End to End Orchestration of Distributed Cloud Applications

by

Saeed Arezoumand

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Department of Electrical and Computer Engineering
University of Toronto

Abstract

End to End Orchestration of Distributed Cloud Applications

Saeed Arezoumand

Master of Applied Science

Department of Electrical and Computer Engineering

University of Toronto

2017

Centralized management provides benefits for cloud providers in terms of efficient and simple management of their infrastructure. However, tenants who use these infrastructures to deliver a software service to the end-users, are handicapped by having to work with traditional network primitives. Current service orchestration tools can automate most of the service configuration and deployment process, but these do not yet include significant SDN capabilities. In this thesis, we propose and examine high-level abstraction models for the orchestration of distributed cloud applications over multiple network domains and multiple infrastructure providers. We provide cloud application developers with a set of useful network functionalities that require no programming effort to provision and use. Our design relies on Hyperexchange, a protocol-agnostic exchange point for peering of virtual networks, to enable orchestration among multiple virtual network providers.

Acknowledgements

To my family: for your unsparing support and your continuous encouragement throughout my years of study. I owe it all to you.

To my supervisor, Alberto Leon-Garcia for his continuous guidance, motivation, and inspiration. I've been so blessed to have such a remarkable source of knowledge with extensive experience, and his great vision has always directed my research to the right way.

To Hadi Bannazadeh for always pushing me to work hard, to accomplish more, and to learn more. He's been an excellent source of invaluable technical expertise helped me on every occasion of my thesis project.

To all my fellow graduate students and researchers in the Network Architecture Lab for their feedback, cooperation and of course friendship. With a special mention to Thomas, Morteza, Kristina, Raj, Atoosa, Bahareh, Weiwei, Samira, and Dr. Park. Being with you guys was fantastic.

To Vladi, for his help and administrative support and of course for his great Italian espressos!

And to Ellie, my dearest!

Contents

List of Figures

Chapter 1

Introduction

The Internet initially evolved to deliver a specific task: universal reachability; and application developers have benefitted from this capability through a simple interface, that is End-to-End sockets. This model greatly simplifies the development of applications by abstracting every other detail about the network from the perspective of an application developer and led to the seminal client-server applications on which we now rely. To make the end-to-end packet delivery possible at a universal scale, the Internet started with a set of greedy architectural elements:

- At Layer-2, The main mechanism to avoid loops is Spanning Tree Protocol which leads to removing links and wasting capacity.
- The basic routing mechanism at Layer-3 for both Inter-domain and Intra-domain levels is the shortest-path algorithm.
- There is no resource management mechanism but a blind congestion control is exerted at the bottleneck links.
- There is no central access control and the Internet is fundamentally open. Every endpoint can send any amount of traffic to every other endpoint unless it is limited by the infrastructure.

Over the past three decades, to meet any new requirement, these initial Internet principles have been patched by introducing new protocols and middleboxes. These patches have made the Internet a diverse collection of artifacts based on a few greedy principles, which has become very complicated and difficult to change. Moreover, the ever-growing scale of the Internet, both horizontally and vertically, has clearly shown the severe inefficiencies of the current architecture. Due to the greedy path selection, 60% of the paths in the entire Internet violate the triangle inequality [10]. Network Service Providers typically over-provision the capacity of their network to virtually mask failures from their clients and links are typically provisioned to 30-40% average utilization [53]. In multi-domain scenarios, the blind end-to-end congestion control usually leaves significant spare capacity at the inter-connection points. A recent study on the interconnections points in the US has revealed that the aggregate peak utilization across interconnects is roughly 50% [35].

The rise of cloud computing and especially distributed computing has introduced a set of applications with new requirements that were never anticipated at the beginning of the Internet. These applications demand far more services than just merely delivering packets, but the Internet is incapable to adapt due to its ossified architecture.

Software defined Networking (SDN) promises to simplify network innovation by separating the control logic from the underlying switching elements and offering a programmable centralized view to the network engineers. Software Defined Infrastructure (SDI) takes a further step by offering programmable control over all the consumable resources in a cloud environment. Since the early rise of these concepts, some valuable achievements have been made in the enterprise networks towards increasing efficiency and reducing failure rates and thus lowering the operational costs. Companies like Google and Microsoft have almost doubled their WAN utilization by means of centralized control [51, 53, 69] and can avoid most of the common failures by automating the infrastructure management procedures [65].

Software Defined Infrastructures promise to redefine the network foundation of autonomous carrier networks and Internet Service Providers (ISPs) towards integrated and multi-tenant clouds offering programmable and fine-grained resources including but not limited to virtual networks. In this new model, the traditional role of ISPs can be segregated into two roles: Infrastructure Providers (InPs), who provide virtualizable network infrastructure and Service Providers (i.e. tenants) who use virtual networks to provide services for end-users [28].

1.1 Problem Statement

The paradigm of centralized management has delivered clear benefits for the cloud providers regarding the efficient and simple management of their infrastructure. However, tenants who use these infrastructures to deliver a software service to the end-users, are still given the traditional network primitives. Current service orchestration tools can automate most of the service configuration and deployment process but there is yet almost no influential presence of SDN capabilities in these tools. The following reasons can be pointed as the main roots of this limitation:

Additional Complexity: SDN can make the job of network engineers easier since they are the front-line of dealing with the complexities of network management. But for application software engineers who have been using networks through imperfect but simple end-to-end sockets, SDN brings yet another piece of complexity that often seems unnecessary to them. This view is supported by the five years of continuous development and operation of the SAVI Testbed [56], a nation-wide deployment of the SDI concept. It is observed over time that network programming is practically inconvenient and error-prone for most tenants, who are not generally familiar with SDN and networking details. Thus, realizing the capabilities of programmable networks [74] is not achievable, unless higher-level abstractions are provided that give tangible benefits specifically for cloud

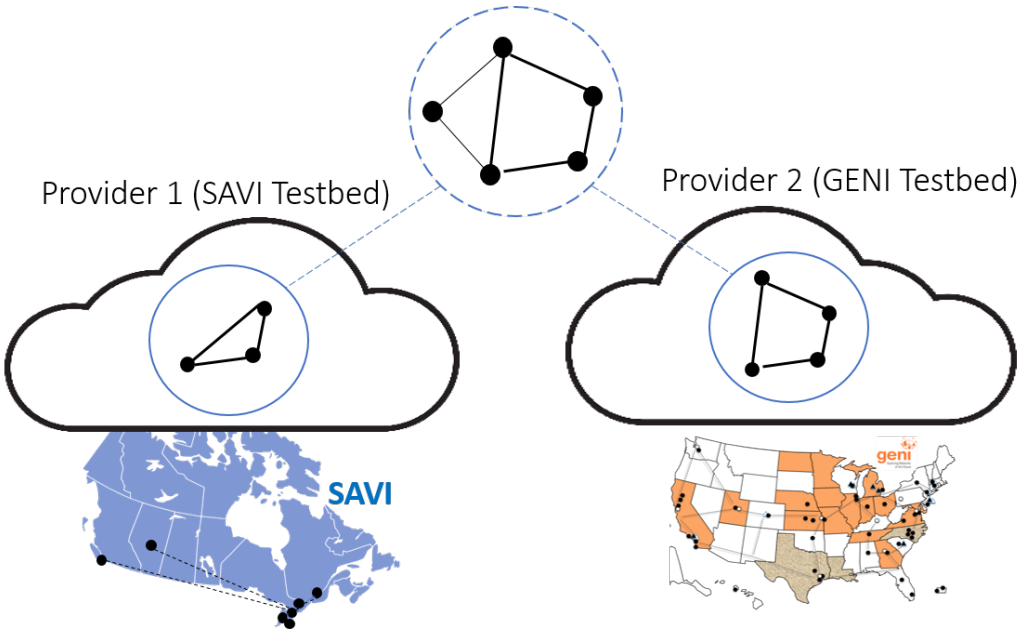


Figure 1.1: Peering of two virtual network from different SDI providers

application developers. It is worth mentioning that defining simple interfaces targeting common application developers is completely different than making new programming abstractions that target network engineers. Programming abstractions for SDN is extensively studied in the literature [38, 63, 83, 95, 108, 109, 112] while an abstraction model that provides easy-to-use network primitives for cloud users yet is missing.

