VINEA: A Policy-based Virtual Network Embedding Architecture

Flavio Esposito Ibrahim Matta
flavio@cs.bu.edu matta@cs.bu.edu
Computer Science Department
Boston University

Technical Report BUCS-TR-2013-010

Abstract

Network virtualization has enabled new business models by allowing infrastructure providers to lease or share their physical network. To concurrently run multiple customized virtual network services, such infrastructure providers need to run a virtual network embedding protocol. The virtual network embedding is the (NP-hard) problem of matching constrained virtual networks onto the physical network.

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

1 Introduction

Network virtualization is a technology that enables multiple virtual instances to co-exist on a common physical network infrastructure. This paradigm has opened up new business models, enabling Infrastructure Providers (InPs) to lease or share their physical resources. Each virtual network is in fact customizable in support of a wide range of customers and applications.

To support such applications, InPs need to embed the virtual networks on their infrastructure. The virtual network (VN) embedding is the (NP-hard) problem of matching such constrained virtual networks onto a physical network, owned by a single provider, or by multiple federated providers. In particular, we identify three interacting mechanisms in the embedding problem: resource discovery, virtual network mapping, and allocation. Resource discovery is the process of monitoring the state of the substrate (physical network) resources. The virtual network mapping mechanism matches requests for virtual networks with the available physical resources, selecting some set of physical components to host the virtual network. The allocation mechanism involves assigning the physical resources that match the VN requests to the appropriate virtual network, considering additional constraints, e.g., the physical limits of the infrastructure.

Existing embedding solutions focus on specific policies under various settings. By policy, we mean a variant aspect of 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 of the physical network resources with minimal virtual machine or path migrations [58]. Other solutions show how the physical network utilization increases by simultaneously embedding virtual nodes and links [10, 41].

Distributed solutions for embedding wider-area virtual networks also exist [26, 9, 62]. Some of them outsource the embedding to a centralized Service Provider (SP) that coordinates the process by either splitting the slice and sending it to a subset of InPs [26], or by collecting resource availability from InPs and later offering an embedding [62].1 Outsourcing the embedding has the advantage of relieving InPs from the entire management complexity, but a single centralized authority [62] could be untrusted, a single point of failure, or both. Although they have systematic logic behind their design, such distributed solutions are limited to a single distribu-

1Slice is an alternative term for virtual network. The physical network is in fact “sliced” into multiple virtual networks.
tion model — the type and amount of information propagated for the embedding.

Both centralized and distributed existing solutions are also restricted to a subset of the three embedding tasks [17]: some do not consider the discovery phase [58, 43], assuming full knowledge of the physical resource availability, while others leave the final allocation decision to the virtual network requester, that chooses the hosting physical resources among a set of candidates [50, 60]. Consider for example solutions ignoring the resource discovery mechanism: the lifetime of a virtual network can range from a few seconds to several months; assuming complete knowledge of the network state, and ignoring the overhead of resource discovery and the embedding time, may be acceptable in wide-area testbed applications [50, 23], as the inter-arrival time between virtual network requests and the lifetime of such requests are typically much longer than the embedding time. In high performance applications instead, where short response time is crucial, e.g., in cluster-on-demand applications such as financial trading, anomaly analysis, or heavy image processing, the resource discovery time may introduce severe delays, that may lead to Service Level Agreements (SLA) violations; for those applications, a resource discovery policy able to limit the overhead would be more appropriate.

In summary, due to the wide range of virtual network applications, providers’ goals and allocation models (e.g., best effort or SLA), a system that tackles the complete virtual network embedding with its three phases, and is able to adapt to different service and infrastructure provider’s goals by instantiating the appropriate policies, to our knowledge, does not yet exist. To this aim, leveraging our earlier system design work on network architectures [18, 14], and our theoretical and simulation results on the virtual network embedding problem [17, 16], we present the design and implementation of a VIrtual Network Embedding Architecture (VINEA), that allows virtual network embedding programmability.

We also propose an object model as a foundation for a protocol specification (Section 3.) The object model consists of: (i) a set of objects, whose attributes can be customized to instantiate different policies, (ii) an interface to such object attributes, and (iii) a set of operations (protocol messages) to share and modify the object attributes. We prototyped our architecture, overviewed in Section 2, including support for all three virtual network embedding mechanisms, for both service and infrastructure providers (Section 4.)

To demonstrate our implementation, we released a virtual network embedding testbed. 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 that leverages Mininet [38]. 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 [51] running in kernel mode to switch packets across virtual interfaces (Section 5.)

Our testbed can be used to experiment with novel embedding policies, or to run virtual network applications in real settings. VINEA can be used as an alternative solution for the stitching problem in wide-area virtual network testbeds as GENI [22], but with guarantees on embedding convergence and performance. Moreover, our prototype can be ported into existing open source “networking as a service” solutions such as OpenStack Neutron [48], that assumes knowledge of the hosting server name before setting up a virtual network.

2 VINEA Overview

In this section we describe the main operations performed by VINEA in embedding a virtual network. We bootstrap an overlay of InP processes that are responsible for allocating resources over the underlying physical network. Such processes participate into a distributed, consensus-based, VN mapping protocol, and then run a final resources reservation.

2.1 InP Overlay Support

Each VINEA node (SP or InP) is authenticated into an InP overlay with a private addressing scheme, to later host virtual network requests. Our architecture starts with the creation of such private overlay. In particular, a Network Management System (NMS) enrolls the new VINEA nodes into the private overlay. The enrollment procedure consists of an authentication (with user and password) and a policy exchange (Figure 1a.) Examples of such policies, whose scope is limited to the InP overlay, include routing update frequency, or address of neighbor InP processes, including a set of SPs that may publish VN requests. Once the enrollment procedure is completed, the NMS starts monitoring the availability of each newly enrolled VINEA node. If the VINEA node is instantiated as an InP, it may also subscribe to at least an SP using a pub/sub mechanism.

