the utility value of InP process 2 was made with knowledge of InP process 1 or not. Therefore, simply updating the timestamp with the message creation time is not enough to correctly and safely implement VINEA in an asynchronous setting. Hence we need to rebroadcast the latest sender information.

The complete set of VINEA conflict resolution rules are reported in Appendix A.

Once a mapping is found, the InP processes that initially had received the slice request respond to the service provider, that, if the response is positive, releases the next virtual network to be embedded, or the next virtual network partition, else it terminates and logs the failed embedding.

**Virtual Network Allocator Service**

Each VINEA node has an interface to the Virtual Network Allocator Service. We provide a Mininet-based [LHM10] implementation for the final binding between physical and virtual resources. When an InP process returns a positive embedding response to the service provider, indicating that an embedding of the slice has been successfully found, the Virtual Network Embedding Service parses the input from a *Slice* object, and uses the Mininet library to generate and bootstrap a virtual network.

**Resource Binding Implementation**

For each InP process hosting at least one virtual node, we need to fork a virtual switch, and attach a virtual host to it (Figure 6-2). For any InP process there exists a virtual switch, and for any virtual node hosted on that InP process, the VINEA allocation service creates a separate interface to the same virtual switch. A virtual switch is implemented using the Open Virtual Switch reference implementation libraries [OVS13]. We use the Mininet 2.0 default Open Virtual Switch (OVS) con-
Figure 6·2: (left) InP overlay with three embedded virtual networks. (right) VINEA resulting configuration: for each InP process hosting a virtual node there is a virtual switch and at least a virtual host attached to it. An OVS Openflow controller is also attached to each virtual switch.

troller, that supports up to 16 switches. By leveraging the Mininet interface, VINEA can also configure CPU limits for each virtual host.

After setting up all virtual hosts and virtual switches, the allocation service configures support for SSH, so that an application running on top of the virtual network can log into each virtual host (our default settings do not require any password). Finally, the virtual links are set up connecting the virtual switches, and the virtual hosts to the virtual switches. For each virtual link, a bandwidth limit can be set up using the Linux traffic control tc system call [Lin13], introducing traffic shaping constraints, and emulating delay and losses on virtual links as needed.
6.4 VINEA Testbed

In order to evaluate our prototype, we implemented a testbed whose architecture is shown in Figure 6.3. Our base system is a host running an Ubuntu distribution of Linux (version 12.04). The InP overlay is emulated via TCP connections on the host loopback interface. Each InP process includes the VINEA modules. Each emulated virtual node is a user-level process that has its own virtual Ethernet interface(s), created and installed with `ip link add/set`, and attached to an Open vSwitch [OVS13] running in kernel mode to switch packets across virtual interfaces. A virtual link is a virtual Ethernet (or `veth`) pair, that acts like a wire connecting two virtual interfaces, or virtual switch ports. Packets sent through one interface are delivered to the other, and each interface appears as a fully functional Ethernet port to all system and application software. The data rate of each virtual link is enforced by Linux Traffic Control (tc), which has a number of packet schedulers to shape traffic to a configured rate. Within the generated virtual hosts, we run real Linux applications, e.g. `ping`, and we measure the reserved bandwidth performance with `iperf` between the virtual hosts.

**Emulation Setup:** in all our experiments, an Ubuntu image was hosted on a VirtualBox instance within a 2.5 GHz Intel Core i5 processor, with 4GB of DDR3 memory. We start our InP overlay configuring each VINEA node, and we launch one or multiple virtual network requests with different size and topologies. We tested the embedding of virtual networks up to 16 virtual nodes, with linear, star (hub-and-spoke), tree and full virtual network topologies. The limit number of virtual nodes was imposed by the Mininet default built-in controller. By default, Mininet runs Open vSwitch (OVS) in OpenFlow mode, i.e., it requires an OpenFlow controller. Each of the controllers supported by Mininet turns the OVS switches into Ethernet

---

2Mininet comes with built-in controller classes to support several controllers, including the OpenFlow reference controller and Open vSwitch’s ovs-controller.
Figure 6-3: Testbed Architecture: A physical machine running Linux Ubuntu (version 12.04) hosts the VINEA prototype. Physical wires are emulated with loopback TCP connections on well-known ports. After the virtual networks are embedded, we can run Linux applications between virtual nodes, e.g., ping, traceroute, or we can send data traffic, and measure the reserved bandwidth performance with iperf.

bridges (learning switches). Using the command `route add`, we set up the default route for each virtual node following the requested connectivity.