Limited Scope: Even though some providers can give enhanced network capabilities to their tenants, still the common denominator of different infrastructure providers is not more than the very basic networking primitives that were traditionally available before the advent of SDN and SDI. Real world cloud applications often require getting deployed over multiple providers for better geographical coverage and to enable cost optimization strategies. In these scenarios, developers must forego those occasional SDN benefits given by only some of the providers.

Technology Diversity: Even in the cases where some advanced networking services are commonly offered by multiple providers, each provider may use different technology enablers to provision that specific service. As an example, Software Defined WANs are one of the common SDN-powered services offered by various infrastructure providers.

However, due to the diversity of the technologies being used by these providers for network slicing and virtualization, it is simply not possible to provision a SDWAN over multiple providers. As depicted in figure 1.1, both SAVI and GENI can provide Layer-2 SD-WANs for their tenants. In SAVI, MAC-based ACLs are used for network slicing and isolation among different tenants While GENI uses VLAN segmentation for network slicing and due to this difference, it is not possible to provision end to end Layer-2 WANs over both testbeds. In general, traffic exchange among multiple providers is not currently possible except for traditional IP networking on both sides and therefore, going beyond the scope of a single provider will limit tenants to the basic IP networking provided by the Internet.

1.2 Research Goals and Requirement Analysis

In this thesis, we propose and examine high-level abstraction models for the orchestration of distributed cloud applications over multiple network domains and multiple infrastructure providers. The following major steps are required to be taken towards this goal:

- A high-level representation is needed to precisely model distributed cloud applications and their networking requirements. The model should be expressive enough to encompass advanced networking features provided in an SDI. Also, it should be intuitive for the cloud application developers to easily provision and consume those advanced features using the proposed model without requiring them for any programming effort.

- A clear mapping is required from the proposed model to the detailed configuration and control rules in an SDI environment.

- The mapping procedure should be extended to support multi-domain and geographically distributed SDI deployments and to possibly include multiple autonomous providers.

We base the high-level application-oriented orchestration model on the Intent Driven Networking (IDN). IDN promises to automate network management procedure and provide a set of provisionable network configuration primitives. This model, which initially appeared in ONOS platform, hides the unnecessary details of the underlying infrastructure from users and allows them to customize network configuration using intents, without needing to program their network [54].

A proper IDN framework for multi-domain SDIs must address certain requirements that pertain to multi-tenant geo-distributed cloud environments:

Simple Abstraction: Simplicity is the central motivation for Intent Driven Networking. SDI users must be able to customize their network without having to be aware of the underlying network topology or the challenges related to programming a network controller. Intents should be abstracted as distributed and logical networking services between any set of resource endpoints. The framework must translate the intent setup requests to a set of low-level network control and configuration updates and enforce them upon the network topology.

Multi-domain Scale: The existing intent frameworks are not designed for multi-domain geographically-distributed SDN deployments (e.g. SAVI Testbed or Google B4 [53]). In these environments, each domain has an autonomous local controller to meet the control plane response time requirements in the local network. An intent framework for these environments must install and maintain end-to-end network intents over multiple domains and hence over multiple control platforms.

Data-path Performance: Due to data-path performance requirements, these configurations cannot be applied using encapsulated overlay tunnels over IP. For example, the SAVI Testbed is comprised of data-path elements with up to 10 or even 100 Gbps of bandwidth. Data-path performance of encapsulated overlay tunnels falls far below this requirement.

Tenant Isolation: Isolation across tenants is a crucial requirement in multi-tenant

environments. Therefore, the intent framework must avoid cross-contamination of intents requested by different tenants.

In order to extend our model to support multi-provider scenarios, providers with diverse technology enablers should get involved in an end-to-end setup. The need for interoperability among different providers requires versatile and extensible exchange points to interconnect autonomous SDIs. Such an exchange point should address the following requirements:

Protocol Agnostic Exchange Points: A central feature of SDI architecture and a Virtual Network Environment is the customizability and thus heterogeneity of network protocols and logic [28]. As a result, exchange points must be introduced to provide exchange services for different types of networks independent of the protocol being used.

Extensibility and Flexibility of Peering: To provide on-demand peering and traffic exchange services, exchange points must enable rich functionalities on network traffic that can range from a simple modification of header values to a complex, stateful Deep Packet Inspection system. The OpenFlow [80] protocol is a proper solution for fine-grained traffic forwarding. However, relying on a hardware switch as the only packet processing pipeline will narrow down the network functionalities of an exchange point to a limited set of header modifications. Thus, software-based packet processing frameworks, such as P4 [22] or DPDK [52], are needed to overcome OpenFlow limitations. To this end, the exchange point architecture must include processing resources in addition to pure networking resources.

Multi-tenancy: Policy enforcement at the exchange point should not be limited to the providers. Tenants (i.e. owners of VN's) should also be able to define their desired exchange policies so that end-to-end orchestration over VN's can be possible. An important requirement to realize this feature is a well-defined network flow space authorization at the exchange points that can isolate incoming or outgoing traffic of VN's from each other.

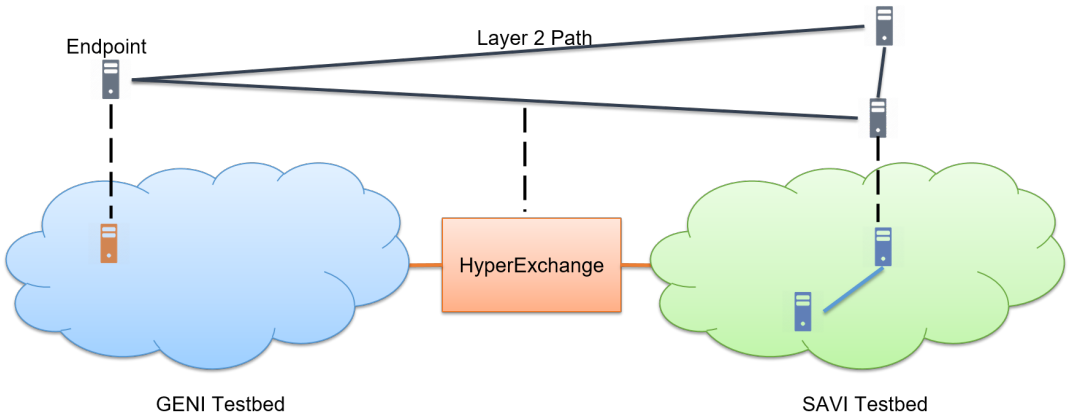


Figure 1.2: Use of HyperExchange for virtual network peering across different SDI providers

1.3 Thesis Overview

The next chapter gives a background of the foundation concepts including SDN, SDI, and it also discusses previous works regarding multi-domain orchestration and network management.

In Chapter 3 we begin with a straw-man architecture for multi-domain orchestration with idealistic assumptions. Based on the architecture we propose a graph-based abstraction model for user-defined intents with a generic intent compilation algorithm that can take provider intents into consideration. The chapter finishes by discussing the practical limitations of this initial proposal and the required changes.

In chapter 4 we present HyperExchange [12], a protocol-agnostic and software defined exchange fabric for peering of Infrastructure Providers and their hosted virtual networks. It provides inter-domain tenant authentication and authorization for network control and is architected as an autonomous SDI node that inter-connects participating networks. A novel data model is provided by HyperExchange to specify heterogeneous networks in a uniform way. Once networks are specified, a tenant can define policies to allow traffic exchange and enable peering of networks. This chapter continues by presenting a prototype implementation of HyperExchange that is deployed between SAVI and GENI testbeds (Figure 1.2). This prototype is an extension of the SDI-manager reference

model written in Python. In the current implementation, the user can specify a VN using a network specification API. The authorization module uses a private API to get VN attributes and to authorize the specification. The current implementation supports a combination of OpenFlow actions as the policies described by user. The authorization module uses remote APIs for network specification requests but for policy flow entries, it uses local specifications of network domains.