2.2 Asynchronous Embedding Protocol

Once the InP overlay is bootstrapped, an SP encodes a request into a Slice object, and publishes it using its

\footnote{In previous work we have shown how such virtual private (overlay) networks of InPs can be dynamically instantiated avoiding the mobility and multihoming shortcomings of the IP protocol [32].}
subscription mechanism (Figure 1b), so that the set of subscriber InP processes can run our Consensus-based Auction for Distributed Embedding protocol (CADE.) CADE is asynchronous and has two phases: a virtual node embedding phase, and a link embedding phase, that can be run sequentially, as in [59], or simultaneously as in [10], but in a centralized or distributed fashion. Each InP process independently bids on a single, or on multiple virtual nodes (depending on the policies), trying to maximize its private utility function (another policy of each InP process.) After bidding on the virtual resources (virtual nodes, virtual paths or both), InP processes exchange their bids and run a max-consensus [45] protocol for a distributed auction winner determination (Figure 1c.) Virtual links are then set up on loop-free physical paths among the winner InP processes, using a k-shortest path algorithm [15]. InP processes exchange also their bid creation times to resolve conflicts when messages arrive out of order.

In [16], we have shown bounds on convergence time and embedding optimality, for an initial synchronous simulated version of our virtual network mapping protocol, without running a resource discovery, and without reserving real bandwidth and CPU.

2.3 Policies and Allocation

Our object model allows policy programmability for both nodes and link embedding policies (Section 3.) Together with the VN constraints, an SP publishes also a set of embedding policies, that are piggybacked from InP processes with the first bid 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 [27, 60].

In our prototype evaluation (Section 6), we tested two representative VINEA embedding policies: the first has an SP release a single virtual network partition at a time, and InP processes bid on a single virtual node in each auction round. We call this policy Single Auction for Distributed embedding (SAD.) The second policy permits an SP to release the entire virtual network at once, and InP processes are allowed to bid on multiple virtual nodes at the same time; we call this policy Multiple Auction for Distributed embedding (MAD.) Note how VN partitioning is not required in MAD, and the entire VN request is released at once.

When the SP receives a positive embedding response from one InP, the allocator service interface is used to start the allocation phase. Virtual nodes and switches are created, and bandwidth among winner InP processes are reserved using the Mininet library [38].

3 Protocol and Object Model Design

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

The objects are updated by the InP processes participating in the virtual network embedding, and stored into a distributed data structure called Slice Information Base (SIB.) As the Management Information Base (MIB) defined in [35] or the Network Information Base (NIB) defined on Onix [36], our SIB is a partially replicated distributed object-oriented database that contains the union of all managed objects within a slice to embed, together with their attributes. In the NIB, attributes are elements of the forwarding table. The SIB represents a generalized case of the Routing Information Base stored in IP routers. Rather than only storing prefixes to destinations, our SIB stores all the states accessible by each component of our virtual network embedding architecture. An example of attribute is a list of virtual nodes of a given virtual network, or the list of InP processes currently mapping a given virtual link. A broker (or SIB daemon) handles the read/write operations on the SIB. The role and responsibilities of the SIB daemon are similar to those of memory management in an operating system: to manage the information stored in the SIB and its veracity, updating and making states available to InP processes participating in the virtual network embedding.

Based on a publish/subscribe model, a distributed set of SIBs and SIB daemons enable infrastructure and service providers to specify different styles of embedding management architectures, ranging from fully decentralized, i.e. autonomic, to centralized, i.e. manager-agents style, to hybrid approaches, e.g. hierarchical: InPs can in fact participate in a distributed embedding and host a set of virtual nodes requested from an SP, and then use the same mechanism to lease the acquired virtual resources to other InPs.

In the rest of this section we describe the broker architecture (Section 3.1), the abstract syntax used to define the objects (Section 3.5), as well as the interface (Section 3.2) and the operations on such objects (Section 3.3), that is, the CADE protocol used to modify the object attributes, such as the InP overlay states during a virtual
Figure 1: (a) A Network Management System (NMS) authenticates and enrolls via a policy exchange a new VINEA node into an InP overlay, assigning a private address whose scope is limited to the overlay. (b) A service provider requests a virtual network embedding from an infrastructure provider. (c) Two InP processes (belonging to possibly different InPs) use our asynchronous consensus-based protocol to embed the virtual network request: InP processes flood the VN request as well as their bids.

network embedding.

3.1 Broker (SIB Daemon)

Similar to traditional existing network service management object models [30, 47], our architecture has a broker responsible for allowing InP processes participating in a virtual network embedding to transparently make requests and receive responses. The broker handles such communication with subscription events. A subscription represents the quantity and type of information to propagate on objects in predefined situations by an embedding instance when specific situations occur. Publishers are SP or InPs, and examples of object attributes being published are the constraints of a virtual network to be embedded, or the routing updates among InP processes participating in a distributed virtual network embedding.

Subscription events are mapped by the broker that recognizes the objects from their type, and acts upon different requests with a set of operations on the objects stored in a local or a remote SIB, on behalf of an embedding application instance. Our subscriptions have equivalent design goals as the notification events defined by the OSI model [34], or traps in the Simple Network Management Protocol (SNMP) [7], though they specifically support virtual network embedding operations.