6.5 VINEA Prototype Evaluation

The goal of this section is to show how, in real settings, different embedding policies may lead to different embedding performance — success rate — across representative virtual network topologies: linear, star, tree, and fully connected (Section 6.5.1). We also dissect the architecture components responsible for the embedding protocol overhead, and compare against two representative embedding policies (Section 6.5.2). We recently surveyed existing embedding solutions [EMI13], and to our knowledge, we are the first to release a system architecture that solves the virtual network embedding problem with its three phases, therefore no comparison with existing approaches has been possible. We compare however SAD and MAD, the two representative em-
Figure 6.4: Five InP processes connected with a linear topology hosting: (left) VNs with linear virtual topology and (right) VNs with full virtual topology. The SAD policy balances the load on the InP processes, and permits an higher acceptance rate as long as only a single loop-free physical path is available.

Figure 6.5: Five fully connected InP processes hosting: (left) VNs with a linear virtual topology and (right) VNs with full virtual topology. MAD is outperformed as collocating more virtual nodes on the same InP process quickly exhausts the available physical link capacity: multiple outgoing virtual links have one end hosted on the same InP process. If the link embedding policy allows multiple physical loop-free paths, the MAD policy performs better.
bedding policies described in Chapter 4.

**Virtual and Physical Network Models:** we vary the virtual network size from 2 till the limit of 16 is reached and we tested VINEA on InP overlays of size 2, 5, and 10 InP processes (without including the ISD, NMS and DNS separate host processes), with both linear and fully connected physical topologies. We only show results for InP overlay of size 5. The other results are similar. We randomly assign physical link capacities between 50 and 100 Mbps, then we assign the InP process capacity to be the sum of its outgoing physical link capacities. We specify the capacities in the InP process configuration file. We then assume the virtual link capacity to be randomly chosen between 1 and 5 Mbps. The virtual node capacity of a virtual network request is assigned to be the sum of its outgoing virtual links. Embedding performance are shown with a 95% confidence interval, while the overhead results refer to a single run.

**Utility model:** all InP processes use the same utility (bidding) function. The goal of the experiment is to embed a set of 100 virtual networks, with one second inter-arrival time between requests, aiming to reach Pareto optimality

$$U = \max \sum_{i=1}^{N_p} \sum_{j=1}^{N_v} b_{ij} x_{ij},$$

subject to the embedding constraints, that is, the distributed algorithm aims to maximize the sum of the utility of every InP process. $N_p$ is the number of InP processes, $N_v$ the number of virtual nodes, $b_i$ the bidding (utility) function used by InP processes, and $x_{ij} = 1$ if an InP process $i$ is hosting virtual node $j$ and zero otherwise. Similarly to previous embedding (centralized) heuristics [ZA06, YYRC08], in attempt to maximize the number of hosted virtual nodes while keeping the physical network load balanced, each InP process bids using its current load stress, *i.e.*, $b_i$ is the sum of the residual InP process capacity, plus the sum of the residual capacity of all its adjacent physical links.

**VINEA evaluation metrics.** Our prototype is evaluated within our single laptop testbed across two different metrics: the efficiency of the embedding and the message
overhead. By efficiency we mean the virtual network allocation ratio, \textit{i.e.}, the ratio between allocated and requested virtual networks. When computing the message overhead, we measured the actual number of bytes exchanged across the InP overlay. In particular, we dissect the message overhead produced by the three VINEA node types: (i) the service provider, (ii) the InP processes responsible to forward the requests and to respond to the embedding request, and (iii) the other InP processes merely participating in the embedding.