In chapter 5 we present MD-IDN [14], a framework for scalable multi-domain IDN. We extend the compilation algorithm presented in chapter 3 to achieve scalability in multi-domain networks: First, user-defined intents are processed over an abstracted multi-graph of network domains and their interconnections, and a set of local intents is then generated for each of the involved domains. Afterwards, the local intents will be compiled and installed in local regions in parallel. MD-IDN is deployed as a public service in the SAVI Testbed over more than ten data centers spanning across Canada. In multi-domain, large-scale environments, our experiments show that MD-IDN can improve intent compilation time by at least one order of magnitude.

Chapter 6 includes five use cases of networking features enabled by MD-IDN: SD-WAN, Distributed Virtual Router, Distributed Firewall, Tapping, and Service Function Chaining. It then continues with an evaluation of these use cases in the SAVI testbed.

We present the thesis conclusions in Chapter 7 and we identify the future directions of our research.

Chapter 2

Background and Related Work

In this chapter, we review the recent efforts towards programmable networks and infrastructure and discuss their real achievements. We start with an overview of Software Defined Networking, its main principles and its major use cases and achievements in the past few years. We then provide an overview of the concept of Software Defined Infrastructure and continue by providing architectural details about two of the major testbeds that have realized the SDI concept. In the last part, we will cover prior efforts for multi-domain network management and end-to-end orchestration.

2.1 Software Defined Networking

The core idea behind Software Defined Networking is breaking the vertical integration of control logic and data forwarding in network switches and routers. Through this separation between the control plane and data plane, network switches become simple forwarding elements that are controllable by a (logically) centralized remote controller (Figure 2.1). In an ideal SDN deployment, network engineers and operators would no longer be required to manually configure network devices using limited and vendor specific interfaces; instead we can write an application program that is integrated with the controller and installs the required flow entry rules in the switches. This transformation

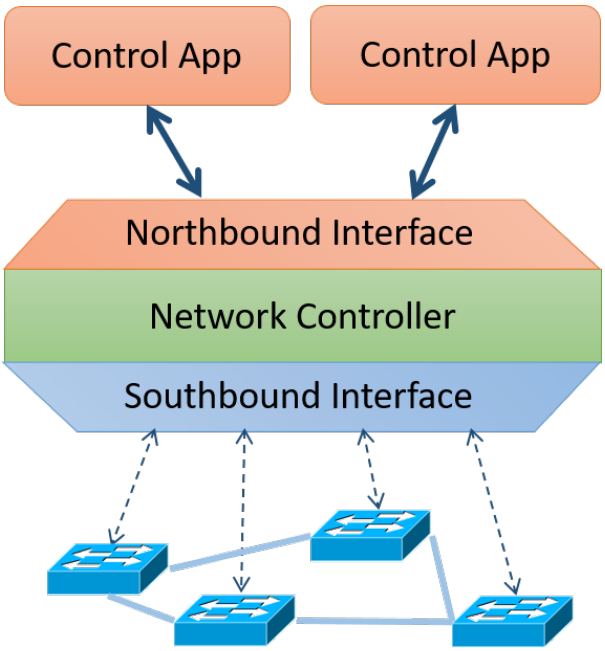


Figure 2.1: Conceptual Model of a Software Defined Network

simplifies the laborious task of network management to writing software programs to control the entire network, and as a result, it reduces the barrier to innovate in computer networks. Since the advent of SDN, its principles have been incorporated in many enterprise networks and led to better efficiency and simplicity in the operation of their underlying network [51, 53, 65, 69]. Among the many proposed control platforms for SDN, Open Networking Operating System (ONOS) [18], OpenDaylight (ODL) [82] and Ryu [99] are the major open source projects that are currently under continuous development and support. Ryu is a light-weight platform written in Python mostly being used for experimental purposes. ONOS and OpenDaylight on the other hand are enterprise-level controllers written in Java, and both are supported by the Linux Foundation.

2.1.1 Southbound Protocol

Currently, the OpenFlow protocol is the de facto choice for southbound interaction between the controller and the underlying switches. An OpenFlow switch has an OpenFlow agent that receives and interprets the OpenFlow messages coming from the controller,

and at least one flow table that stores flow entry rules. Each entry consists of matching criteria, an action list and a set of counters. Despite its vast integration in almost all the current Software Defined Networks and SDN controllers, OpenFlow has the following shortcomings:

- Even though OpenFlow is drafted up to version 1.6, current hardware switching fabrics mostly support only the initial version (e.g., 1.0) and partially cover some of the OpenFlow 1.3 features. Other versions are only supported in software switches (i.e., Open vSwitch - OVS).

- OpenFlow was initially designed for network control that includes setting the forwarding rules in the switches and changing the forwarding behavior of a pre-configured network. However, OpenFlow cannot modify the network configuration such as adding or removing ports and tables or any other device settings in the network. This limitation becomes more critical in virtual environments (i.e., cloud infrastructure providers) where configurations can change on-demand. To resolve this limitation, network controllers should support additional southbound protocols such as NetConf or OVSDB protocol.

- The set of available matching criteria and possible actions in addition to the multi-table capability in OpenFlow 1.1 and later, gives an incredible flexibility to alter the forwarding behavior of an SDN. However, OpenFlow is fundamentally incapable of sophisticated packet processing such as stateful forwarding or conditional pipelines and an alternative solution should be used regarding these requirements (e.g., P4 [22] or Click [64]).

2.1.2 Northbound Abstractions

SDN controllers provide northbound APIs for interaction with external softwares or functionality extensions by network engineers and developers. To simplify network application

development, many projects have tried to define intuitive and high-level domain specific programming languages for SDN [63, 95, 110]. Even though these proposals can greatly simplify network programming, none of them is fully adopted by the major open source SDN controllers.

2.1.3 Intent NBI

The concept of programmable network control has been recently directed towards the definition of high-level intent NBI [30, 107] and Intent-Driven Networking is recently gaining interest. All major SDN control platforms are now providing an intent Northbound Interface (NBI) as a high-level abstraction for network management. With these frameworks cloud users and network operators can conveniently define "what needs to be done", rather than "how it should be done". There are currently many efforts towards implementation and integration of Intent-based networking in SDN platforms [6–8].

2.2 Software Defined Infrastructures

As depicted in Figure 2.2 SDI builds upon both SDN and traditional cloud computing by providing an abstraction from the underlying infrastructure resources, exposing control over compute, network, storage, and other resources via a programmatic interface [59]. In this environment, computer networks can provide services beyond a simple packet delivery. Applications can be deployed dynamically across multi-region and geographically distributed clouds in which resources can be provisioned on-demand, chosen among a variety of types including compute, storage and networking.

SDI promises to redefine the foundation of carrier networks and Internet Service Providers (ISPs) towards integrated, multi-tenant clouds that offer programmable and fine-grained resources, including but not limited to virtual networks [58, 71, 86, 102]. In this new model, the traditional role of ISPs can be separated into two roles: Infrastructure

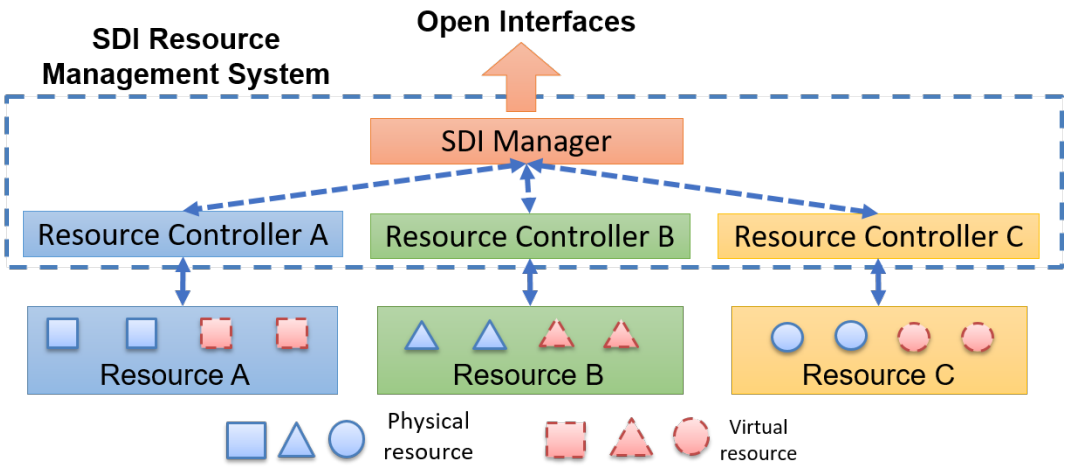


Figure 2.2: Conceptual Model of a Software Defined Infrastructure

Providers (InPs), that provide virtualizable network infrastructure; and Service Providers (i.e. tenants) that use virtual networks to provide services for end-users. There is now an opportunity for a Service Provider to deploy its Virtual Network over multiple InPs, thus achieving greater geographic coverage and creating new cost optimization strategies.