3.2 SIB Interface

We provide an API to simplify the development of sophisticated virtual network embedding solutions. Leveraging the separation between mechanisms and policies, VINEA allows InP processes to potentially read and write any state — set of object attributes — of any other InP process participating in a virtual network embedding (Figure 2.) Object attributes can be read or written through a general subscription mechanism that includes registration for passive (subscribe) or active (publish) notifications of local or remote state changes.

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

3.3 CADE Protocol (Operations on Objects)

To share or modify the states of our architecture, such as routing updates, or virtual to physical mapping information, we define a set of operations executable on (re-
Figure 2: SIB Interface: the SIB daemon is responsible for managing the information stored in the SIB and its veracity, updating and making states available to service and infrastructure provider processes.

The mandatory slice ID field can be encrypted (using public-key cryptography) to avoid malicious InP processes changing the slice specification. It is invoked when an InP process has terminated a virtual network mapping agreement phase, and as a result, the bid data structures need to be propagated for a distributed virtual network embedding. The payload of this message contains the bid objects of the sender InP process.

Upon receipt of this message, an InP process attempts to overbid the requested virtual nodes, and when necessary, the InP process propagates its bids. As the distributed embedding uses a max-consensus strategy on the bids, convergence is guaranteed only if InP processes are not allowed to overbid on “lost” virtual nodes. After sending the first bid message, a bid timeout is set. This is necessary for the asynchronous consensus to terminate. When the virtual nodes have all been assigned, the receiver InP process replies with an Embed Success message to the service provider. Else, an Embed Failure message is sent to SP.

Bid: this primitive is similar to the First Bid, except it does not piggyback the slice specification. It is invoked when an InP process has terminated a virtual network mapping agreement phase, and as a result, the bid data structures need to be propagated for a distributed virtual network embedding. The payload of this message contains the bid objects of the sender InP process.

If the receiver InP process is able to overbid, when receiving this message the bidding data structures are updated and another bid message is propagated; the propagation occurs if at least an InP process have subscribed to i’s bids. InP processes that subscribe for bids after a request has been issued may not participate in the ongoing embeddings.

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

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

Link Embedding: this primitive is invoked by the service provider after receiving an Embed Success message from an InP process. In case the embedding of a virtual link (v1, v2) is needed, upon receipt of this message, the InP process winner of the first end v1 sends to the winner InP process of the other end v2 a Connect request message.

Conect Request: this primitive is invoked by an InP

3The mandatory slice ID field can be encrypted (using public-key cryptography) to avoid malicious InP processes changing the slice request.
process after receiving a Link Embedding message from the service provider. InP processes that receive this message, reply by sending a Connect Response to the other end of the virtual link to embed, and update their states with the final physical to virtual resource binding. **Connect Response:** this primitive is invoked by an InP process after receiving a Connect Request from a service provider or from an InP process. Upon receipt of this message the bandwidth is reserved using the virtual network allocation service interface.

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

**Slice Release Response:** upon receipt of this message, the receiver node releases its reserved virtual resources after terminating all the applications running on each hosted virtual node.

### 3.4 Timers are Enough

InP processes using our max-consensus strategy only need to propagate the maximum bid on each virtual node. Only the InP process that received the embedding request from the SP is responsible to respond to the SP, ensuring state consistency across the InP overlay.

Choosing the "best" value for the timeout parameter would require knowledge of both InPs processing time and network failure models: a valid timeout would in fact require an estimation of the delay of traversing the diameter of the InP overlay, and the processing time at each InP process. When a timeout is set to a positive value, our protocol assumes that InP processes are unavailable or unable to (over)bid when no response is received within the timeout, and future (late) messages are discarded.

### 3.5 Virtual Network Embedding Objects

**Abstract Syntax:** We define each object using the Google Protocol Buffer (GPB) abstract syntax [24]. One of the main advantages of using an abstract syntax is the implementation independence of the framework, i.e., the objects can be serialized and deserialized using any programming language, to enable different (VINEA) implementations to coexist. Many object serialization languages have been proposed. We can classify them into binary serialization, e.g., Binary JavaScript Object Notation (BSON) [31] and Google Protocol Buffer (GPB) [24], and character-based, e.g. the Extensible Markup Language (XML) [57] or the Abstract Syntax Notation 1 (ASN.1) [39]. Character based representations as XML are more powerful than what we need, which leads both to unnecessary complexity and size in implementation. Binary serializations like BSON and GPB are order-of-magnitude more efficient to parse than XML for example, depending on how rich is the XML parser. BSON was a good candidate, but we choose GPB as it provides a compiler to serialize and deserialize the objects.

### 3.6 CADE Protocol Objects

GPB denotes the syntax of each object data structure with the reserved keyword `message`. Each message’s attribute defined with the GPB abstract syntax must have a unique numbered tag associated with it. These tags are used to identify attributes in the message binary format. Tags with values in the range 1 through 15 take one byte to encode, including the identifying number and the field’s type. Tags in the range 16 through 2047 take two bytes and so on. To increase readability we have omitted the tags in our objects definition; the complete Google Protocol Buffer proto files are available at our github repository linked at [19]. In the rest of this chapter we will assume that the reader is familiar with the GBP abstract syntax notation [24].

#### 3.6.1 Format of a CADE Object

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

```protobuf
text message CADE {
  // required fields
  required int32 version
  required int32 sliceID
  // optional fields
  optional Slice sliceRequest
  optional string allocationPolicy
  // repeated fields
  repeated assignment a
  repeated bid b
  repeated int32 m
  repeated bidForgingTime timeStamp
}
```

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

The attribute `sliceID` is an identifier that must remain unique for the entire lifetime of the slice (virtual network), within the scope of both service and

---

4 VINEA also allows SPs to send the same embedding request to multiple InP processes as in [9, 60], by assigning to each request a unique identifier within the scope of the InP overlay.
fracestructure providers. It is a 32 bit identifier, and it could be any integer, including an hash value of the string sliceproviderName.sliceID. The field allocationPolicy allows service providers to specify different virtual network embedding strategies. This attribute is used to specify the form of the assignment vector.