6.5.1 Embedding Success Rate

We conduct a set of experiments to demonstrate how both the service provider partitioning policy, and the InP process auction policies can be instantiated to tune the load on each InP process, and therefore to adjust performance of the applications running on top of the embedded virtual networks. We summarize our prototype evaluation findings on the embedding success rate into the following three key observations:

(1) \textit{The success rate improvement when using SAD with respect to the MAD policy decreases as the number of virtual links to embed increases, and a single physical loop-free path is available.} When a single (shortest) physical path is available, the SAD embedding policy better balances the virtual capacity load, increasing thus the number of accepted virtual network requests. This is because in MAD, multiple virtual nodes hosted on the same InP process require multiple outgoing virtual links to be hosted on the same physical outgoing link, quickly exhausting its available capacity. The load balancing advantage diminishes as the number of physical links to embed increases (Figure 6·5a and 6·5b).

(2) \textit{MAD improves the allocation ratio as the number of virtual links to embed increases, and multiple physical paths are available.} When the virtual links to embed are limited, \textit{e.g.} in a virtual network with linear topology, and the physical capacity
of multiple paths is available, the performance of MAD and SAD are comparable (Figure 6-5c). When instead the number of virtual links to embed increases, e.g. in a fully connected virtual network, the advantage of having multiple physical paths that can host the virtual link requested capacity becomes more relevant, and MAD shows higher embedding performance. This is because virtual links departing from the same InP process have multiple physical link capacity, and virtual links across virtual nodes hosted on the same InP process do not occupy outgoing physical link capacity (Figure 6-5d).

(3) The number of virtual nodes or links to allocate significantly impacts the virtual network allocation ratio. This (sanity-check) result is unsurprising. Comparing the virtual network embedding success rate results across different virtual network topologies, we observe that the allocation ratio decreases when we increase the number of virtual links to embed. Moreover, the allocation ratio always decreases as we attempt to embed virtual networks with more virtual links (Figures 6-5a to 6-5d).
Figure 6.7: InP processes receiving the request have higher overhead as they propagate the virtual network objects, including their bids, and then respond to the SP.

### 6.5.2 Overhead

In this section we show how the MAD policy has lower overhead than the SAD policy, as no virtual network partitioning is needed from the service provider (Figure 6.6). This result demonstrates how an SP can significantly limit the network overhead by selecting a single InP process to send their requests. The result is in contrast with other approaches [ZXB10, CSB10] in which an SP also assumes competition among InPs, but sends the same virtual network request to multiple federated InPs and then selects the best (e.g. the cheapest) embedding solution.

When all InP processes are silent, or when all virtual network requests have timed out, we say that a convergence state has been reached. We measured the embedding overhead of reaching the convergence state after an embedding request. In particular, we attempt to embed a set of virtual networks with linear topology, increasing the number of virtual nodes (in a range $[2, 16]$), onto a linear InP overlay topology of 3 InP processes. The request is sent from a fourth node acting as SP. In this experiment, when using the SAD policy, the SP sends 9 virtual network partitions
to a single InP process (InP1). InP1 then informs the other two InP processes (InP2 and InP3) about the request, together with its first bid on each partition. After the distributed virtual network mapping phase, InP1 sends an Embed Success message. When received, the SP releases the next partition. In this experiment, InP2 can always overbid InP1 and so it propagates its bid to the other two InP processes. The third InP process is never able to overbid, therefore it does not produce any overhead. Note how, since the physical topology is linear, InP3 does not need to rebroadcast to its only physical neighbor after receiving a bid update from it.
Chapter 7

Conclusions

7.1 Summary of Contributions

Network virtualization has unlocked several opportunities for both infrastructure and service providers: the infrastructure and software as a service paradigms are generating billions of dollars in revenue every year. At the same time, the cost of providing and managing such wide-area virtual network services has also increased along with their complexity. One of the major challenges in sharing the underlying physical network is in fact to adapt it to different requirements of providers and customers’ goals.

In this thesis we proposed a policy-based architecture for the virtual network embedding problem, one of the most challenging mechanisms in wide-area virtual network service management. In particular, we propose a distributed embedding mechanism whose policies can be instantiated to adapt to different providers’ embedding goals. We have shown how our mechanism outperforms existing distributed solutions, and provides guarantees on both embedding optimality and convergence time.

