VINEA: An Architecture for Virtual Network Embedding Policy Programmability

Flavio Esposito, *Member, IEEE,* Ibrahim Matta, *Senior Member, IEEE,* and Yuefeng Wang, *Member, IEEE,*

Abstract—Network virtualization has enabled new business models by allowing infrastructure providers to lease or share their physical network. A fundamental management problem that cloud providers face to support customized virtual network (VN) services is the virtual network embedding. This requires solving the (NP-hard) problem of matching constrained virtual networks onto the physical network.

In this paper we present VINEA, a policy-based virtual network embedding architecture, and its system implementation. VINEA leverages our previous results on VN embedding optimality and convergence guarantees, and it is based on a network utility maximization approach that separates policies ($i.e.$, high-level goals) from underlying embedding mechanisms: resource discovery, virtual network mapping, and allocation on the physical infrastructure. We show how VINEA can subsume existing embedding approaches, and how it can be used to design novel solutions that adapt to different scenarios, by merely instantiating different policies. We describe the VINEA architecture, as well as our object model: our VINO protocol and the API to program the embedding policies; we then analyze key representative tradeoffs among novel and existing VN embedding policy configurations, via event-driven simulations, and with our prototype implementation. Among our findings, our evaluation shows how, in contrast to existing solutions, simultaneously embedding nodes and links may lead to lower providers' revenue. We release our implementation on a testbed that uses a Linux system architecture to reserve virtual node and link capacities. Our prototype can be also used to augment existing open-source "Networking as a Service" architectures such as OpenStack Neutron, that currently lacks a VN embedding protocol, and as a policy-programmable solution to the "slice stitching" problem within wide-area virtual network testbeds.

Index Terms—Network Virtualization, Virtual Network Embedding, Network Management, Cloud Computing.

1 Introduction

Network virtualization enables multiple virtual instances to co-exist on a common physical network infrastructure. This paradigm has opened up new business models, enabling infrastructure providers to lease or share their physical resources, or to virtualize in-network hardware or middleboxes to build a so called Virtual Network Function (NFV). Each virtual network (VN) is customizable in support of a wide range of customers and applications. One of the fundamental management protocols, not yet standardized, that cloud providers need to run to support such services is the VN embedding protocol. [1] Running such protocol requires solving the NP-hard problem of matching constrained VNs on the physical network (overlay), owned by a single provider, or by multiple federated providers. The VN embedding problem, defined in § 3, consists of three interacting (and customizable, or policy-programmable) mechanisms: (i) resource discovery, where the space of available, potentially hosting or physical (or overlay) resources[2], is sampled or exhaustively searched; (ii) virtual network mapping, where a subset of the available physical resources is chosen as a candidate to potentially host the requested VN, and (iii) allocation, where each virtual node is bound to at least a physical node, and each virtual link to at least one loop-free physical path.

Existing embedding solutions focus on specific policies under various settings. Policies ($i.e.$, high-level goals) parametrize any of the three embedding mechanisms. For example, some centralized heuristics, devised for small enterprise physical networks, embed virtual nodes and virtual links separately, to adapt the load on the physical network resources with minimal virtual machine or path migrations [1]. Other solutions show how the physical network utilization may increase by simultaneously embedding virtual nodes and links [2]. Distributed solutions for embedding wider-area virtual networks also exist [3], [4], [5], [6]. Some of them outsource the embedding to a centralized Service Provider (SP) that coordinates the process by either splitting the VN request and sending it to a subset of Infrastructure Providers (InPs) [3], or by collecting resource availability from InPs and later offering an embedding [6]. Outsourcing the embedding has the advantage of relieving InPs from the entire management complexity, but a single centralized authority is non-geographically distributed, and could be untrusted, or a single point of failure. On the other hand, centralized solutions are more suitable for smaller enterprise physical networks, where controllability is more important than scalability.

Both centralized and distributed existing solutions are also limited to a single distribution model — the type and amount of information propagated for the embedding — and are often restricted to a subset of the three embedding mechanisms. For example, some do not consider the discovery phase [1], [7], assuming full knowledge of the physical resource availability,

- *F. Esposito is a member of the Advanced Technology Group at Exegy, Inc. St.Louis, MO. E-mail: fesposito@exegy.com*
- *I. Matta and Y. Wang are with the Computer Science Dept. at Boston University, Boston, MA E-mail: {matta,wyf}@cs.bu.edu.*

Manuscript received April 3, 2015; revised XXXXXX.

1. We call Service Providers (SPs) the players that do not own the infrastructure but provide a cloud-based service. Infrastructure Providers (InPs) own instead the physical network resources. A cloud provider can be a lessor or a lessee of the network infrastructure, and can act as both service and infrastructure provider.

2. hosting nodes is a broader and more general term than physical node. A hosting node can be itself a virtual node in a recursive embedding. A service provider may itself virtualize the rented physical resources.

while others leave the final allocation decision to the virtual network requester, that chooses the hosting physical resources among a set of candidates [8]. Other solutions attempt to adapt the embedding with policy instantiations, but their policy set is limited to the notion of provider utility [4], or merely consider node embedding policies, forcing infrastructure providers to relay embedding virtual paths, and only consider synchronous (unrealistic) scenarios [5].

Our Contributions. In summary, due to the wide range of VN applications, providers' goals and allocation models ($e.g.$, best effort or SLA), a system that provides theoretical guarantees on the embedding optimality [5], tackles the complete VN embedding with its three phases (of resource discovery, VN mapping, and allocation), and is able to adapt *via policy programmability* to different service and infrastructure provider's goals, to our knowledge, does not yet exist. [3] To this aim, we present VIrtual Network Embedding Architecture (VINEA), an architecture that via our object model and API allows VN embedding policy-programmability, and we make the following contributions:

We discuss some related work (§ 2) and define the VN embedding problem using optimization theory as a network utility maximization problem, as well as identify its system's design challenges (§ 3.) Then we introduce our VINEA architecture, built as a *distributed application facility* on top of the first prototype of our RINA architecture [9], [10], [11]. In particular, we overview the main mechanisms required to embed a VN (§ 4), and we describe its components, as well as the object model —a baseline for a VN embedding protocol specification— that together with our novel VINO embedding protocol messages is used to maintain the states across the service and infrastructure provider processes. The VINO protocol is agnostic to the differences between inter and intra provider embeddings (§ 5).

Using our object model, as well as classical decomposition theory, we demonstrate with some examples how the VINEA embedding policies can be instantiated to accommodate novel embedding policies, as well as to subsume existing embedding approaches (§ 6 and § 7.) Such novel and existing embedding policies are compared analytically (§ 7.1), with an event-driven simulation campaign, and with our single-host VN embedding testbed containing our prototype (described in § 8.) Our prototype includes support for all three VN embedding mechanisms, for both service and infrastructure providers. To foster research on VN embedding policy programmability, we release our implementation within a testbed [12]. Our base system is a host running an Ubuntu distribution of Linux (version 12.04.) Each InP process includes the modules of our prototype, and an 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 it is attached to an Open vSwitch [13] running in kernel mode to switch packets across virtual interfaces (§ 8.1).

To obtain our simulation results, we used a dataset of 8 years of real VN embedding requests to the Emulab testbed. In our evaluation, both with simulation and with the prototype, we compare a representative set of the possible tradeoffs when attempting to embed multiple VNs of different size and topologies on physical networks that follow different connectivity models (Barabasi-Albert and Waxman), and use primal or dual decomposition techniques [14]. Among our findings, we show how, in contrast to existing VN embedding solutions: (i) partitioning a VN request not only increases the signaling overhead, but may decrease the embedding efficiency, implying a lower cloud providers' revenue, and (ii) simultaneous virtual node and link embedding may limit the number of allocated VNs (§ 9).

Our VINEA testbed, to our knowledge the first VN embedding testbed, can be used to experiment with novel embedding policies, or to run network applications within an emulated environment such as mininet [15]. VINEA can also be used as a flexible solution to the "stitching problem", $i.e.$, the problem of providing a virtual network testbed service using resources from federated and geographically distributed resource aggregates [16]. [4] Moreover, our prototype can augment existing open-source "Networking as a Service" solutions such as OpenStack Neutron [18], that currently lacks a VN embedding protocol.

2 Related Work

Network Policy Programmability. The rapid growth of cloud service markets has fostered significant efforts toward a standardization of protocols to manage networks of virtual machines. The flexibility that network virtualization introduced has enabled network managers to program policies for a wide set of mechanisms. Two examples are the CloudStack [19] and the OpenStack [20] initiatives. Those architectures involve storage, and computation, not only a network virtualization components. By using such architectures, users can customize their VNs, but providers cannot express VN embedding policy programmability as in our VINO object model. As in OpenStack Neutron [18], we also have an API to create and manage virtual networks (objects): we can create (embed), update, and delete VNs, but our prototype does not yet support subnet listing or filtering operations. However, in Neutron, a VN cannot be easily embedded on top of other VNs (recursive embedding.) A popular approach that allows the instantiation of the forwarding mechanisms by customized policies is OpenFlow [21], where a protocol regulates the injection of (high-level) rules into switches, thanks to a centralized controller. Other recent approaches have demonstrated the importance of policy instantiation for other mechanisms, $e.g.$, introducing a middleware to reprogram routing and forwarding policies in software-defined networks [22], [23], or even when the network is not virtualized, for transport [24] and routing protocols [25], [26], in both wireless [25] and wired [26] networks. Similar to all these approaches, VINEA's design focus is on the separation between mechanisms and policies. Our prototype also supports OpenFlow, but VINEA enables SPs and InPs to specify policies for the VN embedding mechanism, while OpenFlow and all other network management approaches operate *after* a (virtual) network has been mapped. Other virtualization-based network architectures have been prototyped, for a single [27], [28] or for multiple co-operating InPs, with [28] or without [27] virtual link performance guarantees. VINEA focuses on the architecture of virtual network embedding, and also provides guarantees, but on convergence time, $i.e.$, embedding response time, and embedding optimality.

3. Choosing the "right" policies when instantiating the VN embedding mechanisms is a challenge; a set of policies that fits every scenario probably does not exist. Our architecture helps provide insights into what VN embedding policy is best to use for each scenario, by permitting policy programmability and hence VN embedding tradeoff analysis.

4. We use the terms virtual network (VN) and slice to mean two different things: a slice is the implementation object which represents the VN but also other states: user credential, the VN's owner, etc [17]. In the literature, these two terms are often confused. For example, a slice is also used to indicate a virtual network isomorphic to the underlying physical network.

Although our system is built in support of such guarantees, the theoretical results are not a contribution of this paper, and are detailed in [5], [29].

Virtual Network Embedding Solutions. Despite the lack of a supporting implementation, approaches that allow InPs to collectively embed a VN already exist [3], [4], [6], [8], [30]; some of them focus on the desirable property of letting InPs use their own utility [4], while others rely on a virtual resource auction [5], [8]. Prototypes for the VN embedding problem have been proposed as well, but without support for policy programmability, both in cloud computing [31], and for wide-area testbed applications [32], for entire VNs [32], or limited to virtual switches [31]. With our VINEA prototype, we demonstrate in a real setting not only how InPs can choose their own embedding utility as in [4], but we study the tradeoff of a wider range of virtual node and link embedding solutions, $e.g.$, the impact of the VN request partitioning, which is hardwired in existing approaches. Moreover, to the best of our knowledge, we release the first VN embedding testbed.