3.6.2 Format of the Assignment Object

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

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

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

Note that if the allocationPolicy is set to its most informative form, the assignment vector reveals information on which physical nodes are so far hosting vNodeID, whereas if the allocationPolicy is set to its least informative form, each physical node only knows if vNodeID is currently being hosted by a physical node or not.\(^5\)

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

3.6.3 Format of the Bid Objects

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

```protobuf
message bid {
  required int32 vNodeId
  optional double bidValue
}
```

```protobuf
message bidTime {
  required int32 vNodeId
  optional int64 time
}
```

In section 3.3 we mentioned how the bidding time is essential to solve conflicts in CADE (agreement phase), and that the time at which the message is sent or received are not sufficient to guarantee convergence to an embedding agreement. This is because CADE is an asynchronous communication protocol, and messages from different sources may arrive out of order, i.e., messages created earlier than another message could potentially arrive at a later time.\(^6\) Every time a physical node bids on a virtual node identified by the vNodeId attribute, a bidValue and the time attributes are forged.

3.6.4 Format of the Slice Specification Object

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

```protobuf
message Slice {
  required int32 sliceID
  optional int64 entryTime
  optional int64 exitTime
  optional string topology
  optional string predicate
  repeated vNode virtualnode
  repeated vLink virtuallink
}
```

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

\(^5\)Note how, as a consequence of the max-consensus, when using the least informative assignment policy, each physical node only knows the value of the maximum bid so far without knowing the identity of the bidder.

\(^6\)Note that CADE is an application protocol, and so it does not perform transport functionalities; this means that message reordering from the same source are not a problem for CADE as they are handled by the reliable transport protocol on which CADE relies on.
3.6.5 Format of Virtual Node and Virtual Link Objects

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

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

The attribute vNodeId is the unique virtual node identifier while vNodeCapacity represents the requested capacity. The vNodeType attribute enables additional expressiveness in the slice constraint specifications. For example, small, large or extra-large virtual node type, as in Amazon EC instance [3]. The vNodeName and the vNodeClass attributes allow the specification of a hierarchical object model for virtual nodes. For example, the vNodeName may be used to specify the name (address) or the region (e.g., the country or the subnetwork) of the physical node on which the virtual nodes must be embedded, while the vNodeClass attribute might be set to geolocation to indicate that this virtual node has a geolocation constraint, specified by the vNodeName attribute.

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

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

3.6.6 Format of the Error Code Object

The CADErrorCode object is needed to specify the particular type of errors that may be encountered. The CADE error code design was inspired by the HEMS protoErrorCode [52]. The abstract syntax defines two fields: a required error code integer, and an optional message description.

```protobuf
def message CADErrorCode {
  required int32 eCode
  optional string description
}
```

The description field gives a more detailed description of the particular error encountered, while the error code integer are defined as follows:

0 - Reserved. This error code is not used.

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

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

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

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

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

VINEA processes join a private overlay before running the CADE protocol to embed the request released by a VINEA node instantiated as service provider. Then InP processes run a physical resource discovery protocol, the asynchronous virtual network mapping phase, and finally, the virtual network is allocated using the Mininet library [38]. Our prototype is implemented in a single host Linux-based testbed (Section 5), and its InP overlay resources are simulated, i.e., physical CPU and link available capacity are not measured but set from a configuration file, and updated as virtual networks are being embedded. Also, the InP overlay connectivity is emulated by TCP connections on the Linux loopback interface. We emulate the allocation phase of the embedding problem by reserving CPU on virtual hosts, attached to virtual switches running in kernel mode, and we use the Linux Traffic Control application to reserve link capacity. Once the virtual network allocation phase is complete, we run real applications such as ping, iPerf, and Openflow [46].

Our VINEA prototype resulted in about 35K lines of Java code, without considering comments, test classes, and the Mininet [38] Python and the C code that VINEA leverages for the virtual link allocation. Logically, the prototype is divided into nine main architecture components (Figure 3): a Network Management system (NMS), the three embedding services of an infrastructure provider —resource discovery, virtual network mapping and allocation, a set of service provider functionalities, a Slice Information Base (SIB), a broker (or SIB daemon), a message parser to serialize and deserialize objects, and a publish/subscribe system. In the rest of this section we describe in detail each of these components and their functionalities.

4.1 Common VINEA Node Capabilities

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

4.1.1 Network Management System

Network Monitoring: in the network management literature, a Network Management System (NMS) is an architecture component usually responsible for monitoring, control, and repair functionalities of a physical network. The NMS component of our architecture includes an InP overlay monitoring task, as in an NMS of a typical telecommunication network, and an identity manager, similar to the Keystone component of the OpenStack architecture [49]. The network monitoring task is a thread that sends at a configurable rate keep-alive messages to all InP processes of the monitored network. When an InP process does not respond to a keep-alive message, the NMS publishes an event to update the members of the InP overlay about the failure status of such node.

Identity Manager: when bootstrapping the InP overlay, or when a new VINEA node wishes to join an existing InP overlay, the identity manager is responsible for authenticating such processes, so that each process can be trusted. Our current VINEA implementation [19] supports two authentication policies: “no authentication” — every InP process requesting to join an existing InP overlay is automatically accepted — and authentication with ID and password. In the latter case, the authentication information are to be specified as a clear text in the private InP process configuration file. We separated the identity manager mechanism from its policies, so that other authentication policies may be easily supported, e.g., a public key encryption scheme such as RSA [54].