We have also defined an object model for our mechanism, as a base for a protocol specification, and we have tested and released our prototype implementation within
a Mininet-based virtual network testbed. To our knowledge, we are the first to implement a policy-based architecture prototype for the virtual network embedding problem.

7.2 Future Work

We believe many interesting questions are left open as a follow up of this work. We give few examples of interesting research problems in the field of algorithmic design, and system implementation perspectives.

**Mechanism Design:** different utility functions can be used to design several embedding mechanisms, in which InPs and SP may cooperate or compete to achieve a common or a selfish embedding goal. An example of research problem would be to analyze and evaluate the performance of a distributed embedding protocol with (selfish) bidders, trying to win multiple slices in a single resource embedding round, modeling the scheme as an auction. Auction-based resource allocation schemes without currency have been floated before, as described in our related work chapter; we believe however that it is worth investigating the auction theory literature to provide, *e.g.*, incentive compatible auction and pricing mechanisms, in which physical nodes could maximize their profit and at the same time have incentives to bid their truthful value on each virtual resource.

**System Implementation:** designing and implementing a platform for scalable, isolated and reproducible distributed system experiments is still an open question, although the GENI community has made significant progress. We believe that an interesting research problem that would leverage our prototype implementation could be to provide a GENI “reproduce experiment” button, in which the same virtual network application is run on top of different physical resources.
**Formal Verification of Protocols:** we believe that an interesting research direction would also be to use formal method tools such as Alloy [Dan13] or Isabelle [WW07], to show correctness of our consensus-based embedding mechanism.
Appendix A

CADE Synchronous Agreement Rules

In this appendix we report the conflict resolution rules used in the agreement phase of the synchronous CADE protocol. This rules were inspired by the Consensus-based Decentralized Auction (CBBA) algorithm used for decentralized robot task allocation [CBH09].

As defined in Chapter 4, a virtual network is denoted by the graph $H = (V_H, E_H)$ and a physical network by $G = (V_G, E_G)$, where $V$ is a set of (physical or virtual) nodes, and $E$ the set of (physical or virtual) edges. $b_i \in \mathbb{R}^{\lvert V_H \rvert}$ is the a vector of utility values. Each entry $b_{ij} \in b_i$ is a positive real number representing the highest utility value known so far on virtual node $j \in V_H$. $a_i \in V_G^{\lvert V_H \rvert}$ is the winner vector—a vector containing the latest information on the current assignment of all virtual nodes, for a distributed auction winner determination. $a_{ij} \in a_i$ is contains the identity of the winner of virtual node $j$, as currently known from physical node $i$. $s_i \in \mathbb{R}^{\lvert V_G \rvert}$ is the a vector of time stamps of the last information update from each of the other physical nodes i.e., the message reception time. There are three possible action when a physical node $i$ receives a vote message from a sender physical node $k$: (i) update, where both the utility vector and the allocation vector are updated according to the sender information; (ii) reset, where the utility value is set to zero,
Table A.1: Rules table for cade synchronous conflict resolution. The sender physical node is denoted with $k$, and the receiver physical node with $i$. The time vector $s$ represents the time stamp of the last information update from each of the other agents.

and the allocation vector to null, and (iii) **leave**, where both the utility vector and the allocation vector are left unchanged by the receiver physical node.
Appendix B

VINEA Asynchronous Agreement Rules

In this appendix we report the conflict resolution rules used in the vinea asynchronous implementation of the cade protocol.