3 The Virtual Network Embedding Problem

3.1 Network Utility Maximization

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 $C(e)$ [5], a virtual network embedding is the problem of: (i) discovering at least V_H virtual instances of some physical node(s), and E_H loop-free physical hosting paths, (ii) finding *at least* one mapping of H onto a subset of G, such that each virtual node is mapped onto at least one physical node, and each virtual link is mapped onto at least a loop-free physical path p, and (iii) assigning (or binding) one of the mappings to the physical resources, while maximizing some utility or minimizing a cost function. [6] The mapping function $\mathcal{M}$ can be decomposed into two functions: virtual node mapping ($\mathcal{M}_\mathcal{N}$) and virtual link mapping ($\mathcal{M}_\mathcal{L}$) where $\mathcal{M}_\mathcal{N} : V_H \rightarrow V_G$ is the virtual node mapping and $\mathcal{M}_\mathcal{L} : E_H \rightarrow \mathcal{P}$ is the virtual link mapping, and $\mathcal{P}$ denotes the set of all loop-free physical paths in G. $\mathcal{M}$ is called a *valid mapping* if all constraints of the request H are satisfied, and for each $l^H = (s^H, r^H) \in E_H$, we have $\mathcal{M}_\mathcal{L}(l^H) = (\mathcal{M}_\mathcal{N}(s^H), \ldots, \mathcal{M}_\mathcal{N}(r^H))$. Note how the definition is valid for VN embedding over a single provider, or across multiple federated providers, $i.e.$, it abstracts away the difference between intra- and inter-domain VN embeddings. Such differences may be captured by VINEA enforcing different VN embedding policies.

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 or centralized 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 our protocol. A natural objective for an embedding algorithm is to maximize some notion of *revenue*. The revenue can be defined in various ways according to economic models. We use the notion of a higher economic benefit (reward) from accepting a VN or virtual request that requires more resources ($e.g.$, bandwidth, CPU) from the physical network.

The utility function is a policy, and it can be chosen independently by each physical node. Different physical nodes may assign different weights to the components forming the utility, such as bandwidth and CPU. In [5] we show how, regardless of the choice of the utility $U_i \ \ \forall i$, our embedding mechanisms guarantee the distributed embedding convergence, as well as performance, assuming a submodular and monotonic utility.

3.2 Design Challenges

In this section we describe the VN embedding problem as a general network utility maximization problem. Previous models have used optimization theory to capture different objectives and constraints of the VN embedding problem (see $e.g.$ [2], [3].) Our model captures all three mechanisms of the VN embedding problem: resource discovery, virtual network mapping, and allocation. We begin the section by defining such mechanisms and describing some of the challenges associated with designing a distributed VN embedding solution.

Resource discovery is the process of monitoring the state of the substrate (physical or overlay) resources using sensors and other measurement processes. The monitored states include processor loads, memory usage, network performance data, etc. The major challenge in designing a resource discovery system is presented by the different VN's arrival rates and durations that the cloud provider might need to support: the lifetime of a VN can range from a few seconds (in the case of cluster-on-demand services) to several months (in the case of a VN hosting a GENI [16] experiment looking for new adopters to opt-in.) In wide-area testbed applications, VNs are provided in a best-effort manner, and the inter-arrival time between VN requests and the lifetime of a VN are typically much longer than the embedding time, so designers may assume complete knowledge of the network state, and ignore the overhead of resource discovery and the VN embedding time. On the other hand, in applications with higher churns, $e.g.$, cluster-on-demand such as financial modeling, anomaly analysis, or heavy image processing, where Service Level Agreements (SLAs) require short response time, it is desirable to reduce the VN embedding time, and employ limited resource discovery to reduce overhead.

Virtual network mapping is the step that matches VN requests with the available resources, and selects some subset of the resources that can potentially host the virtual network. Due to the combination of node and link constraints, this is the most complex of the virtual network embedding tasks. The problem is NP-hard [33]. These constraints include intra-node constraints ($e.g.$, desired physical location, processor speed, storage capacity, type of network connectivity), as well as inter-node constraints ($e.g.$, VN topology).

Designing a VN mapping algorithm is challenging. Within a small enterprise physical network for example, embedding virtual nodes and virtual links separately may be preferable to adapt to the physical network load with minimal virtual machine or path migrations [1]. If the goal is instead to increase physical network utilization, virtual network mapping solutions that simultaneously embed virtual nodes and links may be preferable [2], [34]. The heuristic used to partition the VN, that is the input of the VN mapping algorithm, changes the space of solutions, the embedding time, or both [29].

5. Each $C(e)$ could be a vector $(C_1(e), \ldots, C_\gamma(e))$ containing different types of constraints, $e.g.$ physical geo-location, delay, or jitter.

6. Even though it is feasible to assign a virtual node onto multiple physical nodes, and a virtual link onto multiple loop-free physical paths, the management complexity increases: packets flowing on different physical paths may arrive out of order, and virtual threads and memory would need to be pre-configured to be assigned onto specific physical CPUs, increasing context switching time.

Allocation involves assigning (binding) one set of all physical (or overlay) resources among all those that match the VN query, to the VN. If the resource allocation step fails, the matching step should be reinvoked. The allocation step can be a single shot process, or it can be repeated periodically to either assign or reassign different VN partitions, acquiring additional resources for a partial VN that has already been embedded (allocated).

The design challenges of VN embedding are both architectural, $i.e.$, who should make the binding decisions, and algorithmic, $i.e.$, how should the binding occur. A centralized third party provider can be in charge of orchestrating the binding process collecting information by (a subset of) multiple infrastructure providers [1], [3], [4], [6], or the decision can be fully distributed [5], [8], [30], using a broker [35], an auction mechanism [8], First Come First Serve [36], or maximizing some notion of utility, of a single service provider [1] or of a set of infrastructure providers [5].

In summary, the design space of a VN embedding solution is large and unexplored, and many interesting solutions and tradeoff decisions are involved in this critical cloud resource allocation problem.

The design challenges are exacerbated by the interaction among the three mechanisms (phases.) The VN embedding problem is a closed feedback system, where the three tasks are solved repeatedly; the solution at any given iteration affects the space of feasible solutions in the next iteration: the resource discoverer(s) return(s) a subset of the available resources to the VN mapper(s.) Subsequently, a list of candidate mappings are passed to the allocator(s), that decide(s) which physical (or overlay) resources are going to be assigned to each VN. After a successful binding, the allocator processes communicate with the resource discovery processes, so that future discovery operations are aware of the unavailable resources. This feedback-loop is necessary but not sufficient to guarantee optimal solutions: resources that were discovered and mapped may become outdated during the allocation due to network failures or state changes.

3.3 Modeling Virtual Network Embedding

We model the VN embedding problem with a network utility maximization problem. In particular, we assume that *Pareto optimality* is sought among physical nodes, possibly belonging to different infrastructure or cloud providers. We maximize $\sum_i U_i$, where U_i is a general utility function, measured on each hosting node i. Such function could depend on one, or all the VN embedding phases. In this paper we assume that U_i does not depend on the availability (and mapping) of other hosting nodes.

In our model we assume that a VN request j contains γ_j virtual nodes, and ψ_j virtual links. We model the result of the discovery mechanism with n_{ij}^P and p_{kj}, equal to one if the hosting node i, and physical loop-free path k, respectively, are available, and zero otherwise. We limit the overhead of the discovery of physical nodes and paths by A_N (constraint 1a) and $A_{\mathcal{P}}$ (constraint 1b), respectively. 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. Similarly, we model the VN mapping mechanism with other two binary variables, n_{ij}^V and l_{kj}, equal to one if a virtual instance of physical node i and physical loop-free path k, respectively, are assigned to the VN request j, and zero otherwise. Constraints (1i) and (1j) ensure that each VN request cannot be considered for allocation unless all virtual nodes and all virtual links requested have been mapped ($n_{ij}^V = 1$ and $l_{kj} = 1$).

Constraints $(1a)$ and $(1b)$ refer to the discovery, constraints $(1c)$ and $(1d)$ refer to the VN mapping, while $(1g)$ and $(1h)$ are the standard *set packing problem* constraints, and refer to the allocation, given a physical node capacity C_i^n, and the capacity of each loop-free physical path C_k^l. The VN embedding can be hence modeled as in Problem 1. Constraints $(1e-1f)$ and $(1i-1j)$ are called *complicating constraints*, as they complicate the problem binding the three mechanisms together; without those constraints, each of the VN embedding mechanism could be solved separately from the others, $e.g.$, by a different architecture component. The existential constraints $(1k)$ could be relaxed in the interval $[0,1]$; in this case, the discovery variables could represent the fraction of available resources, while the mapping and allocation variables could model partial assignments; it is feasible to virtualize a node on multiple servers, and a link on multiple loop-free physical paths [1].

$$\text{maximize} \quad \sum_{i=1}^{N_p} U_i(n_{ij}^P, p_{kj}, n_{ij}^V, l_{kj}, y_j)$$

$$\text{subject to} \quad \sum_{i\in N_p}\sum_{j\in J} n_{ij}^P \leq A_N \tag{1a}$$

$$\sum_{k\in\mathcal{P}}\sum_{j\in J} p_{kj} \leq A_{\mathcal{P}} \tag{1b}$$

$$\sum_{i\in N_p} n_{ij}^V = \gamma_j \ \ \forall j \tag{1c}$$

$$\sum_{k\in\mathcal{P}} l_{kj} = \psi_j \ \ \forall j \tag{1d}$$

$$n_{ij}^V \leq n_{ij}^P \ \ \forall i \ \ \forall j \tag{1e}$$

$$l_{kj} \leq p_{kj} \ \ \forall k \ \ \forall j \tag{1f}$$

$$\sum_{j\in J} n_{ij}^V y_j \leq C_i^n \ \ \forall i \tag{1g}$$

$$\sum_{j\in J} l_{kj} y_j \leq C_k^l \ \ \forall k \tag{1h}$$

$$y_j \leq \tfrac{1}{\gamma_j}\sum_{i\in N_p} n_{ij}^V \ \ \forall j \tag{1i}$$

$$y_j \leq \tfrac{1}{\psi_j}\sum_{k\in\mathcal{P}} l_{kj} \ \ \forall j \tag{1j}$$

$$y_j, n_{ij}^P, p_{kj}, n_{ij}^V, l_{kj}, \in \{0,1\} \ \ \forall \ \ i,j,k \tag{1k}$$

$$A_N, \ A_{\mathcal{P}}, \ \gamma_j, \ \psi_j > 0 \tag{1l}$$

4 VINEA System Overview

In this section we describe the main operations performed by VINEA to embed a VN, $i.e.$, to solve Problem (1): we first bootstrap an overlay of InP processes that are responsible for allocating resources over the underlying physical network. Such processes participate in a (centralized or) distributed, consensus-based, VN mapping protocol, and then run a final phase of resource reservation (allocation).

InP Overlay Support. Each VINEA node can be instantiated as SP or InP. Each InP process is authenticated into an overlay with a private addressing scheme to later host VN requests. These InP processes will manage the physical network infrastructure. The VINEA overlay does not have to be overlaid over TCP/IP but

its implementation has a shim layer that adapts to any underlay addresses including IP.