InP Overlay Connectivity and DNS: the Domain Name System (DNS) component is not part of the VINEA architecture (and is not shown in Figure 3), but it is a necessary artifact of our InP overlay connectivity implementation.

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

### 4.1.2 Slice Information Base (SIB)

As described in Section 3, the SIB architecture component is responsible for maintaing the object attributes and managing their veracity. Each VINEA node runs an instance of a SIB daemon, responsible for updating such states within the InP overlay, and for creating new states through our pub/sub mechanism.

We support multiple threads accessing the database efficiently, with a synchronized hash table, and we exposed the SIB interface to enable different implementations. An alternative SIB implementation could use an open source object database as db4o [13].

### 4.1.3 Publish/Subscribe Service

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

### 4.2 Service Provider Capabilities

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

The virtual network generator translates incoming virtual network requests into slice objects, that are later serialized and sent to the InPs. The virtual network partitioning problem is NP-hard [27]. VINEA supports a simple virtual network partitioning heuristic, that merely extracts sequentially the next yet-to-be-embedded virtual link from the virtual network request. The partition being sent is hence formed by a single virtual link, and its two adjacent virtual nodes. Each service provider has an interface to the virtual network partitioning service, enabling support for additional (more complex) virtual network partitioning implementations, for example a “hub-and-spoke” heuristic as proposed in [28].

### 4.3 Infrastructure Provider Capabilities

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

#### 4.3.1 Resource Directory Service

The resource discovery service is the logical set of mechanisms needed to collect and propagate physical and virtual network states such as neighbor discovery, or physical resource availability. The neighbor discovery is useful for the InP overlay monitoring operation performed by the network management system (Section 4.1.1), while the knowledge of the available physical resource is used by the virtual network mapping and allocation services to make informed embedding decisions. The resource discovery service can be divided into two architecture components: (i) registration and bootstrap, and (ii) discovery.

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

**Inter-Slice Discovery:** after the DNS registration, necessary for InP overlay connectivity, InP processes register also with an Inter-Slice Discovery service (ISD) in order to establish a private InP overlay [56]. The ISD component of the architecture can be thought of a DNS across all private InP overlays potentially hosting a virtual network. An InP process may wish to participate in the embedding of a particular virtual network, being unaware of whether there are other InP processes currently bidding on it.

When a physical VINEA node belonging to some InP subscribes to an SP to participate in a distributed embedding, it queries the ISD service to obtain the (IP) address of the network management system in charge of the authentication (Section 4.1.1.) If the authentication is successful, the network manager enrolls the new VINEA node enforcing the policies instantiated on that particular InP overlay. Examples of such policies include node and link embedding policies (Section 2.3) or a given subset of all InP processes currently in the InP overlay, so that the new VINEA node may subscribe to their message updates.

**Enrollment:** we define by enrollment the procedure of authentication and policy exchange among ISD, NMS
and the new VINEA node. Only the VINEA nodes enrolled in a private InP overlay are allowed to later exchange CADE messages to participate in a virtual network embedding. We say that the InP overlay is private as it uses customized private addresses. VINEA nodes do not process incoming CADE messages whose source is not a member of a private InP overlay.  

The ISD service, when queried with a slice identifier, returns the (IP) address of the manager of the InP overlay that is currently or has previously embedded a given slice. We implemented the ISD service as a single centralized synchronized database. Each VINEA node has an interface to query the ISD service. The modularity of our prototype however enables alternative (distributed) ISD implementations: we envision a more scalable scenario with many peer ISDs, each one containing a partially replicated subset of all the objects, communicating to retrieve the queried object. In such distributed cases, a request to the ISD is forwarded across the peer ISDs until the destination application (the ISD process that contain the sought NMS address is found), or until a predefined termination condition is met (a policy to limit the discovery overhead.)

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

4.3.2 Virtual Network Mapping Service

This service is responsible for deciding which InP process hosts which virtual node, and what physical loop-free path hosts each virtual link using the CADE protocol (Section 3.3.) After the InP overlay bootstrapping phase, InP processes subscribe to the slice objects to be released by a service provider, and bid on virtual resources as they receive embedding requests. In our implementation, we have assumed that the service providers’ names are known after reading them from an InP process configuration file, while their addresses are dynamically resolved with DNS. We could however also acquire the name (or address) of the service provider from the NMS during the bootstrapping phase.

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

Agreement Phase: in our system implementation of the CADE protocol, we relaxed the assumption of a synchronous conflict resolution phase. By relaxing this assumption, the synchronous rules in Table [2], used for our simulations became invalid to correctly update the InP process states on a asynchronous distributed virtual network embedding protocol. In particular, for an asynchronous conflict resolution, we needed to a (i) concept of re-broadcasting a message, and (ii) a new concept of time-stamp $t$, that is, the time at which the bid was generated, as opposed to the time $s$ at which a bid message is received in the synchronous version of CAD.

When a VINEA bid message is sent from InP process $k$ and received by InP process $i$, the receiver follows the rules in Tables 2 and 3 to resolve the allocation conflicts asynchronously. If none of the conditions in such conflict resolution tables is met, the receiver InP process $i$ applies the default rule, that is “leave” its states as they are without broadcasting any update. We denote with $b_{ij}$ the value of the bid known by InP process $i$ on virtual node $j$, while with $t_{ij}$ we denote the time at which the bid on virtual node $j$ was made by InP process $i$. $\epsilon$ is a small positive number. For a correct asynchronous agreement phase, the receiver InP process may need to rebroadcast (propagate) the sender states, or the receiver states. In particular:

- If rebroadcast is coupled with leave or with update, the receiver broadcasts its own CADE states.
- If rebroadcast is coupled with update or with reset, the receiver broadcasts the sender’s states.