2.2.1 SAVI Testbed

The SAVI testbed is a multi-tenant Software-Defined Infrastructure (SDI) composed of heterogeneous resources including virtual machines, bare-metal, FPGA and GPU servers connected through virtual networks. Users manage their resource slices through programmatic interfaces (i.e., APIs). The testbed is comprised of twelve regions spanning from the Postech Edge in South Korea to several other university edges in Canada, more specifically two core data-centers located in Toronto and ten other university edge regions. Figure x displays the current geographical deployment of the SAVI testbed.

In each region of the testbed, the network data plane follows a simple leaf-spine topology enabled through physical and virtual OpenFlow supporting switches. However, data-path elements within different regions may vary based on the supported OpenFlow version ranging from 1.0 to 1.3, as well as based on the bandwidth capacity ranging

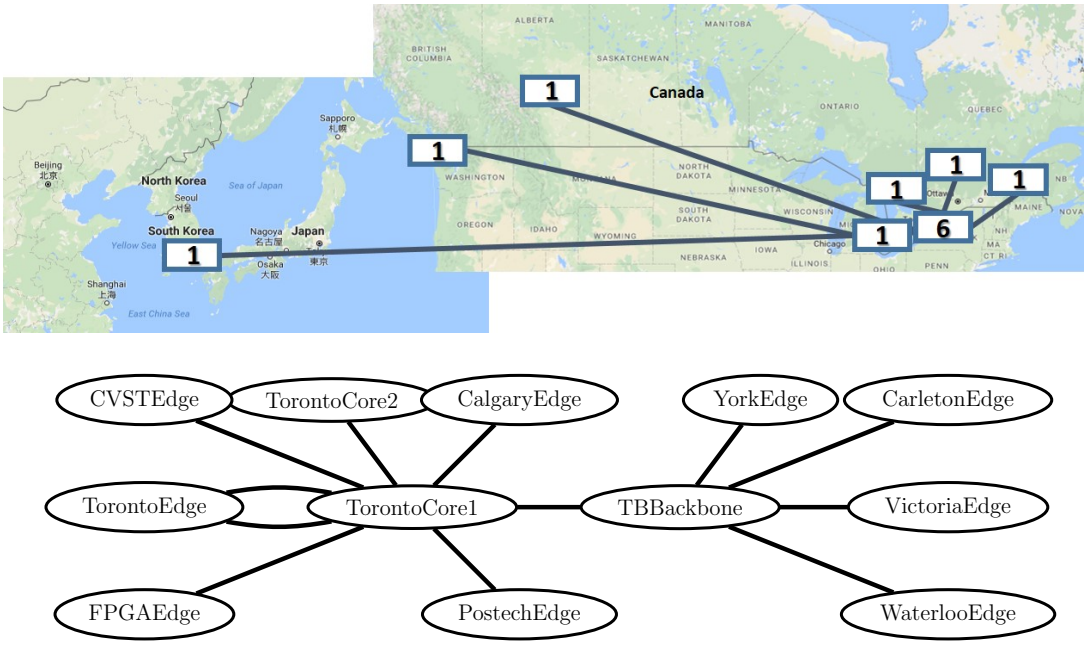


Figure 2.3: SAVI Testbed data centers as of June 2017

from 1 Gbps to 100 Gbps. Furthermore, among different regions, the data plane is interconnected through 1 or 10 Gbps dedicated links. This technological diversity and ranging capacities of the links and switches represents a natural outcome of evolution over time, and it is yet a challenge to be dealt with in practical scenarios.

The management plane in the SAVI Testbed is distributed with each region having its own autonomous SDI management system [87] that communicates only with a central identity management system. In each region, the SDI Management system controls the corresponding resource controllers to provide a compatible interface and configuration for each specific infrastructure resource. Figure 2.4 shows the architecture of a SAVI SDI node, which is designed for joint management of cloud and networking resources. The design of the SAVI node leverages the open-source cloud computing platform OpenStack [101], and the de facto standard SDN protocol OpenFlow [81]. While an out-of-the-box deployment of OpenStack only supports the virtualization of traditional computing resources (e.g., CPU, memory, and storage), in SAVI testbed it is extended to support unconventional resources including FPGAs and GPUs by adding new device drivers.

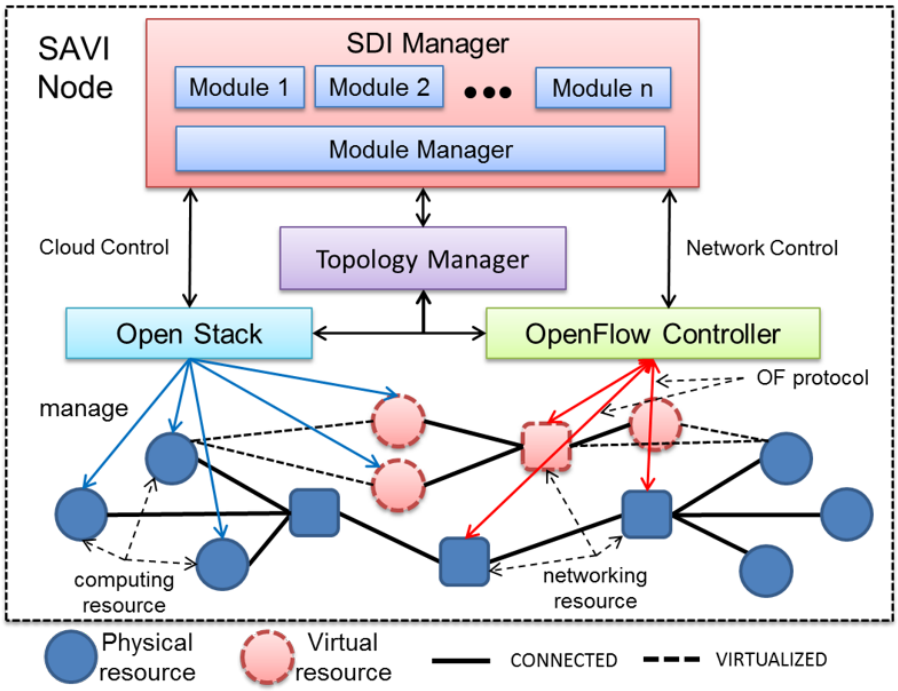


Figure 2.4: The Architecture of a SAVI Node

From a networking perspective, each region is an autonomous SDN domain with a separate controller. Having a dedicated SDN control platform for each domain is a typical design practice in large-scale and widely distributed SDN deployments. This practice allows the internal network control per domain to be independently working and the controller response time to intra-domain network events is minimized. In each domain, the controller is only aware of the internal topology and the interconnection points to the other domains.