In particular, a Network Management System (NMS) process *enrolls* the new VINEA nodes into the private overlay. The enrollment procedure consists of an authentication (with username and password) and a policy exchange. Examples of such policies, whose scope is limited to the InP overlay, include routing update frequency, or addresses of neighbor InP processes, including a set of SPs that may publish VN requests. Once the enrollment procedure is complete, the NMS starts monitoring the availability of each enrolled VINEA node. If the VINEA node is instantiated as an InP, it may also subscribe to at least one SP using a publish/subscribe mechanism.

Asynchronous VN Embedding Protocol. Once the InP overlay is bootstrapped, an SP encodes a VN request into a `Slice` object, and publishes it so that the set of subscriber InP processes can run our embedding protocol. Then the VN mapping service starts. The VINEA embedding mechanism is *asynchronous*, and can be instantiated to run the virtual node embedding phase and the virtual link embedding phases sequentially, as in [1], or simultaneously (in one shot) as in [2], in a centralized or distributed fashion. Each InP process independently assigns utility values on a single, or on multiple virtual resources (virtual nodes, virtual paths, or both), depending on the policies (see § 6), trying to maximize its private utility function (a policy of each InP process.) After assigning values on the virtual resources, InP processes exchange such values to run a max-consensus [37] protocol for a winner determination: assuming sequential mapping of virtual nodes followed by virtual links, the InP process i with the highest (maximum) utility U_{ij} hosts virtual node j. Virtual links are then set up on loop-free physical paths obtained using a k-shortest path algorithm [38]. InP processes also exchange their utility creation times to resolve conflicts when messages arrive out of order (Figure 1c.) Appendix A contains the conflict resolution rules for the asynchronous agreement phase. [7]

Policies and Allocation. Our object model allows policy programmability for both virtual node and link embedding policies. Together with the VN constraints, an SP publishes a set of embedding policies, that are piggybacked by InP processes with the first *utility* message. SPs also have their own policies, $e.g.$, they may partition the VN request before releasing it, to distribute the load across the InPs, or to obtain different embedding solutions from competing InPs as in [3], [8]. When the SP receives a positive embedding response from one InP, the allocator service interface is used to start the allocation phase. VINEA also supports multiple simultaneous embeddings, by assigning to each VN request an identifier, unique within the scope of the physical network overlay.

5 Architecture and Object Model

The core idea behind VINEA is to support a VIrtual Network Object-based (VINO) embedding protocol to let physical nodes independently assign utility values on virtual nodes and links, using a private utility function, and then run a max-consensus protocol for a (centralized or distributed) assignment of the virtual resources. In this section, we first describe the main VINEA architecture components and the object model, and later we give examples of embedding policies that can possibly be instantiated using our API.

5.1 Architecture Components and Object Model

The VINEA's architecture is designed to support the three mechanisms of the VN embedding problem via an object model. The object model is composed by (i) a set of VN Objects, (ii) an interface to access such object's attributes locally, and (iii) a set of VINO protocol messages, to remotely operate on the states (set of attributes) necessary to embed a VN. Such states are stored into a data structure called Slice Information Base (SIB.) Similarly to the Management Information Base [39], or to a Routing Information Base (RIB) defined in every router, our SIB is a partially replicated distributed object database that contains the union of all managed objects within a Slice (VN) to embed. Via a broker (or SIB Daemon), VINEA processes handle the inter-process communication generated by publish and subscribe requests from both SP and InP processes participating in a VN embedding (see § 7 for some API usage examples).

Each VINEA process can be instantiated as an SP (generating and partitioning VN requests), or as an InP, to handle the three VN embedding mechanisms: (i) resource discovery, (ii) virtual network mapping, $i.e.$, a utility generation component, as well as an algorithm to reach (distributed) consensus on the VN mapping, and (iii) allocation, handling the final binding between virtual and physical resources. Every VINEA process also has an interface to a Network Management System component for monitoring, $e.g.$, to ping the neighbor physical nodes, and control operations, $e.g.$ to terminate an existing VN, and for authenticating VINEA processes into the private physical network overlay (Figure 2a).

5.2 The VINO Embedding Protocol

Each physical node i, where $i \in V_G$, uses its private utility function $\mathbf{U}_i \in \mathbb{R}_+^{|V_H|}$ to assign a value on a set of virtual nodes, knowing that it could be the winner of a subset of the VN request, and stores such values in a vector $\mathbf{b}_i \in \mathbb{R}_+^{|V_H|}$. Each entry $b_{ij} \in \mathbf{b}_i$ is a non-negative number representing the highest value of utility known so far on virtual node $j \in V_H$. [8] Also, physical nodes store the identifiers of the virtual nodes on which they attempt their hosting in a list (bundle vector) $\mathbf{m}_i \in V_H^{T_i}$, where T_i is a policy — the target number of virtual nodes mappable on i. Each physical node exchanges then the utility values with its neighbors, updates an assignment vector $\mathbf{a}_i \in V_G^{|V_H|}$ with the latest information on the current assignment of all virtual nodes, and applies a max-consensus algorithm [37] to determine the hosting physical nodes.

The winner hosting nodes communicate the mapping to the SP which, if possible, releases the next VN(s) or the next slice (or VN) partition, if any (see Figure 2b for the structure of the main VINO protocol message.) The way the VN is partitioned is a policy of our embedding mechanism. The mechanism iterates over multiple bidding and consensus phases simultanously and asynchronously (the bidding phase of a virtual resource can start without having

7. Even though VINEA relies on the underlying transport protocol for reliable communications, the max-consensus distributed auction is resilient to non-Byzantine failures [37]. Physical node or link failures are equivalent to bids that were never made, hence they do not affect the protocol convergence to the highest bid. The design and implementation of alternative more robust protocols that would also guarantee distributed consensus of a VN embedding in the presence of Byzantine failures are left for future research.

8. Note that a sequence of virtual nodes represent a *virtual path*, and so the protocol permits utility assignments also on virtual nodes and links simultaneously, not merely on virtual nodes.

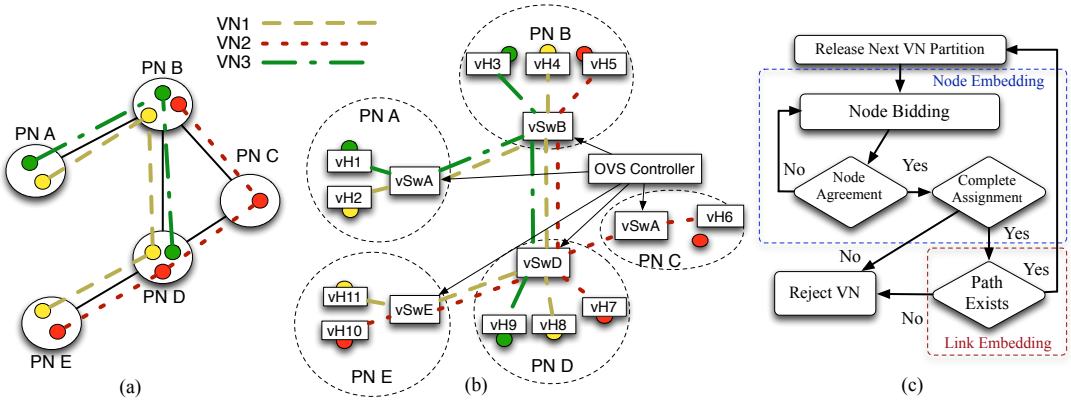


Fig. 1. VINEA System Overview: (a) Each dashed line type represents an embedded VN. Resulting VINEA configuration: for each physical node (PN) process hosting a virtual node there is a virtual switch (vSw) and at least a virtual host (vH) attached to it. We also attach an Open Virtual Switch (OVS) Openflow controller to each virtual switch, to manage the forwarding mechanism after the VN has been embedded. (b) InP overlay with three embedded virtual networks. (c) VN Embedding workflow: physical nodes use a distributed consensus-based auction to agree on who is hosting each virtual resource: the consensus is reached on the maximum bid on each virtual node.

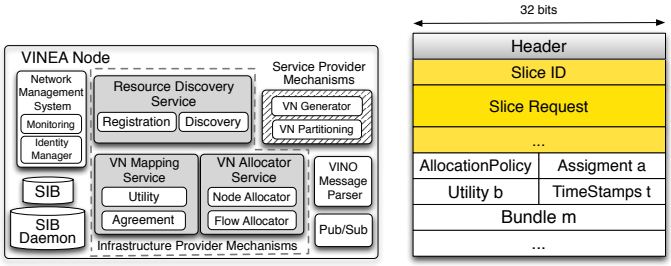


Fig. 2. (a) VINEA main architecture components: each node can be instantiated as a service provider or an infrastructure provider. (b) A VINO protocol packet: physical nodes piggyback a VN partition object (Slice request) together with the embedding policies, and information useful to attempt the (distributed) max-consensus embedding (§ 5.2).

to wait for the consensus reached on previous resources under bidding). [9]

6 VINEA Policies Programmability

VINEA's design is centered on flexibility, *i.e.*, the ability to create customizable VN embedding algorithms so as to meet desired goals by merely programming a few policies. We summarize a representative set of the policies and we show examples of how the VINEA management objects can be instantiated to satisfy novel and existing embedding approaches. In particular, we start with a set of decomposition policies in § 6.1, and then move to other, more specific set in § 6.2. Finally, we discuss how such language can be used to come up with novel (§ 7), or subsume existing (§ 7.1), VN embedding solutions.

Note that the VINO protocol still converges if InP processes use different policies, but it is not guaranteed that the Pareto optimal VN embedding will be reached.

9. In this paper our focus is on the VINEA system design, implementation, object model and policy programmability tradeoff analysis capabilities. The details of the asynchronous VN embedding algorithm that VINEA supports are in [29], while its algorithmic guarantees, both on performance and convergence, are described in [5], where an earlier synchronous version of the algorithm is described. To achieve distributed asynchronous state consistency, the deconfliction rules were about tripled.

6.1 Decomposition Policies

Due to the rich structure of problem (1), many different decompositions are possible. By supporting decomposition policy programmability, the VINEA object model can be used as a design and software engineering tradeoff mechanism to evaluate different virtual network embedding solutions. Each alternative decomposition leads to a different virtual network embedding distributed algorithm, with potentially different desirable properties. The choice of the adequate decomposition method and distributed algorithm for a particular problem depends on the infrastructure providers' goals, and on the offered service or application. The idea of decomposing problem (1) is to convert it into equivalent formulations, where a master problem interacts with a set of subproblems. Decomposition techniques can be classified into *primal* and *dual*. Primal decompositions are based on decomposing the original primal problem (1), while dual decomposition methods are based on decomposing its dual. In a primal decomposition, the master problem allocates the existing resources by directly assigning to each subproblem the amount of resources that it can use. Dual decomposition methods instead correspond to a resource allocation via pricing, *i.e.*, the master problem sets the resource price for all the subproblems, that independently decide if they should host the virtual resources or not, based on such prices.