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

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

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

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

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

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

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

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

### 4.3.3 Virtual Network Allocator Service

Each VINEA node has an interface to the Virtual Network Allocator Service. We provide a Mininet-based [38] implementation for the final binding between physical and virtual resources. When an InP process returns a positive embedding response to the service provider, indicating that an embedding of the slice has been successfully found, the Virtual Network Embedding Service parses the input from a `Slice` object, and uses the Mininet library to generate and bootstrap a virtual network.

#### 4.3.4 Resource Binding Implementation

For each InP process hosting at least one virtual node, we need to fork a virtual switch, and attach a virtual host to it (Figure 4.) For any InP process there exists a virtual switch, and for any virtual node hosted on that InP process, the VINEA allocation service creates a separate interface to the same virtual switch. A virtual switch is implemented using the Open Virtual Switch reference implementation libraries [51]. We use the Mininet 2.0 default Open Virtual Switch (OVS) controller, that supports up to 16 switches. By leveraging the Mininet interface, VINEA can also configure CPU limits for each virtual host.

After setting up all virtual hosts and virtual switches, the allocation service configures support for SSH, so that an application running on top of the virtual network can log into each virtual host (our default settings do not require any password.) Finally, the virtual links are set up connecting the virtual switches, and the virtual hosts to the virtual switches. For each virtual link, a bandwidth limit can be set up using the Linux traffic control `tc` system call [40], introducing traffic shaping constraints, and emulating delay and losses on virtual links as needed.
Figure 5: Testbed Architecture: A physical machine running Linux Ubuntu (version 12.04) hosts the VINEA prototype. Physical wires are emulated with loopback TCP connections on well-known ports. After the virtual networks are embedded, we can run Linux applications between virtual nodes, e.g., ping, traceroute, or we can send data traffic, and measure the reserved bandwidth performance with iperf.

5 VINEA Testbed

In order to evaluate our prototype, we implemented a testbed whose architecture is shown in Figure 5. Our base system is a host running an Ubuntu distribution of Linux (version 12.04). The InP overlay is emulated via TCP connections on the host loopback interface. Each InP process includes the VINEA modules. Each emulated virtual node is a user-level process that has its own virtual Ethernet interface(s), created and installed with ip link add/set, and attached to an Open vSwitch [51] running in kernel mode to switch packets across virtual interfaces. A virtual link is a virtual Ethernet (or veth) pair, that acts like a wire connecting two virtual interfaces, or virtual switch ports. Packets sent through one interface are delivered to the other, and each interface appears as a fully functional Ethernet port to all system and application software. The data rate of each virtual link is enforced by Linux Traffic Control (tc), which has a number of packet schedulers to shape traffic to a configured rate. Within the generated virtual hosts, we run real Linux applications, e.g., ping, and we measure the reserved bandwidth performance with iperf between the virtual hosts.

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

6 VINEA Prototype Evaluation

The goal of this section is to show how, in real settings, different embedding policies may lead to different embedding performance —success rate— across representative virtual network topologies: linear, star, tree, and fully connected (Section 6.1.) We also dissect the architecture components responsible for the embedding protocol overhead, and compare against two representative embedding policies (Section 6.2.) We recently surveyed existing embedding solutions [17], and to our knowledge, we are the first to release a system architecture that solves the virtual network embedding problem with its three phases, therefore no comparison with existing approaches has been possible. We compare however SAD and MAD, the two representative embedding policies described in Section 2.

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

Utility model: all InP processes use the same utility (bidding) function. The goal of the experiment is to embed a set of 100 virtual networks, with one second inter-arrival time between requests, aiming to reach Pareto optimality $U = \max \sum_{i=1}^{N_p} \sum_{j=1}^{N_i} b_i x_{ij}$, subject to the embedding constraints, that is, the distributed auction aims to maximize the sum of the utility of every InP process. $N_p$ is the number of InP processes, $N_i$ the number of virtual nodes, $b_i$ the bidding (utility) function used by InP
processes, and \(x_{ij} = 1\) if an InP process \(i\) is hosting virtual node \(j\) and zero otherwise. Similarly to previous embedding (centralized) heuristics \([61, 59]\), in attempt to maximize the number of hosted virtual nodes while keeping the physical network load balanced, each InP process bids using its current load stress, \(i.e., b_i\), is the sum of the residual InP process capacity, plus the sum of the residual capacity of all its adjacent physical links.

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

### 6.1 Embedding Success Rate

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

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

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

3. **The number of virtual nodes or links to allocate significantly impacts the virtual network allocation ratio.** This (sanity-check) result is unsurprising. Comparing the virtual network embedding success rate results across different virtual network topologies, we observe that the allocation ratio decreases when we increase the number of virtual links to embed. Moreover, the allocation ratio always decreases as we attempt to embed virtual networks with more virtual links (Figures 6a to 6d).
6.2 Overhead

In this section we show how the MAD policy has lower overhead than the SAD policy, as no virtual network partitioning is needed from the service provider (Figure 7.) This result demonstrates how an SP can significantly limit the network overhead by selecting a single InP process to send their requests. The result is in contrast with other approaches [60, 9] in which an SP also assumes competition among InPs, but sends the same virtual network request to multiple federated InPs and then selects the best (e.g. the cheapest) embedding solution.