Users can benefit from SDN capabilities by using user-flows in SAVI SDI. This feature allows users to enforce their forwarding policies to the endpoints that they own. By leveraging this API, users can perform RESTful requests to match a flow and perform a set of actions on it. User-defined SDN applications are enabled on SDI using the Network Control Module, a technology-agnostic SDN platform with access to infrastructure-wide information [74]. Tenants are defined in form of separate projects in SAVI SDI and users can participate in multiple projects. All the resources in a project including the

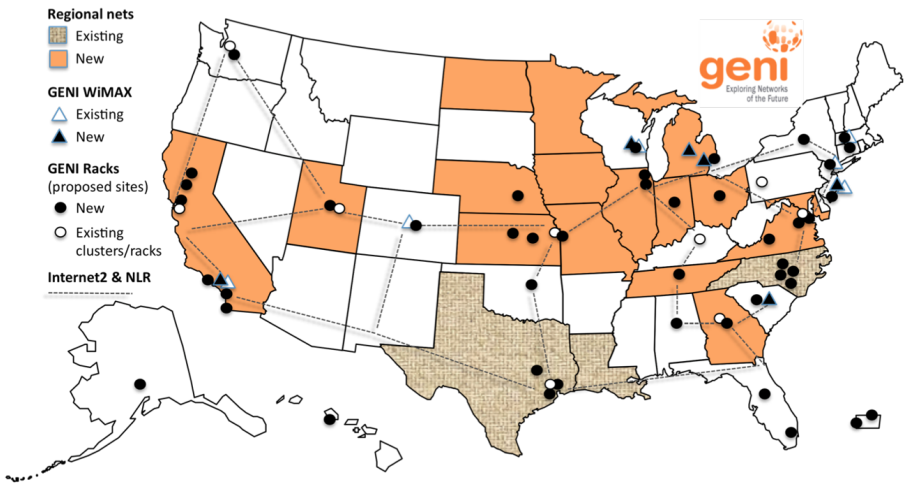


Figure 2.5: Map of the GENI Testbed [4]

networking resources are isolated and protected against other projects.

2.2.2 GENI Testbed

GENI, the Global Environment for Networking Innovation [20], is a distributed testbed sponsored by the U.S. National Science Foundation (NSF) for the development of future-oriented network experiments. The initial efforts for the testbed design goes back to 2005 and 2006, essentially before the rise of SDN. GENI is built on the federation of regional campus networks and racks across multiple sites and universities in the U.S. (Figure 2.5). It provides sliceable resources for experimenters to examine new network models and Internet architectures. Sliceability here means the ability to slice and virtualize a set of resources with a degree of isolation among different experimenters. The set of resources allocated for a specific experiment is called slice and each individual resource is an sliver.

Unlike in SAVI, layer-two networks in GENI are isolated using VLAN tags. To setup a layer-two connection between VMs, a user must define a topology in the form of an Rspec and pass it as a request to the Stitching tool. This tool, acting as an orchestrator, will then call Aggregate Managers involved in the topology to request the necessary resources, including VMs and VLAN tags.

2.3 Multi-Domain Orchestration

Real-world applications can involve multiple service providers due to the geographical distribution of the end users or to optimize the cost of service delivery. End-to-end orchestration is a challenge when orchestration has to be done among multiple cloud regions that possibly have different authorization domains and control logic. In this section, we will review the key technology enablers as well as the recent studies regarding multi-domain orchestration and network management.

2.3.1 Software Defined Exchange

The increasing prevalence of Intern Exchange Points (IXP) [26] has made the Internet topology denser and flatter [24, 33, 42, 72]. Traditional IXPs are comprised of a shared Layer-2 switching fabric that interconnects more than two autonomous systems [9]. Through this Layer-2 fabric, gateway routers of the participating networks can exchange BGP routes and normal traffic. In the initial deployments, different AS pairs had to separately initiate a BGP session to exchange their reachability information. A Central Route Server was proposed to improve the architecture by eliminating the need for AS-to-AS sessions [97]. In the new architecture, each participating network initiates only one session with the route server.

Software Defined Exchange points (SDX) are proposed to enable incremental innovation in the Internet inter-domain connections [37]. SDX replaces the central route server with an OpenFlow controller as well as the underlying switches with OpenFlow-enabled switches (Figure 2.6). The first SDX implementation was mainly able to alter the default BGP route by giving an illusion of a virtual OF switch to each participating network [48]. Later, the industrial scale SDX was introduced to improve scalability issues regarding the first SDX architecture [46]. After the first introduction of SDX, it has been studied in many other recent works [77, 105] with the goal of providing flexible inter-domain

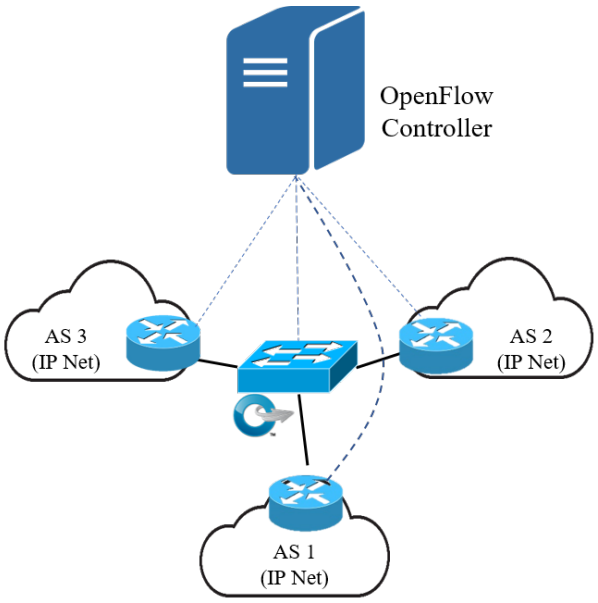


Figure 2.6: Conceptual model of a Software Defined Exchange Point

routing for carrier network operators.

2.3.2 End-to-End Routing

Given the capability of SDX, it is now possible to propose centralized and programmable approaches for inter-domain routing. In [68] Control Exchange Points (CXP) are used to stitch inter-domain paths over the IXP multigraph which is an abstraction of the Internet topology with IXPs as vertices and edges are the virtual links between IXPs over an AS (Figure 2.7). CXP provides more path-diversity compared to the default set of BGP-based paths which follows valley-free routing policies [39, 40].

There are also other works to centralized inter-domain routing using SDN [67] or via a small set of brokers [73]. In [17], the potential of multipath routing is studied for the case of centralized inter-domain routing. A case of multi-provider embedding of service function chains is studied in [34] which is mainly concerned with the optimization of allocation and placement. Also, an emulation of Intent-based management of multiple OpenFlow/SDN domains is proposed in [25].

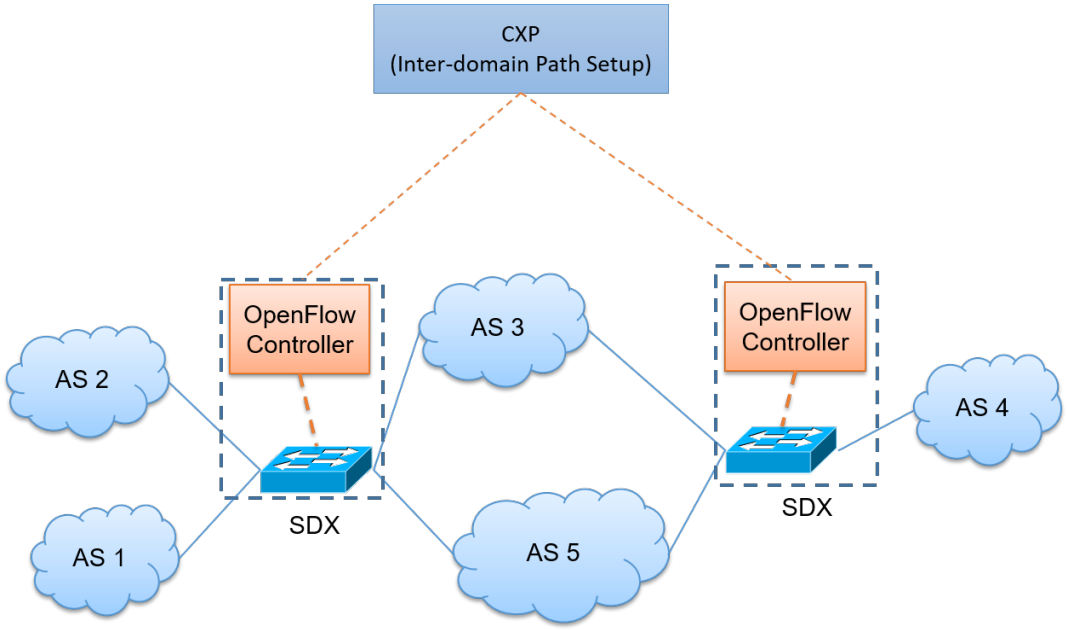


Figure 2.7: The role of Control Exchange Points to stitch inter-domain paths over the IXPs

2.3.3 Current Orchestration Platforms

Over the past few years, there have been several works around moving the architecture of legacy carrier networks towards integrated clouds and datacenters [58, 71, 86, 102] offering on-demand resource provisioning and automated orchestration. As a result of this evolution, cloud service orchestration has been studied in many recent works [16, 75, 92] as a challenge and yet a potential for future distributed applications. These works have essentially focused on orchestration regarding efficient placement or service modeling in a single cloud region

Cloud brokers are considered as a solution for orchestrating services over multiple service providers [89, 111]. However, this model does not necessarily address inter-domain authorization, peering and routing problems between multiple autonomous service providers.