Primal decompositions are applicable to problem (1) by an iterative *partitioning* of the decision variables into multiple subsets. Each partition set is optimized separately, while the remaining variables are fixed. For example, we could first optimize the set of virtual node variables n_{ij}^V in a node embedding phase, fixing the virtual link variables l_{kj}, and then optimize the virtual links in a path embedding phase, given the optimal value of the variables n_{ij}^V, obtained from the node embedding phase, as done in [1], [5]. Alternatively, a distributed VN embedding algorithm could simultaneously optimize both virtual node and virtual link embedding for subsequent VN partitions, *e.g.*, sorting first the partitions by the highest requested virtual node and virtual link capacity, as in [30]. Primal decompositions can also be applied with respect to the three VN embedding mechanisms. For example, by fixing the allocation variables, the embedding problem can be solved by optimizing the discovery and VN mapping first as in [4], or by optimizing the discovery variables n_{ij}^P and p_{kj} first, and

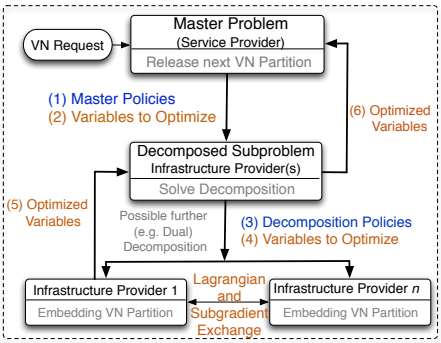


Fig. 3. Different instantiations of virtual network embedding decomposition policies result in different embedding solutions. The service provider instantiates a problem formulation according to its policies (1), and picks an objective function U (2). The infrastructure provider processes solve the decomposed subproblems, possibly further decomposing them (3-4). Finally, the optimal embedding variables are returned to the service provider (5-6), that eventually releases the next VN.

then simultaneously the mapping and allocation variables later as in [36].

Dual decomposition approaches are based on decomposing the Lagrangian function formed by augmenting the master problem with the relaxed constraints. Also in this case, it is possible to obtain different decompositions by relaxing different sets of constraints, hence obtaining different distributed VN embedding algorithms. For example, by relaxing constraints (1i) and (1j), we can model solutions that separate the VN mapping and allocation phases, such as [35], [40]. Regardless of the number of constraints that are relaxed, dual decompositions are different than primal in the amount of required parallel computation (all the subproblems could be solved in parallel), and the amount of message passing between one phase and the other of the iterative method. The dual master problem communicates to each subproblem the shadow prices, $i.e.$, the Lagrangian multipliers, then each of the subproblems (sequentially or in parallel) is solved, and the optimal value is returned, together with the subgradients. It is also possible to devise VN embedding solutions in which both primal and dual decompositions are used.

In general, a service provider can instantiate a set of policies at the *master problem*, after receiving a VN embedding request, dictating the order in which the variables need to be optimized and on which VN partition. The subproblems resulting from the decomposition can also instantiate other sets of decomposition policies, to decide which variables are to be optimized next, in which order, or even further decomposing the subproblems (Figure 3).

6.2 Other VINEA Embedding Policies

Aside from the decomposition policies, infrastructure providers may use our object model to instantiate other policies supported by the VINEA architecture. In this section we overview the other policies, supported by our system implementation.

A straightforward example of policy is (i) *the (normalized) utility function* $\mathbf{U}$ that provider processes use to attempt a virtual resource mapping. We have seen from related work, $e.g.$, [6] [3], how embedding protocols may require SPs to split the VN request. [10] VINEA can express this requirement by instantiating (ii) *the VN partitioning policy*, enforcing a limit on the length of the utility vector $\mathbf{b}_i$, so that physical nodes may host only a given partition of the VN request.

Each InP process (for either an inter- or an intra-provider embedding) can also instantiate a (iii) *load profiling policy*, to enforce a load target on its physical nodes (and links) by configuring a limit on its target allocatable capacity. In our evaluation section we show how, by merely instantiating different thresholds on different physical hosting nodes, an InP can impose a target profile load vector $\mathbb{T}$, for the purpose of balancing the load on the physical resources.

Another policy is the (iv) *privacy mode* of the vector $\mathbf{a}_i$, that is, a vector that keeps track of the identities of the current hosts of the VN during an embedding process; $\mathbf{a}_i$ may assume two forms: *least* and *most informative*. Setting the privacy policy to its least informative form requires the SP to orchestrate the virtual link embedding, since SP would be the only entity with knowledge of the InP nodes that won the virtual nodes, similar to the embedding approaches envisioned in [3], [6], while a privacy policy set to its most informative form enables a fully distributed embedding as in [5], [30]. Some of the VINEA policies are also extracted from the link embedding mechanism. We observe that all virtual link embedding schemes have two commonalities: (1) the physical path information known at each physical node, and (2) the algorithm for determining the best physical paths that host a virtual link. Two examples of such policy instantiations are the number of available paths for any source-destination, $e.g.$, k in a k-shortest path algorithm [38] for virtual path splitting, as shown in [1], and the selection of physical path(s) over which a virtual link is mapped, $e.g.$, the shortest, the least loaded, or the least used over the past n VN requests.

7 VINEA Policy Programmability

The usability of a (VN embedding) protocol should be driven by its efficiency (ability to embed as many VN requests as possible), as well as by its flexibility, namely, its applicability to a wide range of scenarios accommodating both VN users and providers' goals. To prevent ossification and to foster *agile* deployment [41], a protocol should also adapt to evolving requirements, with minimal implementation efforts. To demonstrate VINEA's flexibility, in this subsection we show examples of a few object instantiations (policies) that either adapt to different requirements, or subsume existing VN embedding solutions.[11]

In VINEA, physical nodes asynchronously exchange with their logical neighbors their utility values on virtual nodes, and optionally piggyback the VN (slice) partition to be embedded. These information are collected in a VINO object, which using the Google Protocol Buffers [42] abstract syntax notation, is represented as follows:

```
message VINO {
  required int32 version
  required int32 sliceID
  optional Slice sliceRequest
  optional char allocationPolicy
  repeated assignment a
  repeated utility b
  repeated int32 m
  repeated ForgingTime timeStamp
}
```

10. The VN partitioning problem has been shown to be NP-hard, $e.g$, in [3] and it is outside the scope of this paper.

11. The current VINEA prototype only supports static policy configurations. An interesting open question is to use learning algorithms to adapt (re-embed with a different policy configuration) the virtual network based on network measurements, $e.g.$, current CPU usage or load on a given virtual path.

The first three attributes of the VINO object are used to identify the version of the VINO protocol, the VN (or slice) identifier, and its requested constraints, such as requested CPU capacity or geolocation. The definition of a `Slice` object contains all the VN identifiers and constraints —it is straightforward and we omit it for lack of space. Note that a `Slice` object can also accommodate physical network location constraints. This means that VINEA can be used to embed partial VNs as well where the rest of the VN has already been embedded. The interested reader can find a full version of the object model and a definition of all objects in our technical report [29]. The `allocationPolicy` is used to invoke preset embedding behaviors on physical nodes, and it can be used to extend the protocol in support of customized VN embedding policy configurations (see below for examples). The last four VINO object attributes, repeated for each virtual node whose embedding needs to be found, are used by the agreement mechanism of the mapping protocol. The `assignment` attribute contains information on the current assignment of virtual nodes to physical nodes (see below.) The `utility` attribute (whose syntax is defined later in this section) contains a vector of numeric values $\mathbf{b}_i$ that specifies the utility that physical node i has on the virtual nodes. The bundle vector $\mathbf{m}_i$ contains the identities of the virtual nodes currently hosted by physical node i. The `timeStamp` contains the generation times of the latest known utility values, and is used together with the utility values to asynchronously resolve conflicts in the max-consensus agreement phase.

SAD policy configuration. Let us consider an inter-provider embedding scenario in which (1) InPs wish to reveal the least possible information to other InPs, (2) they are interested in the quickest possible response time for a VN request, and (3) they are interested in balancing the load on their physical nodes. To accommodate these goals, we set in the configuration file the privacy policy to its least informative form, `Privacy = least`, the VN partition policy to 2, `partitionSize = 2` so that a VN is rejected as soon as the smallest VN partition (one of the two virtual nodes or their adjacent virtual link) is not allocatable, and the load profiling policy to one, `bVectorLength = 1`, so that the embedding decision is on a single item (virtual node) per released VN partition. Finally, we set the utility policy to be the physical node residual capacity `nodeUtility = res_cap`.[12] As we are forcing physical nodes to embed a *single* virtual node per round, we refer in the rest of the paper to this policy configuration as Single Allocation for Distributed embedding (SAD).

Remark. In this policy configuration, each virtual node will be embedded on the physical node with the highest residual capacity, hence greedily balancing the load on the physical network (overlay).

Setting the privacy policy to the least informative form practically means that InPs use only the `assigned` optional attribute of the `assignment` VINO object, without revealing the InP identity, $i.e.$ leaving (the identity of the hosting physical node) `hostingPnodeName` attribute unused. Setting the VN partitioning policy practically means that as many utility objects will be exchanged to embed virtual nodes as specified by the partitioning policy. For example, in the case of the SAD configuration, the SP would piggyback together with its VN request a `partitioningPolicy` attribute value set to 2, so that each physical node sets (and sends) at most two utility objects, one for each virtual node:

```
message utility {
   required int32 vNodeId
   optional double utilityValue
}
```

Together with the `partitoningPolicy` attribute, the SP also piggybacks with its VN request a `loadProfiling` object, whose attributes include the identity of each physical node, and an integer value representing the limit on the number of admissible hosting virtual nodes and outgoing virtual links. [13] The `assignment` object is hence:

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

MAD policy configuration. Let us now consider an (intra or inter-provider) scenario in which: (1) embedding VNs with the smallest embedding time is more desirable than hiding information from other physical nodes, and (2) infrastructure (or service) providers prefer collocating, $i.e.$, packing virtual nodes on physical nodes to save energy (or renting costs as in a collocation game [44].) To this end, an SP does not partition the VN request so that each physical node has an offline knowledge of the entire VN request (as opposed to the SAD policy configuration, which requires releasing the VN partitions in an online fashion.) Moreover, SP sets the privacy policy to its most informative form, so that the max-consensus is run simultaneously on both the utility vector and on the assignment vector, that stores the identities of the physical nodes currently winning the virtual nodes. As we are enabling physical nodes to host *multiple* virtual nodes per embedding round [14], we refer in the rest of the paper to this policy configuration as *Multiple* Allocation for Distributed embedding (MAD).

PAD policy configuration. In the previous two policy configurations, the embedding is solely on virtual nodes. Virtual links are later embedded (using a k-shortest path [38] algorithm) once InPs agree on the identities of the hosting physical nodes. This approach has the advantage of a faster embedding response when an embedding is not possible, but the resulting embedding may have adjacent virtual nodes hosted on non-adjacent physical nodes. This may be a problem as in a distributed VN embedding scenario, similar to the case of Border Gateway Protocol (BGP), intermediate providers may or may not agree to relay traffic flowing on virtual links. The VINEA policies can be instantiated to require InPs to attempt embeddings only on virtual nodes such that they do not act as relays for virtual links for which they are not hosting any of its ends. In the rest of the paper we refer to this policy configuration as *Path* Allocation for Distributed embedding (PAD).