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

7 Related Work

Architectures for Network Virtualization: the rapid growth of cloud service markets has fostered significant efforts toward a standardization of a communication architecture for large networks of virtual machines. Two examples are the CloudStack [4] and the OpenStack [49] initiatives. Those architectures are more complex than VINEA, as they involve storage, and computation, not only a network virtualization component. As in OpenStack Neutron [48], we also have an API to create and manage virtual networks (objects): we can create (embed), update, and delete virtual networks, but our prototype implementation does not support subnet creation, or listing and filtering operations. VINEA uses a consensus-based protocol to decide how to embed the virtual network in a centralized or distributed environment. Also, being inspired by the RINA architecture [14], VINEA does not inherit the shortcomings of the IP protocol as in [4, 49], as each InP overlay has a (private) naming and addressing scheme.

Other virtualization-based network architectures have been prototyped, for a single [55, 25] or for multiple cooperating InPs [21], with [25] or without [21, 55] virtual link performance guarantees. VINEA focuses on the architecture of a virtual network embedding, and also provides guarantees, but on convergence time and on embedding optimality.

Virtual Network Embedding Solutions: most centralized or distributed solutions either solve a specific task of the embedding problem, or are hybrids of two tasks [17]. Some solutions jointly consider resource discovery and virtual network mapping [29, 1], discovery and allocation [2] (mapping single virtual machines), others only focus on the mapping phase [61, 42, 12], or on the interaction between virtual network mapping and allocation [59, 41]. Others yet consider solely the allocation step [5, 6, 37, 20, 11]. Moreover, there are solutions that
assume the virtual network mapping task is solved, and only consider the interaction between resource discovery and allocation [53]. Distributed solutions that allow different InPs to collectively embed a virtual network already exist [28, 27, 60, 9, 62]; some of them focus on the desirable property of letting InPs use their own (embedding) policies [9], while others rely on truthfulness of a virtual resource auction [60].

VINEA is the first architecture that includes an implementation of all three mechanisms of the embedding problem, and can be instantiated in centralized and distributed settings. Moreover, we also let InPs choose their own embedding policies as in [9], but we include also virtual network partitioning policies for SP, instead of forcing a partitioning heuristic as in [27]. Moreover, we release a testbed [19] to foster virtual network embedding policy-programmability.

Network Management Protocols and Architectures: network management has been historically split into the Open System Interconnection (OSI) paradigm, i.e. Common Management Information Protocol (CMIP) [34], whose goal has been to deal with distributed network management of large public telecommunication networks, and the Internet management, i.e. the Simple Network Management Protocol (SNMP) [7], whose management target has been limited to Internet devices. Object-based management architectures have also been proposed, [47, 44], the most popular being CORBA defined by OMG [47], an attempt to unify existing paradigms supporting various management systems (e.g. CMIP and SNMP.) We extract the commonalities of previous service management architecture efforts to instantiate an object model in support of the virtual network embedding problem, a specific virtual network management mechanism. In particular, VINEA uses an object-oriented paradigm similar to the CORBA and OSI models, as opposed to SNMP that represents information only with variables. The object model is however different, in the way operations on objects and notifications are defined: we operate on our objects using our own Common Distributed Application Protocol (CDAP) [14], and we use pub/sub events as opposed to notifications. VINEA is also different in its resource specification language. We use the Google Protocol Buffer [24] to specify data types and the managed objects, while to define data types, the OSI and SNMP models use the ASN.1 [33]. CORBA instead uses a less expressive Interface Description Language (IDL) [47]. Finally, VINEA allows InP processes to directly access management objects through the SIB-API as in CORBA. Through a Network Management System (Section 4.1.1), VINEA also supports the manager-agents paradigm, as in both the OSI and Internet models.

8 Conclusions
To concurrently run multiple (wide-area) customized virtual network services, infrastructure providers need to embed the virtual networks hosting such services. In this paper, we presented the design and implementation of VINEA, a policy-based architecture for embedding virtual networks on top of a shared physical infrastructure. By policy, we mean a variant aspect of any of the (invariant) mechanisms of the embedding problem: resource discovery, virtual network mapping, and allocation. Our design and implementation include an asynchronous distributed protocol to embed a virtual network over a single provider, or across multiple federated providers, and an object model as a foundation for a virtual network embedding protocol specification.

We compared the performance of representative policy configurations over a prototype implementation and analyzed their performance and overhead tradeoffs. Each VINEA node can be instantiated as a service or infrastructure provider. Our VINEA prototype can augment existing open source “Networking as a Service” solutions such as OpenStack, and it provides an alternative architecture (with guarantees) to the slice stitching problem for the GENI testbed, i.e., the problem of providing a virtual network testbed service using resources from federated and geographically distributed GENI aggregate managers [23]. We also released a local testbed [19] that enables users to test their own embedding policies, and run applications within the embedded virtual networks.

Acknowledgment
This work is supported in part by the National Science Foundation under grant CNS-0963974.

References


Appendix A: Synchronous Agreement Rules

In this appendix we report the conflict resolution rules used in the agreement phase of the synchronous CADE protocol. The conflict resolution rules were inspired by the Consensus-based Decentralized Auction (CBBA) algorithm used for decentralized robot task allocation [8].

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. Each entry $b_{ij} \in b_i$ is a positive real number representing the highest bid known so far on virtual node $j \in V_H$. Each entry $a_i \in V_G$ is the winner vector—a vector containing the latest information on the current assignment of all virtual nodes, for a distributed auction winner determination. $a_{ij} \in a_i$ contains the identity of the winner of virtual node $j$, as currently known from physical node $i$. $s_i \in \mathbb{R}^{[V_G]}$ is the a vector of time stamps of the last information update from each of the other physical nodes i.e., the message reception time. There are three possible action when a physical node $i$ receives a bid message from a sender physical node $k$: (i) update, where both the bid vector and the allocation vector are updated according to the sender information; (ii) reset, where the bid is set to zero, and the allocation vector to null, and (iii) leave, where both the bid vector and the allocation vector are left unchanged by the receiver physical node.