The allocation vector \( a \) and the utility vectors \( b \) are defined in (Chapter 4 and in) Appendix A. The time stamp vector \( t_i \in \mathbb{R}_{+}^{\left| V_H \right|} \) is a vector of time stamps where each entry \( t_{ij} \in t_i \) is a positive real number representing the forging time of the bid on virtual node \( j \) as currently known from physical node \( i \). This vector is necessary for an asynchronous conflict resolution.
Table B.1: Rules table for vinea asynchronous conflict resolution. The sender physical node is denoted with $k$, and the receiver physical node with $i$ (Table 1 of 2).
<table>
<thead>
<tr>
<th>$k$ thinks $a_{kj}$ is</th>
<th>$i$ thinks $a_{ij}$ is</th>
<th>Receiver’s action (default leave &amp; no broadcast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m \notin {i, k}$</td>
<td>$i$</td>
<td>if $b_{kj} &gt; b_{ij}$ → update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $b_{kj} = b_{ij}$ and $a_{kj} &lt; a_{ij}$ → update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $b_{kj} &lt; b_{ij}$ → update time and rebroadcast</td>
</tr>
<tr>
<td></td>
<td>$k$</td>
<td>if $b_{kj} &lt; b_{ij}$ → update and rebroadcast (sender info)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $t_{kj} &gt; t_{ij}$ → update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $t_{kj} &lt; t_{ij}$ → leave and rebroadcast</td>
</tr>
<tr>
<td>$n \notin {i, k, m}$</td>
<td></td>
<td>if $b_{kj} &gt; b_{ij}$ and $t_{kj} \geq t_{ij}$ → update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $b_{kj} &lt; b_{ij}$ and $t_{kj} &lt; t_{ij}$ → leave and rebroadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $b_{kj} &lt; b_{ij}$ and $t_{kj} &gt; t_{ij}$ → update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $b_{kj} &gt; b_{ij}$ and $t_{kj} &lt; t_{ij}$ → leave and rebroadcast</td>
</tr>
<tr>
<td>none</td>
<td></td>
<td>update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td>$i$</td>
<td>leave and rebroadcast</td>
</tr>
<tr>
<td></td>
<td>$k$</td>
<td>update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td>$m \notin {i, k}$</td>
<td>update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td>none</td>
<td>leave and no rebroadcast</td>
</tr>
</tbody>
</table>

Legend

- rebroadcast with leave or update time, broadcast receiver states
- with update or reset, broadcast sender states

Table B.2: Rules table for vinea asynchronous conflict resolution. The sender physical node is denoted with $k$, and the receiver physical node with $i$ (Table 2 of 2).
Appendix C

Physical Node Configuration File

C.1 Physical Node Configuration File

vinea.pnetwork.name = PhysicalNetwork1
vinea.pnode.userName = BU
vinea.pnode.passWord = BU

#this is the local TCP port this IPC is going to listen to
TCPPort = 21115

vinea.pnode.name = pnode1

#neighbors
#every vinea node should be connected to the ISD
neighbour.1 = isd1
neighbour.2 = sliceManagerIPC
neighbour.3 = pnode3

##### DNS configuration #####
dns.name = localhost
dns.port = 21111

##### PN authentication #####
enrollment.authenPolicy = AUTH_PASSWD
##### ISD configuration ######
vinea.isd.name = isd1

### cade Policies ###
# unique pnode id
cade.id = 3

# SAD, MAD or write your own
cade.allocationPolicy = MAD

# least or most informative
cade.assignmentVectorPolicy = most

# bid vector length
enade.bidVectorLength = 10

# bid (utility) function
cade.nodeUtility = residual_node_capacity

# authenticator
cade.owner = sliceManagerIPC

# service provider to subscribe
cade.mySP = sp

# physical link capacity for to be split among all outgoing hosting flows
cade.outgoingLinkCapacity = 100

# physical link capacity for to be split among all incoming hosting flows
cade.incomingLinkCapacity = 100
C.2 Service Provider Configuration File

vinea.pnode.name = sp
vinea.pnetwork.name = PhysicalNetwork1
vinea.pnode.userName = BU
vinea.pnode.password = BU
TCPPort = 11112

#neighbors needs to be set up in advance, like the wires
neighbour.1 = idd
neighbour.2 = InPManager
neighbour.3 = pnode1
neighbour.4 = pnode3
neighbour.5 = pnode2

### DNS

dns.name = localhost
dns.port = 21111

vinea.enrollment.authenPolicy = AUTH_PASSWD

## For ISD, only name is needed, the port is queried to the DNS
vinea.isd.name = isd

# trusted pnodes
vinea.sp.trusted.1 = pnode1
#vinea.sp.trusted.2 = pnodeName ...

# node embedding policies
vinea.sp.vnode.auction = sad

# partition size
vinea.sp.partitionSize = 1

# service provider timeout in seconds (if set to -1 waits for response indefinitely)
vinea.sp.timeout = -1
Bibliography


Curriculum Vitae

Flavio Esposito

CONTACT INFORMATION

E-mail: flavio.esposito@gmail.com

EDUCATION

- Ph.D. in Computer Science, Boston University Oct, 2013
  College of Arts and Sciences, Boston, Massachusetts, USA.
  Area: Computer networking and distributed systems.