Remark. Under PAD InPs embed simultaneously nodes and links. The policies are set as in the MAD configuration, plus one additional constraint: a physical node i merely sets its utility for

12. All these object attributes are intuitive and their detailed definition is in our technical report [29].

13. Second price auctions on a *single* good are known to have the strong property of being truthful in dominant strategies [43], $i.e.$, the auction maximizes the revenue of the bidders who do not have incentives to lie about their true (utility) valuation of each virtual node. In our settings however, truthful strategies may not work as there is uncertainty on whether more VNs, or even more virtual nodes in the same VN, are to be assigned in the future; bidders may have incentives to preserve resources for stronger future bids. We leave the investigation of pricing mechanisms for a VN embedding protocol as an interesting open question.

14. A round is an asynchronous bidding and agreement phase on at least one virtual resource.

virtual node j, $b_{ij} = 0$ if the current winner of virtual nodes adjacent to j is not i itself, or it is not a physical node adjacent to i, namely, at most one physical hop away.

7.1 VINEA Policies Can Subsume Existing Solutions

In this subsection we show with two representative examples how VINEA can subsume existing VN embedding approaches. To our knowledge, the first distributed VN embedding algorithm was presented in [30]. We call it *hub-and-spoke*, as the solution attempts to iteratively embed the residual hub-and-spoke virtual topologies of a VN: first the SP releases a VN partition composed of the hub virtual node (the virtual node with the highest requested CPU capacity) and all its first-hop neighbors —spokes— together with the adjacent virtual links; if an embedding for the first hub-and-spoke VN partition is found, the SP releases the next hub-and-spoke VN partition. The hub virtual node is mapped onto the physical node whose residual capacity is largest. VINEA can subsume the VN embedding algorithm described in [30] by instantiating the partitioning policy so that the VN requests are split into multiple hub-and-spoke virtual topologies, and by requiring all physical nodes to bid using their CPU residual capacity as utility.

To our knowledge, PolyVINE [4] is the first distributed VN embedding approach that lets InPs specify their embedding policies (our utility $\mathbf{b}_i$ vector.) The algorithm requires an SP to send the entire VN request to at least one InP, that embeds what it can, given its available capacity, and then forwards the residual VN partition to some neighbor InPs. We can subsume PolyVINE by letting each InP i use its own utility $\mathbf{b}_i$, and by setting the VN partitioning policy according to the residual VN partition.

In both the PolyVINE and the hub-and-spoke policy configurations, the VINEA privacy policy is set to its most informative form (the identities of the physical nodes are known), and the load profiling policy does not impose any bound, $i.e.$, each physical node is free to use all its physical available capacity to host the requested VNs.

7.2 API Usage Example

The VINEA API, together with the configuration file settings, allows VN embedding policy programmability. The expressiveness of the policy language is a result of different object attributes, instantiated via our object model. Let us assume that an SP wants to receive updates every 3 seconds on the embedding request of a VN, identified by a `sliceID`. First, the SP creates a subscription event object, and then uses the SIB Daemon API to create and send the publish request to all its subscribers. The `writePub` call is then converted into a sequence of VINO messages that *write* into the SIB of the subscriber processes:

```
SubEvent = new SubEvent(EventType.PUB_VN, 3, sliceID);
int pubID = SIBDaemon.createEvent(VNRequest, SubEvent);
SIBDaemon.writePub(pubID, sliceID);
```

Similarly, if an InP wants to be updated on the VN embedding requests generated by one of the available SPs, it subscribes to the slice objects published from such SP as follows:

```
SubEvent = new SubEvent(EventType.SUB, null, VN);
int subID = SIBDaemon.createEvent(SP_Address, SubEvent);
SIBDaemon.writePub(subID, sliceID);
```

InP processes also subscribe to the utility object attributes of other InP processes participating to the embedding. After these subscriptions, an enrollment phase begins, overlooked by a slice manager, and the InP processes join the overlay of nodes participating to the current embedding. Other examples of API usage and applications that the same underlying publish/subscribe engine supports are illustrated in our technical reports [11], [29], [45]. In the next subsection we show analytically with a case study how by instantiating VINEA with different policies we may obtain VN embedding algorithms with different behavior, and that may suit different InP and SP goals.

7.3 Primal vs Dual Decomposition Policy

In this section we analyze the tradeoffs between primal and dual decompositions, for a simple VN embedding subproblem. We later use this case study to show the results of a tradeoff analysis between optimality and speed of convergence of the iterative method used by a CPLEX solver. This section also clarifies the connection between the VINEA object model interface and the analytic interpretation of such interfaces (Lagrangian multipliers) used to obtain different distributed virtual network embedding algorithms.

We consider a subproblem of problem (1): the virtual node embedding problem, where the VN request is split in two partitions. The problem can be formulated as follows:[15]

$$\max_{\mathrm{u,v}} \quad c^T u + \tilde{c}^T v$$

$$\text{subject to} \quad Au \leq b \tag{2a}$$

$$\tilde{A}v \leq \tilde{b} \tag{2b}$$

$$Fu + \tilde{F}v \leq h \tag{2c}$$

where u and v are the sets of decision variables referring to the first and to the second VN partition, respectively; F and $\tilde{F}$ are the matrices of capacity values for the virtual nodes in the two partitions, and h is the vector of all physical node capacity limits. The constraints (2a) and (2b) capture the separable nature of the problem into the two partitions. Constraint (2c) captures the complicating constraint.

Embedding by Primal Decomposition Policy. By applying primal decomposition to problem (2), we can separately solve two subproblems, one for each VN partition, by introducing an auxiliary variable z, that represents the percentage of physical capacity allocated to each subproblem. The original problem (2) is equivalent to the following master problem:

$$\max_{\mathrm{z}} \ \phi(z) + \tilde{\phi}(z) \tag{3}$$

where:

$$\phi(\mathrm{z}) = \quad \sup_u c^T u \tag{4a}$$

$$\text{subject to} \quad Au \leq b \tag{4b}$$

$$Fu \leq z \tag{4c}$$

and

$$\tilde{\phi}(\mathrm{z}) = \quad \sup_v \tilde{c}^T v \tag{5a}$$

$$\text{subject to} \quad \tilde{A}v \leq \tilde{b} \tag{5b}$$

$$\tilde{F}v \leq h - z. \tag{5c}$$

The primal master problem (3) maximizes the sum of the optimal values of the two subproblems, over the auxiliary variable z. After z is fixed, the subproblems (4) and (5) are solved separately, sequentially or in parallel, depending on the cloud provider's policy. The master algorithm updates z, and collects the two

15. In Section 3 we model the VN embedding problem as a whole. Here instead we use the matrix notation to highlight the complicating variables and complicating constraints. Our aim is to communicate that the problem modeled in Section 3 can be remodeled using different groups of variables which may lead to different distributed algorithms (as shown in Figure 3).

Algorithm 1: Distributed Embedding by Primal Decomp. Policy

1: Given z_t at iteration t, solve subproblems to obtain ϕ and $\tilde{\phi}$ for each VN partition, and dual variables $\lambda^\star(z_t)$ and $\tilde{\lambda}^\star(z_t)$
2: Send/Receive ϕ, $\tilde{\phi}$, $\lambda^\star$ and $\tilde{\lambda}^\star$
3: Master computes subgradient $g(z_t) = -\lambda^\star(z_t) + \tilde{\lambda}^\star(z_t)$
4: Master updates resource vector $z_{t+1} = z_t - \alpha_t g$

Algorithm 2: Distributed Embedding by Dual Decomp. Policy

1: Given λ_t at iteration t, solve the subproblems to obtain the optimal values $u^\star$ and $v^\star$ for each VN partition
2: Send/Receive optimal node embedding $u^\star$ and $v^\star$
3: Master computes the subgradient $g = Fu^\star + \tilde{F}v^\star - h$
4: Master updates the prices $\lambda_{t+1} = (\lambda_t - \alpha_t g)_+$

subgradients, independently computed by the two subproblems. To find the optimal z, we use a subgradient method. In particular, to evaluate a subgradient of $\phi(z)$ and $\tilde{\phi}(z)$, we first find the optimal dual variables $\lambda^\star$ for the first subproblem subject to the constraint $Fu \leq z$. Simultaneously (or sequentially), we find the optimal dual variables $\tilde{\lambda}^\star$ for the second subproblem, subject to the constraint $\tilde{F}v \leq h - z$. The subgradient of the original master problem is therefore $g = -\lambda^\star(z) + \tilde{\lambda}^\star(z)$; that is, $g \in \partial(\phi(z) + \tilde{\phi}(z))$.[16] The primal decomposition algorithm, combined with the subgradient method for the master problem is repeated, using a diminishing step size, until a stopping criteria is reached (Algorithm 1).

The optimal Lagrangian multiplier associated with the capacity of physical node i, $-\lambda_i^\star$, tells us how much worse the objective of the first subproblem would be, for a small (marginal) decrease in the capacity of physical node i. $\tilde{\lambda}_i^\star$ tells us how much better the objective of the second subproblem would be, for a small (marginal) increase in the capacity of physical node i. Therefore, the primal subgradient $g(z) = -\lambda(z) + \tilde{\lambda}(z)$ tells us how much better the total objective would be if we transfer some physical capacity of physical node i from one subsystem to the other. At each step of the subgradient method, more capacity of each physical node is allocated to the subproblem with the larger Lagrange multiplier. This is done with an update of the auxiliary variable z. The resource update $z_{t+1} = z_t - \alpha_t g$ can be interpreted as shifts of some of the capacity to the subsystem that can better use it for the global utility maximization.

Embedding by Dual Decomposition Policy. An alternative method to solve problem (2) is to use dual decomposition, relaxing the coupling capacity constraint (2c.) From problem (2) we form the partial Lagrangian function:

$$L(u, v, \lambda) = c^T u + \tilde{c}^T v + \lambda^T (Fu + \tilde{F}v - h) \tag{6a}$$

Hence, the dual function is:

$$q(\lambda) = \inf_{u,v} \{L(u, v, \lambda) | Au \leq b, \tilde{A}v \leq \tilde{b}\} \tag{7a}$$

$$= -\lambda^T h + \inf_{Au \leq b} (F^T\lambda + c)^T u + \inf_{\tilde{A}v \leq \tilde{b}} (\tilde{F}^T\lambda + \tilde{c})^T v,$$

and the dual problem is:

$$\max_{\lambda} \quad q(\lambda) \tag{8a}$$
$$\text{subject to} \quad \lambda \geq 0,$$

We solve problem (8) using the projected subgradient method [46]. To find a subgradient of q at λ, we let $u^\star$ and $v^\star$ be the optimal solutions of the subproblems:

$$u^\star = \max_{u} \quad (F^T\lambda + c)^T u \tag{9a}$$
$$\text{subject to} \quad Au \leq b \tag{9b}$$

and

$$v^\star = \max_{v} \quad (\tilde{F}^T\lambda + \tilde{c})^T v \tag{10a}$$
$$\text{subject to} \quad \tilde{A}v \leq \tilde{b} \tag{10b}$$

respectively. Then, the infrastructure provider processes in charge of solving the subproblems send their optimal values to the master problem, so that the subgradient of the dual function can be computed as:

$$g = Fu^\star + \tilde{F}v^\star - h. \tag{11}$$

The subgradient method is run until a termination condition is satisfied (Algorithm 2); the operator $(\cdot)_+$ denotes the non-negative part of a vector, $i.e.$, the projection onto the non-negative orthant. At each step, the master problem sets the prices for the virtual nodes to embed. The subgradient g in this case represents the margin of the original shared coupling constraint. If the subgradient associated with the capacity of physical node i is positive ($g_i > 0$), then it is possible for the two subsystems to use more physical capacity of physical node i. The master algorithm adjusts the price vector so that the price of each overused physical node is increased, and the price of each underutilized physical node is decreased, but never negative.