There are enterprise orchestration tools such as Cloudify [1], ansible [50], JuJu [3], providing automation for deployment and configuration of a software stack. Also, Heat [2] is the main orchestration component in OpenStack. Docker Swarm [98] and Kuber-

netes [15] are container orchestration tool and could be used when the entire software stack is containerized as a group of micro services.

2.4 Remarks

Current IDN frameworks pose two main limitations that affect deployment in production grade and multi-domain networks. They are mainly concerned with a single network domain, and thus enabling end-to-end network intents over a multi-domain and large-scale setup is still a challenge. Furthermore, these frameworks do not consider any differentiation between user intents and provider intents, and only a limited set of intent classes are available for both.

Current SDX proposals mainly target the traditional all-IP participating networks and provide policy enforcement capability for the operators of the participating ASes. However, the ability to peer heterogeneous virtual networks with different protocol stacks is yet to be studied.

All of the current orchestration tools are mainly concerned about modeling and deployment of an application service and they rely on the legacy IP networking primitives. In multi-domain cases, GRE or VXLAN encapsulation are created over the underlying IP connections which inherits all the main limitations of IP networks.

Chapter 3

The Straw-man Proposal

Cloud service orchestration involves the process of provisioning resources as well as configuration and deployment of application services in an automated way. This process should ideally include monitoring and performing possible reconfiguration based on the real time measurement results. Our focus is essentially on the end-to-end network management of a distributed cloud application and the main goal is to provide multi-domain network orchestration for these applications.

3.1 Design principles

In this part we discuss two main principles followed to design an end-to-end orchestration platform as well as their justifications:

The first principle is basing the abstraction model on an Intent-based API rather than a fully programmable SDN. Software Defined Networking allows users to control the network using programmable control and configuration APIs. However, in multi-tenant environments providing users the ability to have full and direct access to the shared underlying network carries many potential issues and risks including the following:

- **Complexity:** Regular users with no networking expertise, find it inconvenient

to have to program the network controllers with low level abstractions such as the OpenFlow protocol. Instead, they should be given the ability to specify their networking requirements, without having to know or understand the underlying network topology or the technology-enablers for that matter.

- **Lack of Robustness:** If users are given the ability to define their own custom flow rules in the shared underlying network, chances of potential network failures would be increased. These failures could occur as a result of human errors or lack of networking expertise.

- **Provider Privacy:** In general, providers do not expose details of their internal topology or network configurations to their users. Reasons could involve both, business and security aspects. Intent Driven Networking can preserve users' ability to customize network behavior, while abstracting the details that providers wish to keep private.

- **Pricing Model:** Giving users the ability to define low level flow entries or configuration changes to the shared underlying networks makes it difficult or even impossible for providers to define a fine-grained pricing model for network infrastructure usage; since the logical affect of low level flow entries to the shared network is not always clear. Whereas, in Intent Driven Networks the provider can define a clear pricing model for each of the offered intent classes.

- **Control Authorization:** Authorizing the network control at the flow entry level is challenging and can incur false negatives or false positives. Whereas, in the Intent Driven Networking the authorization would be done at the logical level. This is supported by the fact that for each intent, the provider is already well aware of the logical behaviour, as opposed to the unknown behaviour that could be produced by a set of custom flow entries defined by a user.

- **Conflict Detection:** User defined flow entries have a great potential of causing network control conflicts, even in the scope the users' own network. In contrast, IDN enables providers to detect and resolve potential control conflicts at the logical level, in an abstracted way from the users perspective, which in turn will improve the overall productivity and simplicity of the network.

Therefore, Intent Framework as a high level abstraction for network management can eliminate those issues and risks, since it prevents the exposure of the underlying network to consumers. We proceed by elaborating on the reasons that gave precedence to the Intent Framework, as opposed to users having network programmability rights.

The other principle is following a hierarchical foundation to decouple the orchestration model from the underlying resource management procedures in individual domains. In the current SDN platforms, intents are implemented as a set of programming modules to automate the mapping between user inputs and the low-level flow rules in the network. This mapping procedure (i.e. intent compilation) may vary based on the control platform, infrastructure vendor and even the intent type within the same platform. As an analogy to this, consider having a dedicated compiler for each program which makes it difficult to improve or optimize the compilation process independent of the intent types and SDN frameworks. To overcome this issue, a hierarchical design as well as a uniform intent abstraction model are required to decouple the high-level intent-based model from the underlying control platforms.

3.2 Proposed Architecture

Figure X shows our initial design where the most bottom layer include SDI domain each having a controller instance. Each domain offers primitive resource types, specially for networking and centralized controller is responsible. We assume that all infrastructure providers participating in an end-to-end orchestration have the following basic capabili-

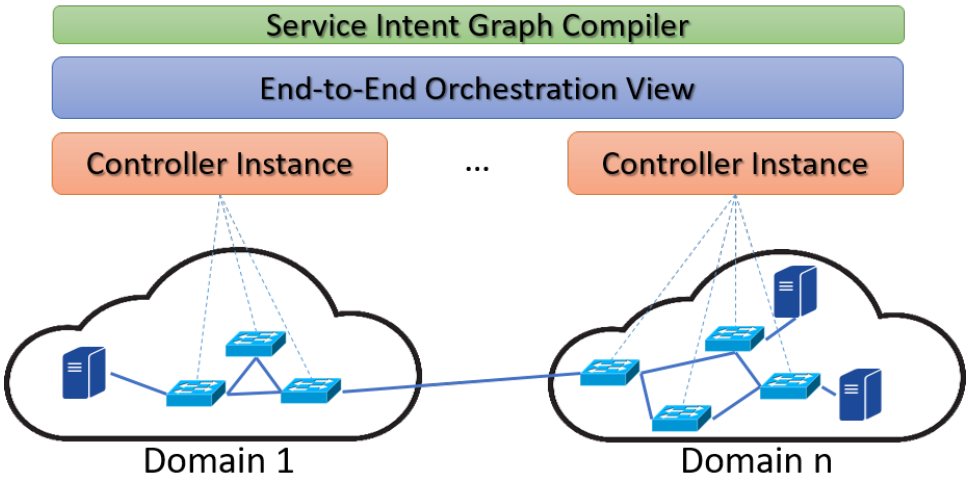


Figure 3.1: Abstraction Layers for End-to-End Orchestration.

ties:

Multi-tenancy: Multi-tenancy essentially involves isolation and protection of tenants resources (including physical and virtual resources) against each others. This means that tenants use of resources poses no threat to the provider or other tenants.

Self Service Capabilities: All resources and features offered by providers should need no manual intervention of the provider to provision or consume.

The set of controllers collaborate on forming the end-to-end global topology view and Network Information Base (NIB) which is the middle layer. Finally, above the global topology view, the orchestration platform can receive, compile and install application specific intent-model. The global orchestration platform relies on the ability to define custom flow rules in each domain through local controllers.

3.3 Intent Graph Abstraction Model

In this section we first highlight the natural differences between user and provider intents in a multi-tenant environment and mention the requirements of both sides. Based on the user requirements, we continue by defining a uniform graph-based abstraction model as a precise description of intents. As a result of this uniform foundation, a generic

compilation process for user intents is provided, which can consider provider intents as well. This uniformity allows us to decouple the intent compilation and installation process apart from the intent type and network controller platform. In a multi-tenant cloud environment, there is a fundamental difference between providers' intents and customers' intents. Providers are more concerned with the infrastructure and the effective usage of resources based on the overall user demands, whereas users are not aware about the infrastructure details or the underlying topology details. Using a networking analogy, users are only aware about their own endpoints and the end-to-end networking services among them, while a provider deals with routing and optimal resource usage decisions. Considering this difference, a proper orchestration framework should directly target users' demands with a simple and intuitive interface, while also being able to consider providers' policies concerning infrastructure usage, topology views and path selection.