Appendix B: Asynchronous Agreement Rules

In this appendix we report the conflict resolution rules used in the asynchronous implementation of the CADE protocol. The allocation vector $a$ and the bid vectors $b$ are defined in Appendix 8. The time stamp vector $t_i \in \mathbb{R}^{[V_G]}$ is a vector of time stamps where each entry $t_{ij} \in t_i$ is a positive real number representing the forgoing time of the bid on virtual node $j$ as currently known from physical node $i$. This vector is necessary for an asynchronous conflict resolution.
<table>
<thead>
<tr>
<th>InP process $k$ thinks $a_{kj}$ is</th>
<th>InP process $i$ thinks $a_{ij}$ is</th>
<th>Receiver’s action (default leave)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k$</td>
<td>$i$</td>
<td>if $b_{kj} &gt; b_{ij}$ → update</td>
</tr>
<tr>
<td></td>
<td>$k$</td>
<td>update</td>
</tr>
<tr>
<td></td>
<td>$m \notin {i, k}$</td>
<td>if $s_{km} &gt; s_{im}$ or $b_{kj} &gt; b_{ij}$ → update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>none → update</td>
</tr>
<tr>
<td>$i$</td>
<td></td>
<td>leave</td>
</tr>
<tr>
<td></td>
<td>$k$</td>
<td>reset</td>
</tr>
<tr>
<td></td>
<td>$m \notin {i, k}$</td>
<td>if $s_{km} &gt; s_{im}$ → reset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>none → leave</td>
</tr>
<tr>
<td>$m \notin {i, k}$</td>
<td>$i$</td>
<td>if $s_{km} &gt; s_{im}$ and $b_{kj} &gt; b_{ij}$ → update</td>
</tr>
<tr>
<td></td>
<td>$k$</td>
<td>$s_{km} &gt; s_{im}$ → update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>else → reset</td>
</tr>
<tr>
<td></td>
<td>$n \notin {i, k, m}$</td>
<td>if $s_{km} &gt; s_{im}$ and $s_{kn} &gt; s_{in}$ → update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $s_{km} &gt; s_{im}$ and $b_{kj} &gt; b_{ij}$ → update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $s_{kn} &gt; s_{in}$ and $s_{im} &gt; s_{km}$ → reset</td>
</tr>
<tr>
<td></td>
<td></td>
<td>none → if $s_{km} &gt; s_{im}$ → update</td>
</tr>
<tr>
<td>none</td>
<td>$i$</td>
<td>leave</td>
</tr>
<tr>
<td></td>
<td>$k$</td>
<td>update</td>
</tr>
<tr>
<td></td>
<td>$m \notin {i, k}$</td>
<td>if $s_{km} &gt; s_{im}$ → update</td>
</tr>
<tr>
<td></td>
<td></td>
<td>none → leave</td>
</tr>
</tbody>
</table>

Table 1: Rules table for CADE synchronous conflict resolution. The sender physical node is denoted with $k$, and the receiver physical node with $i$. The time vector $s$ represents the time stamp of the last information update from each of the other agents.
### Table 2: Rules table for CADE asynchronous conflict resolution.

The sender physical node is denoted with \( k \), and the receiver physical node with \( i \) (Table 1 of 2).
<table>
<thead>
<tr>
<th>InP process $k$ thinks $a_{kj}$ is</th>
<th>InP process $i$ thinks $a_{ij}$ is</th>
<th>Receiver’s action (default leave &amp; no broadcast)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$m \notin {i,k}$</td>
<td>$i$</td>
<td>if $b_{kj} &gt; b_{ij}$ → update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $b_{kj} = b_{ij}$ and $a_{kj} &lt; a_{ij}$ → update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $b_{kj} &lt; b_{ij}$ → update time and rebroadcast</td>
</tr>
<tr>
<td>$k$</td>
<td></td>
<td>if $b_{kj} &lt; b_{ij}$ → update and rebroadcast (sender info)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $t_{kj} &gt; t_{ij}$ → update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $t_{kj} &lt; t_{ij}$ → leave and rebroadcast</td>
</tr>
<tr>
<td>$n \notin {i,k,m}$</td>
<td></td>
<td>if $b_{kj} &gt; b_{ij}$ and $t_{kj} \geq t_{ij}$ → update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $b_{kj} &lt; b_{ij}$ and $t_{kj} &lt; t_{ij}$ → leave and rebroadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $b_{kj} &lt; b_{ij}$ and $t_{kj} &gt; t_{ij}$ → update and rebroadcast</td>
</tr>
<tr>
<td></td>
<td></td>
<td>if $b_{kj} &gt; b_{ij}$ and $t_{kj} &lt; t_{ij}$ → leave and rebroadcast</td>
</tr>
<tr>
<td>none</td>
<td></td>
<td>update and rebroadcast</td>
</tr>
<tr>
<td>$i$</td>
<td></td>
<td>leave and rebroadcast</td>
</tr>
<tr>
<td>$k$</td>
<td></td>
<td>update and rebroadcast</td>
</tr>
<tr>
<td>$m \notin {i,k}$</td>
<td></td>
<td>update and rebroadcast</td>
</tr>
<tr>
<td>none</td>
<td></td>
<td>leave and no rebroadcast</td>
</tr>
</tbody>
</table>

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

Table 3: Rules table for CADE asynchronous conflict resolution. The sender physical node is denoted with $k$, and the receiver physical node with $i$ (Table 2 of 2).