8 Prototype Implementation and Testbed

To establish the practicality of our architecture, we tested it on a system implementation. The prototype enables users to write real applications on top of the embedded VNs.

Each process joins a private overlay before running the VN embedding protocol. A VN request is released by a VINEA node instantiated as SP. Then InP processes run a physical resource discovery protocol, the asynchronous virtual network mapping protocol, and finally, the virtual network is allocated by reserving CPU and bandwidth. Our prototype is implemented in a single-host Linux-based testbed (§ 8.1), 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 for 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 basic applications such as `ping`, `iperf` or more complex SDN scenarios using OpenFlow [21].

Our VINEA prototype (whose software architecture is shown in Figure 2a) resulted in about $50K$ lines of Java code, without considering test classes, Python scripts and Linux C code that VINEA leverages for the final allocation phase of embedded VNs.

Logically, the prototype is divided into nine main architecture components: a Network Management System, the three embedding services of an infrastructure provider —resource discovery, virtual network mapping and allocation, a set of service provider functionalities, a database containing all the objects, a VINO message handler, a message parser to serialize and deserialize

16. For the proof please refer to §5.6 of [46].

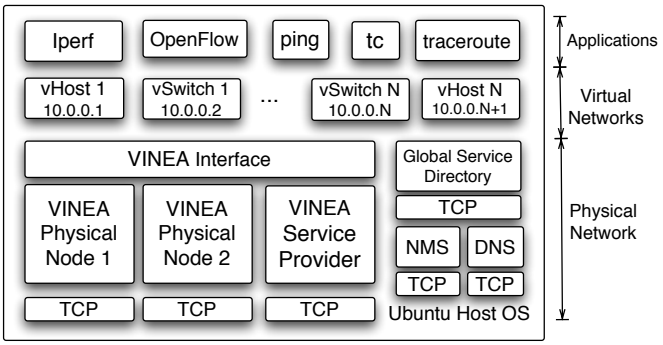


Fig. 4. VINEA testbed architecture: physical wires are emulated with loopback TCP connections on well-known ports. After the virtual networks are embedded, we can run Linux applications between virtual nodes, *e.g.*, `ping`, `traceroute`, or we can send data traffic, and measure the reserved bandwidth performance with `iperf`.

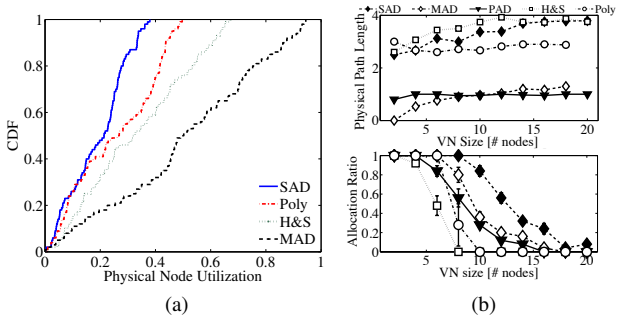


Fig. 5. **Simulation results (a-b)**: Comparison between two related distributed VN embedding solutions and two VINEA policies: (a) The CDF shows how VINEA policies may be instantiated to balance the load on physical nodes, or to collocate multiple virtual nodes allowing more physical nodes to remain idle. (*b*)-*top*. The physical loop-free path lengths hosting at least a virtual link show how, as designed, the PAD policy enforces the embedding of virtual links on single-hop physical paths. (*b*)-*bottom*. Different policy instantiations result in different VN allocation ratios (and so different provider's revenue).

objects with the Google Protocol Buffer libraries [42], and a publish/subscribe system to release/acquire VN (partition) objects, and to subscribe to other InP processes for VN embedding updates. Our technical report [29] contains the implementation details of each of these components, while the code is available at [12].

8.1 Linux Testbed

In order to evaluate our prototype, we implemented a testbed whose architecture is shown in Figure 4. Our base system is a host running an Ubuntu distribution of Linux (version 12.04.) Each InP process includes the VINEA modules. Each 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 [13] 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 with `iperf` between virtual hosts.

8.2 Testbed Resiliency and Traffic Shaping

By running the asynchronous VINO protocol, VINEA applies a set of local deconfliction rules that do not require access to a global embedding state, consistently handle out-of-order messages and ignore redundant information. This means that a set of VINEA processes is guaranteed to converge to the Pareto optimal embedding, while being resilient to all failures that do not result into a persistent physical network partition over the embedding lifetime, *i.e.*, losses, delays, or disconnections. Our VINEA testbed supports queuing delay or physical network congestions with simple Linux commands such as `tc` and `netem`.

9 VINEA Evaluation

In this section we focus on evaluating, with simulations and via our prototype, the impact and the tradeoffs that different VINEA policies have on the physical network load, on the VN allocation ratio (ratio between VN allocated and requested), on the VINO protocol overhead, and on the length of the hosting physical paths.

We were able to replicate our results across several physical network sizes and (Barabasi-Albert and Waxman) connectivity models. We only show a representative set of results, obtained with 95% confidence intervals. We leave as an interesting open question a sensitivity analysis of how the structure of the physical or virtual network network topology affects the convergence speed, or even the efficiency of the embedding, generalizing our results beyond random graphs generated with synthetic methods and/or parameters.

Results Summary. We implemented an event-driven simulator to test VINEA's scalability, and a local testbed (that can be run on a single host) to establish its practicality. Our results on the physical node utilization show how different VINEA policies may lead to a physical network with balanced load, or with multiple virtual nodes collocated on a small set of physical nodes (Figure 5*a*.) The latter virtual node packing effect allows InPs to keep idle a higher number of physical nodes, therefore reducing the InP energy costs. Independently from the physical network size and connectivity model, embedding policies that partition the VN request, *e.g.*, SAD or PolyVINE, distribute better the load across the physical network compared to other *node packing* policies, *e.g.*, MAD and hub-and-spoke, but not always result in higher VN allocation ratios (Figure 5*b*-bottom).

In our prototype evaluation we focus on confirming how, in an emulated environment, different embedding policies lead to different VN allocation ratios across representative virtual network topologies: linear, star, tree, and fully connected (Figure 7.) We also dissect the architecture components responsible for the embedding protocol overhead, and compare two representative embedding policy configurations (Figure 8.) Finally, our results show how, in contrast with other embedding approaches [2], [34], *policies that require coordinated node and link embedding (e.g., PAD) may lower the embedding performance* (Figure 5*b*-top).

Simulation Scenario. Our goal is to compare different VINEA policies with previous distributed VN embedding solutions and among each other. We also choose a scenario that attempts to reproduce real VN embedding requests on popular physical network connectivity models. To this end, we compute the physical node utilization after embedding 100 VNs with three different VINEA policy configurations —MAD, SAD and PAD— as well as with

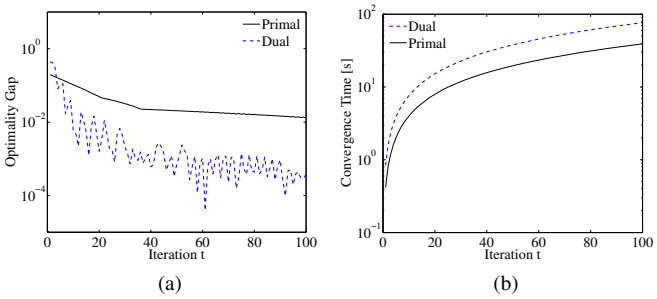
our implementation of the PolyViNE [4] and Hub&Spoke [30] embedding heuristics. The sizes of our VNs were sampled from a distribution generated by a trace dataset of 8 years of VN requests to the Emulab testbed [47], where each VN has on average about 50 virtual nodes. The virtual links (not in the dataset) are generated at random with 0.5 probability of having an edge between any two virtual nodes. The physical network, synthetically generated with BRITE [48] has 500 nodes (typical size of a medium ISP) and follows one of the two popular connectivity models: Barabasi-Albert or Waxman.

Single vs. Multiple Node Embedding Policy Tradeoff. When applying the SAD configuration, all physical nodes have utilization lower than 35%, with over half of the physical nodes less than 20% utilized. When instead we instantiate VINEA with the MAD policy configuration, we obtained a higher physical node utilization: some physical nodes reached a 75% utilization (Figure 5a.) In this experiment, the available physical node and link capacities are enough to embed all the requested VNs (typical realistic scenario.) This means that a higher physical node utilization is a consequence of a higher number of idle physical nodes, since the (node and link) physical capacity necessary to embed the requests is the same across all embedding algorithms.

Expecting InPs to relay VN traffic may be undesirable. The PAD policy configuration avoids InP relays for virtual links, which leads to an average physical path length in the range $[0, 1]$ hops for embedding a virtual link (Figure $5b - top$); a loop-free physical path has length 0 when the two end virtual nodes of a virtual link are both hosted by the same physical node, and length 1 when the two end virtual nodes of a virtual link are hosted by two neighboring physical nodes. Although PAD avoids relays, and simultaneously embed both nodes and links, our results show how *the physical link capacity provided by relay physical nodes helps improve the VN allocation ratio (Figure 5b-bottom)*. From the same experiment, we note how, in this setting, our load balancing policy configuration (SAD) resulted in a higher number of allocated VN requests, outperforming the other approaches, including our implementation of PolyVINE and Hub&Spoke.

Decomposition Policy Tradeoff: We evaluate a few representative decomposition policies of our architecture. Our simulations use a CPLEX solver [49] to analyze the tradeoff between optimality and the speed of convergence of the primal and dual decompositions solved by the iterative methods described in Algorithms 1 and 2. We embed a typical VN request of 50 virtual nodes onto a physical network overlay of 10 physical (hosting) nodes. Since we can always embed a VN leaving no residual capacity on the hosting nodes, the *Slater's condition* [46] is satisfied for Problems (3) and (8.) This means that there is no duality gap, but it is not desirable to wait for the optimal node embedding when the improvements relative to the previous iterations are small. Hence, using a diminishing step size rule $\alpha = 0.5/t$, where t is the iteration step, we stopped our simulations after $t = 100$ (Figures 6a and b.) We note that the solutions found using a dual decomposition policy reduces its duality gap more quickly, at the expense of a longer convergence time.

Testbed Emulation Setup. In all our prototype evaluation experiments, an Ubuntu image was hosted on a VirtualBox instance over a 2.5 GHz Intel Core $i5$ processor, with $4GB$ of DDR3 memory. We start our InP overlay, and we launch one or multiple VN requests with different size and topologies. We tested the embedding of VNs with up to 16 virtual nodes, with linear, star (hub-and-spoke), tree and full virtual network topologies



(a) (b)

Fig. 6. **Decomposition Policy Tradeoff.** Using a diminishing step size rule $\alpha_t = 0.5/t$ to complete the first 100 iterations, a node embedding solved by dual decomposition leads to a smaller duality gap (a), at the expense of a longer convergence time.