- Master of Science in Telecommunication Engineering Dec 2005
  University of Florence, Italy. Undergraduate full scholarship awarded.
  University of Oulu, Finland. Master thesis development. (Erasmus scholar).

RELEVANT WORKING EXPERIENCE

- Computer Science Department, Boston University Jan 2007- Aug 2013
  Boston, MA

  - Research Assistant Conduct research on policy-based architectures for distributed systems.

  - Teaching Assistant Taught lecture and laboratory sections for Computer Networks, Introduction to Internet Technologies and Web Design, Introduction to Computer Science, Introduction to (Java) programming. Graded student exams and laboratory assignments.

- Bell Laboratories, May 2009 - Aug 2009
  Holmdel, NJ (Research Intern)

  - Conduct research on distributed resource discovery and allocation in cloud computing.
• Raytheon BBN Technologies, Cambridge, MA (Research Intern) May 2008 - Aug 2008
  – Conduct research on virtualization over different technologies for large-scale testbed services.

• Child Cognitive Laboratory, Boston University (Volunteer Research Assistant) Fall 2012
  – Design and Development with Eng.Filippo Poggini of a CakePHP platform to collect real time web user responses.

  – Design and implementation of scheduling algorithms for peer-to-peer networks.

• Centre for Wireless Communication, Oulu, Finland (Res. Scientist) Oct 2006 - Jan 2007
  – Design and implementation of MAC protocols on a real-time operating system of UWB radios.

• Alcatel-Lucent, Florence, Italy (Bid Manager) Jan 2006 - Oct 2006
  – Sales support for telecommunications subsystems mainly on wireless and oil & gas markets.

MediaTeam Oulu, Oulu, Finland (Research Intern) Mar 2005 - Nov 2005
  – Design and implementation of Java and C++ applications to enable discovery and handover between GPRS and Bluetooth interfaces for seamless, RFID-triggered connectivity in heterogeneous mobile networks.

PERSONAL SKILLS AND COMPETENCES

Languages: Italian (native), English (fluent)
Programming: Java, Python, C/C++, Perl, Bash, ATS (func. progr.), HTML, CSS, PHP.
Database: MySQL, PostgreSQL.
Other Tools: R, MATLAB, Sage, NS, User Mode Linux, Xen, OpenFlow, Mininet, OpenStack.
Platforms: Unix, VxWorks (RTOS).
SERVICE

- Computer Science department representative for the Graduate Student Organization from 2010 till 2013
- Technical Program Committee (TPC) member of DCOSS: the IEEE International Workshop on Internet of Things — Ideas and Perspectives (IoTIP-13)
- Technical Program Committee (TCP) member of ACVR: the First International Workshop on Assistive Computer Vision and Robotics — in conjunction with the 17th International Conference on Image Analysis and Processing (ICIAP)
- Technical Program Committee (TPC) member of the IEEE International Workshop on Seamless Connectivity in Vehicular Networks (SCVN 2011)
- Volunteer Organizer of ACM Internet Measurement Conference (IMC) 2012

AWARDS

- Undergraduate Scholarship. Azienda Regionale per il Diritto allo Studio Univ. (ARDSU), 1998.

MEMBERSHIP

- Member of the Italian Professional Registry of Engineers since March 2006.
- IEEE member since 2006.
Publications


4. Y. Wang, F. Esposito, and I. Matta, Demonstrating RINA using the GENI Testbed. The Second GENI Research and Educational Experiment Workshop (GREE2013), Salt Lake City, Utah 21-22 March 2013.


15. F. Esposito, F. Chiti and R. Fantacci, Voice and Data Traffic Classes Management for Heterogeneous Networks Enabled by Mobile Phones, in Proc. of INFOCOM, April 2006.