3.3.1 Application Intent Graph

To build upon a uniform foundation, a simplistic graph-based representation for network intents is proposed. This abstraction model is application-centric, meaning that it is based on application requirements over the endpoints. There are mainly two types of generic network intents regarding the endpoints in the network: Forwarding and Blocking. Forwarding can be expressed in the form of $n_1 \overset{f}{\rightarrow} n_2$, where the edge label f defines the traffic type. Additionally, a blocking intent can be defined as $n_1 \overset{f}{\not\rightarrow} n_2$. As an example, to specify a forwarding intent for layer 2 traffic, the intent can be defined as $n_1 \overset{eth}{\rightarrow} n_2$. Also, to block layer 3 communication from one node to the other, the intent can be specified in the form of $n_1 \overset{ip}{\not\rightarrow} n_2$.

An intent graph is a collection of generic network intents in the form of a directed

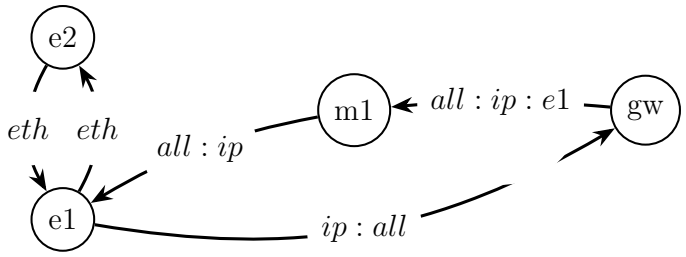


Figure 3.2: Sample policy graph. e1 and e2 are endpoints (e.g. VM), m1 is a middlebox and gw is the Internet gateway.

graph representing the desired policies between endpoints, without any involvement of the underlying topology. A sample intent graph is shown in Figure 3.2, where the intents include a bi-directional layer 2 communication among e1 and e2, a one way layer 3 connection from e1 to the gateway, and the returning path from the gateway to e1 redirected through a middlebox. In order to define intent classes, a mapping function is needed to map the inputs of that specific intent class to an adequate intent graph. Once the mapping is done, the compilation, installation and the life cycle follow a uniform process. This design enables adding new intent classes as simple as writing the mapping function only. In contrast to this uniform design, within current intent-based frameworks each new intent class should have a specific compiler.

3.3.2 Intent Graph Compilation

Each request to the orchestration framework will be submitted to the compilation process in the form of an intent graph. This process is outlined in Algorithm 1 and it involves three main steps:

Authorization: Upon receiving a new intent graph as an input, an examination based on the endpoint ownership information supplied by a reference monitor is completed. Since the intent graph only includes the involved endpoints and abstracts away the topology, the authorization process becomes quite straightforward. For each policy edge, the source and destination endpoints must be checked against the user's ownership

Algorithm 1: Policy graph compilation

```
CompilePolicyGraph (G_P, G_T)
    inputs : A policy graph G_P = (V_P, E_P); network topology graph
             G_T = (V_T, E_T)
    output: List of flow entries denoted by F
    F ← ∅;
    assert authorization
    assert feasibility
    foreach policy edge e ∈ E_P do
        if e.type is FORWARD then
            path ← getPath(G_T, e.src, e.dst);
            foreach hop h = (in_port, dpid, out_port) ∈ path do
                f ← forwarding_rule( e.traffic_type, e.src, e.dst)
                F.append(f)
        else if e.type is BLOCK then
            s ← getEdgeSwitch(G_T, e.src)
            f ← blocking_rule( e.traffic_type, e.src, e.dst)
            F.append(f)
    return F;
```

and privileges. In current practices [44] where tenants have to directly define low-level flow entries to control their networks, every new flow-entry in every switch along the path has to be authorized. This enforces a considerable amount of unnecessary workload and complexity to the authorization process. In contrast, using a intent graph abstraction, the authorization process is decoupled from the network topology and flow-entries and instead it is precisely focused on the end-points.

Feasibility Check: In the second step of the compilation process, the input intent graph gets checked by the existing intent graph for the user to ensure that the new setup does not have any conflict with the previously installed intents. In case of conflict, the user has the ability to force the intent installation.

Flow Entry Generation: After the authorization and feasibility check, the main compilation process starts. During this process, for each policy edge in the intent graph, the compiler inquires for a path between the source and destination nodes from the global topology and generates flow-entries based on the traffic type specified on the policy edge.

3.4 Discussion and Remarks

The proposed architecture can easily be emulated using a distributed SDN control platform such as ONOS. However, some of the assumptions in this design are too idealistic to be realizable in a practical scenario. We argue about these limitations in two scenarios: Multiple domains under one administration and multiple domains under multiple providers.

In case of multi-domain with single administration, the following reasons can justify the unsuitability of the intent graph compilation and installation over a global graph:

- **Scalability and Geographical Distribution:** In multi-domain environments two common features are the incremental behaviour in terms of the network size and the geographical dispersion of the network itself. Intent graph compilation over a global topology will not be able to scale appropriately in multi-domain and geographically distributed networks. Use of distributed controllers can improve scalability and fault tolerance as long as different instances have a reasonable geographical distance and round trip time for communication. For instance, in the SAVI Testbed if we consider only Victoria, Toronto and Korea regions, it is not technically reasonable or even required to share the topology information and network control state. The global topology in a multi-region deployment can easily grow to thousands of nodes and any intent compilation process over such a huge graph is not practically appropriate.

- **Technology Variation:** The technical architecture of the internal network control is usually diversified among different network domains due to the evolution over time. For instance, in the SAVI Testbed, there are four regions that are using the latest OpenStack version, whereas the rest of them use a previous version. The same occurrence can be noted within the Geni Testbed, where ExoGeni uses OpenStack, while ProtoGeni does not use OpenStack. Therefore having a set of

controllers following the same technology and version is not the case in real SDI deployments and it is technically challenging to have a uniform global topology over all of the regions.

In case of multi-domain with multiple administration (multi-provider) in addition to the above issues, there may be the following challenges:

- **Provider Privacy:** Different providers would not publicly share the topology and network information with each other. Therefore, a multi-domain intent framework should work with a minimal set of information provided by each domain not the entire topology.

- **Inter-domain Network Slicing:** Network slicing is a critical task in a multi-tenant environment to keep virtual networks isolated against each other. However, different providers would not use the same slicing criteria and thus it is not possible to simply inter-connect SDI networks. Consider a provider which uses VLAN isolation for network slicing and enforces security check only on the VLAN tags. The other provider may use MPLS or MAC-based isolation and be only sensitive about MAC addresses of the frames. The networks of these two provider cannot be directly peered.

To address these limitations, we first introduce HyperExchange as an exchange point between SDI providers with dissimilar architectures in the next chapter. Furthermore, To overcome the issues raised for scalability in multi-domain environments we present MD-IDN in Chapter 5.

Chapter 4

HyperExhcange

Current inter-domain networking has evolved based on a standard structure: A group of Autonomous Systems (AS) using IP as the internal network structure while relying on BGP for inter-domain peering. This structure made inter-networking possible in the first place, but afterwards it led to a notorious ossification. The problem, more precisely, is rooted in two major sources: First, the existing all-IP foundation of ASs forces a fixed addressing scheme which is location-based and assigned through standard address registries. Moreover, routing based solely on destination IP prefixes results in a severe control inflexibility in internal operation of ASs.

The other major challenge of the current Internet arises where different ASs interconnect with each other at exchange points. Use of BGP as the basis of inter-domain networking has caused a set of unresolved issues [23] including:

- Difficult and painful troubleshooting and security maintenance,
- Large convergence times,
- Anomalies caused by possible routing inconsistencies,
- Adversity of QoS policy expression and enforcement,
- Unoptimized end-to-end paths due to triangle inequality violations.