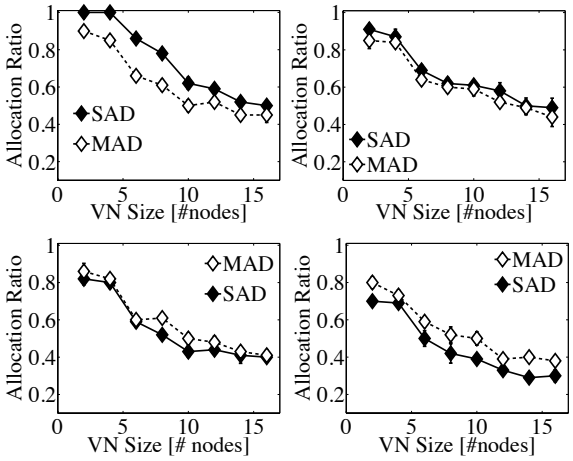


Fig. 7. **Prototype Evaluation**: Policy configuration tradeoff. Five InP processes hosting: (top-left) VNs with linear (top-right) star (bottom-left) tree and (bottom-right) full virtual topology. Policies that require partitioning perform worse as the number of virtual links increases (confidence intervals are almost always too small to be seen).

(Figure 7.) [17]

Note that we have tested the scalability of VINEA with our simulator; the limit on the number of virtual nodes in our prototype was imposed by the use of OpenFlow. In particular, we used Open vSwitch (OVS) in OpenFlow mode, that (at this time) requires an OpenFlow controller with at most 16 interfaces. Each of the controllers supported turns the OVS switches into Ethernet bridges (learning switches.) Using the command `route add`, we set up the routes for each virtual node following the requested VN connectivity.

Testbed Network Models. We vary the virtual network size from 2 till the limit of 16 nodes is reached and we tested VINEA on several InP overlay sizes, and with linear, star, tree, and fully connected physical (InP overlay) topologies, with a wide range of virtual topologies. We only show results for InP overlay of size 10. The other results are similar. We randomly assign physical link capacities between 50 and 100 Mbps (as in [1]), then we assign the InP process capacity to be the sum of its outgoing

17. Note that when the number of physical paths is limited, $e.g.$, in a linear physical topology, MAD may perform worse (Figure 7 top-left). This is because MAD can assign multiple virtual nodes in the same auction round on the same physical node, the outgoing virtual capacity might exhaust the physical link capacity hence causing future VN requests to be rejected.

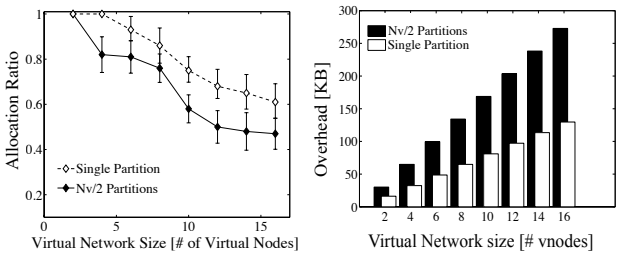


Fig. 8. **Prototype Evaluation**. Partitioning a VN results in a lower VN request allocation ratio, leading to lower cloud revenue (a), and to higher signaling overhead (b). This result is an improvement with respect to existing approaches [3], [4], [8], in which the VN embedding mechanism requires VN partitioning.

physical link capacities. We specify the capacities in the InP process configuration file. We assume the virtual link capacity to be randomly chosen between 1 and 5 Mbps. The virtual node capacity of a VN request is assigned to be the sum of its outgoing virtual links. Results are shown with 95% confidence intervals, while the overhead results refer to a single run.

Utility Model and VN Lifetime. All InP processes use the same utility function. The goal of the experiment is to embed a set of 100 virtual networks, with one second inter-arrival time between VN 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 embedding 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, $\mathbf{b}_i$ the utility function used by InP processes, and $x_{ij} = 1$ if an InP process i is hosting virtual node j and zero otherwise. VINEA releases the allocated resources after the VN lifetime has expired. We sample the lifetime of the VNs from our dataset of VN requests to Emulab.

Embedding Overhead and Load Tradeoff. The load balancing policy (SAD) which requires partitioning the VN request to a set of smaller requests made of each virtual link with its two end virtual nodes, performs worse in terms of VN allocation ratio, as the number of virtual links increases (Figure 8a.) To assess the message overhead, we measured the actual number of bytes exchanged across the InP overlay (Figure 8b). Our results show how an SP can significantly limit the network overhead by selecting a single InP process to send its requests. This result is an improvement with respect to existing approaches [3], [4], [8] in which the VN embedding mechanism requires VN partitioning.

10 Conclusions and Open Problems

In this paper we presented VINEA, an architecture for virtual network embedding policy programmability. VINEA separates policies ($i.e.$, high-level goals) from underlying mechanisms of the embedding problem: resource discovery, virtual network mapping, and allocation. VINEA's design leverages an asynchronous consensus mechanism, and enables VNs to be built using resources of a single provider, or across multiple federated providers; our object model serves as a foundation for a VN embedding protocol specification. We compared the performance of representative policy configurations with simulations and over a prototype implementation, and we illustrated their performance and overhead tradeoffs. Each VINEA node can be instantiated as a service or infrastructure provider. Our prototype can augment existing open-source "Networking as a Service" solutions such as OpenStack [20], enabling VN users to program their own embedding policies, and it provides a clean solution to the slice stitching problem for the GENI testbed [16]. We released our single-host Linux-based virtual network testbed to enable users to test their own embedding policies, and run applications within the embedded VNs.

Our work leaves a number of open research problems. From the system perspective, the VINEA embedding object model can be extended to the problem of virtual network function placement, and to the problem of steering virtual links after a network has been embedded. From the algorithmic design perspective, we believe the analysis of embedding performance with heterogenous policies, and a secure VN embedding protocol able to function even in the presence of misconfigurations, malicious users, or (Byzantine) failures are also interesting research directions.

References

[1] M. Yu, Y. Yi, J. Rexford, and M. Chiang, "Rethinking Virtual Network Embedding: Substrate Support for Path Splitting and Migration," *SIGCOMM CCR.*, vol. 38, no. 2, pp. 17–29, 2008.
[2] Chowdhury, M. et al., "ViNEYard: Virtual Network Embedding Algorithms with Coordinated Node and Link Mapping," *IEEE/ACM Trans. Netw.*, vol. 20, no. 1, pp. 206–219, Feb. 2012.
[3] Houidi, I. et al., "Virtual Network Provisioning across Multiple Substrate Networks," *Computer Networks*, vol. 55, no. 4, Mar. 2011.
[4] M. Chowdhury, F. Samuel, and R. Boutaba, "PolyViNE: Policy-Based Virtual Network Embedding Across Multiple Domains," ser. SIGCOMM VISA Workshop, 2010.
[5] F. Esposito, D. Di Paola, and I. Matta, "On distributed virtual network embedding with guarantees," *IEEE/ACM Transactions on Networking.*, Feb. 2014. [Online]. Available: http://dx.doi.org/10.1109/TNET.2011.2159308
[6] Y. Zhu et al., "Cabernet: Connectivity Architecture for Better Network Services," in *CoNEXT*, 2008, p. 64.
[7] J. Lu and J. Turner, "Efficient Mapping of Virtual Networks onto a Shared Substrate," Washington Univ. in St. Louis, Tech. Rep., 2006.
[8] F. Zaheer, J. Xiao, and R. Boutaba, "Multi-provider Service Negotiation and Contracting in Network Virtualization," in *IEEE NOMS*, 2010.
[9] J. Day, I. Matta, and K. Mattar, "Networking is IPC: A Guiding Principle to a Better Internet," in *Proc. of the CoNEXT*. ACM, 2008, pp. 67:1–67:6. [Online]. Available: http://doi.acm.org/10.1145/1544012.1544079
[10] F. Esposito, Y. Wang, I. Matta, and J. Day, "Dynamic Layer Instantiation as a Service," in *In Proc. of 10th USENIX Symp. on Networked Systems Design and Implementation (NSDI 2013)*, Lombard, IL, April 2013.
[11] Y. Wang, I. Matta, F. Esposito, and J. Day, "Introducing ProtoRINA: A prototype for programming recursive-networking policies," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 129–131, Jul. 2014.
[12] F. Esposito. (2014) The Virtual Network Embedding Architecture (VINEA) Project. http://csr.bu.edu/vinea.
[13] Open Virtual Switch Community. (2013) Open Virtual Switch. [Online]. Available: http://openvswitch.org/
[14] F. Esposito and I. Matta, "A decomposition-based architecture for distributed virtual network embedding," in *Proc. of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing*, ser. DCC '14. New York, NY, USA: ACM, 2014, pp. 53–58.
[15] B. Lantz, B. Heller, and N. McKeown, "A Network in a Laptop: Rapid Prototyping for Software-Defined Networks," in *Proc. of ACM SIGCOMM Hotnets-IX*, 2010, pp. 19:1–19:6. [Online]. Available: http://doi.acm.org/10.1145/1868447.1868466
[16] GENI. http://www.geni.net.
[17] F. Esposito, I. Matta, and V. Ishakian, "Slice Embedding Solutions for Distributed Service Architectures," *ACM Computing Surveys*, vol. 46, no. 2, March 2014.
[18] Neutron. https://wiki.openstack.org/wiki/Neutron.
[19] CloudStack. http://cloudstack.apache.org/.
[20] OpenStack. http://openstack.org/,.
[21] McKeown N. et al., "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM CCR.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
[22] Qazi, Z. A. et al, "SIMPLE-fying Middlebox Policy Enforcement Using SDN," ser. SIGCOMM '13, New York, NY, USA, 2013, pp. 27–38.
[23] Voellmy, A. et al., "Maple: Simplifying sdn programming using algorithmic policies," ser. SIGCOMM '13, 2013, pp. 87–98.

[24] Mattar K. et al., "Declarative Transport: A Customizable Transport Service for the Future Internet," in *In Proc. of NetDB 2009, co-located with SOSP 2009, Big Sky, MT*, 2009.
[25] Liu C. *et al*, "Declarative Policy-based Adaptive Mobile Ad Hoc Networking," *IEEE/ACM Trans. Netw.*, vol. 20, no. 3, Jun. 2012.
[26] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan, "Declarative Routing: Extensible Routing with Declarative Queries," ser. SIGCOMM. ACM, 2005, pp. 289–300.
[27] Schaffrath G. *et al*, "Network Virtualization Architecture: Proposal and Initial Prototype," in *ACM SIGCOM VISA workshop*. New York, NY, USA: ACM, 2009, pp. 63–72. [Online]. Available: http://doi.acm.org/10.1145/1592648.1592659
[28] Guo, C. *et.al*, "SecondNet: a Data Center Network Virtualization Architecture with Bandwidth Guarantees," in *Proc. of CoNEXT 2010*. New York, NY, USA: ACM, 2010, pp. 15:1–15:12. [Online]. Available: http://doi.acm.org/10.1145/1921168.1921188
[29] F. Esposito, "A Policy-based Architecture for Virtual Network Embedding," Ph.D. dissertation, Boston University, Computer Science Department., Sept. 2013.
[30] I. Houidi, W. Louati, and D. Zeghlache, "A Distributed Virtual Network Mapping Algorithm," in *IEEE International Conference on Communications (ICC)*, May 2008, pp. 5634 –5640.
[31] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?id=1924943.1924969
[32] Y. Xin, I. Baldine, A. Mandal, C. Heermann, J. Chase, and A. Yumerefendi, "Embedding virtual topologies in networked clouds," in *Proceedings of the 6th International Conference on Future Internet Technologies*, ser. CFI '11. New York, NY, USA: ACM, 2011, pp. 26–29. [Online]. Available: http://doi.acm.org/10.1145/2002396.2002403
[33] B. Chun and A. Vahdat, "Workload and Failure Characterization on a Large-Scale Federated Testbed," IRB-TR-03-040, Intel Research Berkeley, Tech. Rep., 2003.
[34] J. Lischka and H. Karl, "A Virtual Network Mapping Algorithm based on Subgraph Isomorphism Detection," *ACM SIGCOMM VISA Workshop*, Aug. 2009.
[35] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat, "SHARP: an Architecture for Secure Resource Peering," *SIGOPS Operating System Review*, vol. 37, no. 5, pp. 133–148, 2003.
[36] J. Albrecht, D. Oppenheimer, D. Patterson, and A. Vahdat, "Design and Implementation Trade-offs for Wide-Area Resource Discovery," *ACM Transactions on Internet Technology*, vol. 8, no. 4, pp. 1–44, 2008.
[37] N. A. Lynch, *Distributed Algorithms*, 1st ed. Morgan K., Mar. 1996.
[38] D. Eppstein, "Finding the k Shortest Paths," *SIAM Journal of Computing*, vol. 28, no. 2, pp. 652–673, 1999.
[39] K. McCloghrie and M. Rose, "Management Information Base for Network Management of TCP/IP-based internets. http://www.ietf.org/rfc/rfc1156.txt," 1990.
[40] A. AuYoung, B. Chun, A. Snoeren, and A. Vahdat, "Resource allocation in federated distributed computing infrastructures," in *Proceedings of the 1st Workshop on Operating System and Architectural Support for the Ondemand IT InfraStructure*, October 2004.
[41] Beck Kent *et al*., "Manifesto for Agile Software Development," *Agile Alliance*, June 2010.
[42] Google Protocol Buffers. (2013) Developer Guide http://code.google.com/apis/protocolbuffers.
[43] B. Lucier, R. P. Leme, and É. Tardos, "On revenue in the generalized second price auction," in *Proceedings of the 21st International Conference on World Wide Web*, ser. WWW '12. New York, NY, USA: ACM, 2012, pp. 361–370. [Online]. Available: http://doi.acm.org/10.1145/2187836.2187886
[44] A. B. Jorge Londono and S. Teng, "Collocation games and their application to distributed resource management," in *In Proceedings of USENIX HotCloud'09: Workshop on Hot Topics in Cloud Computing, San Diego, CA,*, June 2009.
[45] Y. Wang, F. Esposito, I. Matta, and J. Day, "Recursive InterNetworking Architecture (RINA) Boston University Prototype Programming Manual," Boston University, Tech. Rep. BUCS-TR-2013-013, Nov 2013.
[46] S. Boyd and L. Vandenberghe, *Convex Optimization*, online, Ed. http://www.stanford.edu/people/boyd/cvxbook.html, 2004.
[47] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," *SIGOPS Operating System Review*, 2002.
[48] A. Medina, A. Lakhina, I. Matta, and J. Byers, "BRITE: An Approach to Universal Topology Generation," in *Proceedings of the Ninth International Symposium in Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, ser. MASCOTS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 346–. [Online]. Available: http://dl.acm.org/citation.cfm?id=882459.882563
[49] M. Grant and S. Boyd, "CVX: Matlab software for disciplined convex programming, version 2.1," http://cvxr.com/cvx, Mar. 2014.
[50] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
[51] S. Mirzaei and F. Esposito, "An alloy verification model for consensus-based auction protocols," in *Inter. IEEE Workshop on Assurance in Distributed Systems and Networks, In conjunction with IEEE ICDCS 2015*, ser. ADSN, 2015.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers, and Dr. Robert Ricci for handing us the non-public Emulab dataset. This work is supported in part by the National Science Foundation under grant CNS-0963974.

Flavio Esposito (M'11) received his Ph.D. in computer science at Boston University in 2013, and his Master of Science in Telecommunication Engineering from University of Florence, Italy in 2005. Flavio is currently a member of the Advanced Technology Group at Exegy, Inc. and a Visiting Research Assistant Professor in the Computer Science & IT Dept. at University of Missouri, Columbia. His research interests include network management; design, implementation and evaluation of algorithms and protocols for service-based architectures, such as Software Define Networks (SDN) and Delay-Tolerant Networks (DTN); modeling and performance evaluation of wireless, and peer-to-peer networks. Flavio worked at Alcatel-Lucent, interned at Bell Laboratories, Holmdel, NJ, at Raytheon BBN Technologies, Cambridge, MA, and at EURECOM, France. He was also a visiting researcher at MediaTeam, Oulu and at the Center for Wireless Communications, Oulu, Finland. He is a member of both the ACM and the IEEE.

Ibrahim Matta (M'93-SM'06) received his Ph.D. in computer science from the University of Maryland at College Park in 1995. He is a professor of computer science at Boston University. His research involves network protocols, architectures, and performance evaluation. His current projects include recursive networks, their management and economics implications, and their experimental evaluation. He has published over 100 peer-reviewed articles. He received the National Science Foundation CAREER award in 1997, and won a patent (in 2011) and two best-paper awards (in 2008 and 2010) on his work on wireless ad hoc and sensor networks. He has served as the chair or co-chair of many technical committees, including IEEE 2011 CCW and 2005 ICNP. He is a senior member of ACM and IEEE.

Yuefeng Wang Yuefeng Wang (S'14) is a Ph.D. candidate in CS Dept. at Boston University. He received his Master of Science in Computer Science from University of Windsor, Canada in 2010. His research interests include design, implementation and evaluation of future network architecture; performance evaluation of network management architecture. Yuefeng worked as a research intern at Akamai Technologies, Cambridge, MA, focusing on platform performance. He is a student member of the IEEE.

11 VINEA Asynchronous Agreement Rules

In this appendix we report the conflict resolution table rules used in the VINEA asynchronous implementation of the VINO protocol, see Table 11.

As defined in Section 3, 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. The first column of Table 11 shows the view of the sender physical node k on the assignment (or winner) vector $\mathbf{a}$. The second column shows the view on the receiver physical node, and the third column the receiver's action to guarantee asynchronous consensus. In Table 11 vector $\mathbf{b}_i \in \mathbb{R}_+^{|V_H|}$ is the a vector of utility values. Each entry $b_{ij} \in \mathbf{b}_i$ is a positive real number representing the highest utility value known so far on virtual node $j \in V_H$. $\mathbf{a}_i \in V_G^{|V_H|}$ is the winner vector —a vector containing the latest information on the current assignment of all virtual nodes. $a_{ij} \in \mathbf{a}_i$ is the identity of the winner of virtual node j, as currently known by physical node i. There are three possible actions when a physical node i receives a bid 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, 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.

The time stamp vector $\mathbf{t}_i \in \mathbb{R}_+^{|V_H|}$ is a vector of time stamps where each entry $t_{ij} \in \mathbf{t}_i$ 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. We implemented these rules in our VINEA prototype, whose code is available at [12] while we verified and discussed the correctness of our asynchronous max-consensus auction strategy used with an Alloy [50] model in [51].

k **thinks** $\mathbf{a}_{kj}$ **is**	i **thinks** $\mathbf{a}_{ij}$ **is**	**Receiver's action (default leave & no br.)**
k	i	if $\mathbf{b}_{kj} > \mathbf{b}_{ij} \rightarrow$ update and rebroadcast if $\mathbf{b}_{kj} = \mathbf{b}_{ij}$ & $\mathbf{a}_{kj} < \mathbf{b}_{ij} \rightarrow$ update & rebr. if $\mathbf{b}_{kj} < \mathbf{b}_{ij} \rightarrow$ update time & rebroadcast
	k	if $\mathbf{t}_{kj} > \mathbf{t}_{ij} \rightarrow$ update & rebroadcast if $\lvert\mathbf{t}_{kj} - \mathbf{t}_{ij}\rvert < \epsilon \rightarrow$ leave & no broadcast if $\mathbf{t}_{kj} < \mathbf{t}_{ij} \rightarrow$ leave & no rebroadcast
	$m \notin \{i, k\}$	if $\mathbf{b}_{kj} > \mathbf{b}_{ij}$ & $\mathbf{t}_{kj} \geq \mathbf{t}_{ij} \rightarrow$ update & rebr. if $\mathbf{b}_{kj} < \mathbf{b}_{ij}$ & $\mathbf{t}_{kj} \geq \mathbf{t}_{ij} \rightarrow$ leave & rebr. if $\mathbf{b}_{kj} = \mathbf{b}_{ij} \rightarrow$ leave & rebroadcast if $\mathbf{b}_{kj} < \mathbf{b}_{ij}$ & $\mathbf{t}_{kj} < \mathbf{t}_{ij} \rightarrow$ rebroadcast if $\mathbf{b}_{kj} > \mathbf{b}_{ij}$ & $\mathbf{t}_{kj} < \mathbf{t}_{ij} \rightarrow$ update & rebr.
	none	update & rebroadcast
i	i	if $\mathbf{t}_{kj} > \mathbf{t}_{ij} \rightarrow$ update & rebroadcast if $\lvert t_{kj} - t_{ij}\rvert < \epsilon \rightarrow$ leave & no-rebroadcast if $\mathbf{t}_{kj} < \mathbf{t}_{ij} \rightarrow$ leave & no rebroadcast
	k	reset & rebroadcast*
	$m \notin \{i, k\}$	$\rightarrow$ leave & rebroadcast
	none	$\rightarrow$ leave & rebroadcast*
$m \notin \{i, k\}$	i	if $b_{kj} > b_{ij} \rightarrow$ update and rebroadcast if $b_{kj} = b_{ij}$ and $a_{kj} < a_{ij} \rightarrow$ update and rebr. if $b_{kj} < b_{ij} \rightarrow$ update time and rebroadcast
	k	if $b_{kj} < b_{ij} \rightarrow$ update and rebr. (sender info) if $t_{kj} > t_{ij} \rightarrow$ update and rebroadcast if $\lvert t_{kj} - t_{ij}\rvert < \epsilon \rightarrow$ leave and no rebroadcast if $t_{kj} < t_{ij} \rightarrow$ leave and rebroadcast
	$n \notin \{i, k, m\}$	if $b_{kj} > b_{ij}$ and $t_{kj} \geq t_{ij} \rightarrow$ update and rebr. if $b_{kj} < b_{ij}$ and $t_{kj} < t_{ij} \rightarrow$ leave and rebr. if $b_{kj} < b_{ij}$ and $t_{kj} > t_{ij} \rightarrow$ update and rebr if $b_{kj} > b_{ij}$ and $t_{kj} < t_{ij} \rightarrow$ leave and rebr
	none	update and rebroadcast
none	i	leave and rebroadcast
	k	update and rebroadcast
	$m \notin \{i, k\}$	update and rebroadcast
	none	leave and no rebroadcast
Legend	rebroadcast	alone, with leave, broadcast receiver states with update time, broadcast receiver states with update, broadcast sender states with reset, broadcast sender states
	rebroadcast*	broadcast empty bid with current time