While these challenges can be traced back to the technology limitations in the early years of the Internet, recent advancements in SDN and NFV have provided new capabilities to motivate a reconsideration of inter-domain networking beyond its conventional restrictions. Along these lines, two major trends are ongoing in the research community and industry. In one direction, Software Defined Exchanges (SDX) [36] have been introduced to make Internet Exchange Points (IXP) [9] more flexible and to facilitate inter-domain routing while keeping ASs with the traditional all-IP structure. In the other direction, several attempts has been made to redefine the network foundation of autonomous carrier networks and Internet Service Providers (ISPs) towards integrated and multi-tenant clouds and datacenters offering programmable and fine-grained virtual networks [57] [70] [85]. In this new model, the traditional role of ISPs will be segregated into two roles: Infrastructure Providers (InPs), who provide virtualizable network infrastructure and Service Providers (i.e. tenants) who use virtual networks to provide services for end-users [28]. SAVI [55] and GENI [19] testbeds are two real deployments of such InPs. However, flexible peering of InPs and their hosted VNs has remained a challenge.

To address the aforementioned trends, we have introduced the concept of HyperExchange [12] as an exchange fabric for InP's and their hosted VN's. HyperExchange, in summary, provides the following particular contributions:

- A formal model that redefines the conventional concepts of network domain and sub-nets and a uniform data-model for networks and their sub-nets. We have extended the model to formally specify the pipeline of HyperExchange.

- An extensible design for the data-plane pipeline that leverages OpenFlow and custom VNF's to provide arbitrary packet processing capabilities.

- A control-plane design based on SDI model to provide multi-tenancy and to enable tenants of InPs to enforce network control policies on their own slice of traffic.

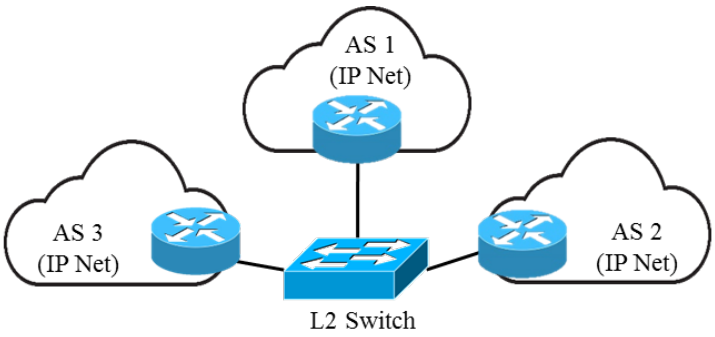


Figure 4.1: A conventional IXP peering three AS's

4.1 Formal Specifications

The network model at an exchange point specifies a clear semantic to bind slices of traffic (incoming or outgoing) to each participating network. This binding is central to define any traffic management in the exchange point. Figure 4.1 demonstrates a sample IXP and three participating ASs. In this case the IXP is modeled as a Layer 2 switching fabric and each participating network is connected to the IXP via a physical connection. Since all of the participating networks are IP networks, the mapping between networks and traffic at IXP is defined by source and destination IP addresses. This model forms a simple two-dimensional flow space in which source and destination IP addresses are dimensions. Figure 4.2 represents the slice of traffic coming from AS1 and going to AS2. While this model greatly simplifies the traffic slicing in IXP, it enforces two main limitations for inter-networking. First, only public IP networks can participate in an exchange point and second, forwarding logic at IXP is mainly defined by IP prefixes. Neither conventional IXPs nor current implementations of SDXs have targeted exchange of traffic between VNs with customized network protocols. Depending on vendor specific features, some exchange points may offer limited support of non-IP protocols but no exchange point have been introduced with a protocol-agnostic model that satisfies VN peering requirements mentioned in the previous section.

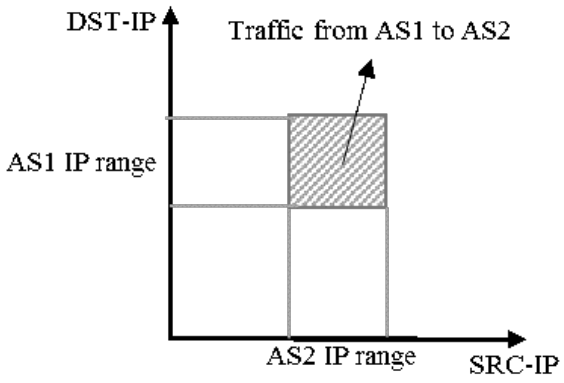


Figure 4.2: Two-dimensional flow space of current exchange points

4.1.1 Geometric Model

The network model of HyperExchange is built on a geometric model which is inspired by HSA (Header Space Analysis) [62] with some modifications. The perspective in HSA is a network with a set of switching boxes and their interconnecting links; while HyperExchange model is defined for traffic at a single point (exchange point) that interconnects arbitrary InPs and VNs. As a result, topologies or protocols of participating networks are not important and a network at exchange point is specified by the set of packets that belongs to it. In this model, each packet is considered as a point in a geometric space. Using this formalism, a clear specification for networks and subnets are provided. The model is then extended to define InP and tenant control policies and the main pipeline of HyperExchange.

Basic concepts

We Start with a brief introduction of the basic spaces and concepts of our model.

Total Header Space: A header space of H with length of L can be represented by $\{1, 0\}^L$ [62]. For example, the header space of IP address header can be defined by $\{1, 0\}^{32}$. A header field can be any bit sequence in the packet including the content. Consider the set of headers $\{h_1, \ldots h_n\}$ with length of $\{L_1, \ldots L_n\}$ as list of interested

fields; the total header space is the cross product of all header spaces defined by this list:

$$\mathbb{H} = \times_{i=1}^{n} \{0,1\}^{L_i} \tag{4.1}$$

Port Space: A space represented by the total set of interconnecting ports of exchange point. These ports can be physical or logical ports. In order to preserve the generality of the model, a packet drop can be modeled by sending the packet to a logical port associated with drop.

$$\mathbb{P} = \{P_1, \ldots, P_m\} \tag{4.2}$$

Flow Space: The flow space is the cross product of $\mathbb{H}$ and $\mathbb{P}$. Each packet in an exchange point belongs to two networks: source and the destination. To solve this conflict, we exclude incoming flow space and outgoing flow space at exchange point. To distinguish these spaces, we use a single bit binary vector. Thus the total flow space of HyperExchange will be:

$$\mathbb{F} = \mathbb{H} \times \mathbb{P} \times \{0,1\}^1 \tag{4.3}$$

Match Expression: A Boolean expression defined over header values and/or port numbers. For those headers that have source and destination values such as IP and MAC, the source value will be matched in incoming space while the destination value will be used to match in outgoing space. The same principle is applied for incoming and outgoing port numbers. Each match expression is associated with a region in the flow space which is the basic building block of flows. A flow is represented by "F" and formally is a subset of the flow space.

Filter Function: Given a set of packets, a Match Expression M, returns a subset of packets which constitutes the flow defined by M. The behavior of the Filter Function for

a single packet is:

$$\Psi_M(\{pkt\}) = \begin{cases} \{pkt\} \ if \ M \ holds \ for \ pkt \\ \phi \qquad\quad otherwise; \end{cases} \tag{4.4}$$

The output of the Filter Function on the entire Flow Space is the flow associated with M, that is the region defined by M in $\mathbb{F}$.

$$F_M = \Psi_M(\mathbb{F}) \tag{4.5}$$

The next rules show how the Filter Function is expanded by logical expressions on M. The formal proof of the following rules follows from the definition of filter function and due to the space limitation we simply state them.

$$\Psi_{M_1 \wedge M_2}(F) = \Psi_{M_1}(F) \cap \Psi_{M_2}(F) \tag{4.6}$$

$$\Psi_{M_1 \vee M_2}(F) = \Psi_{M_1}(F) \cup \Psi_{M_2}(F) \tag{4.7}$$

$$\Psi_{M_1} \mathrm{o}\ \Psi_{M_1}(\mathrm{F}) = \Psi_{M_1}(\Psi_{M_2}(F)) = \Psi_{M_1 \wedge M_2}(F) \tag{4.8}$$

Network Space: A region in $\mathbb{F}$ that binds all packets coming from or going to a participating network and is represented by F_N. The filter function $\Psi_M(\mathbb{F}) = F_N$ that filters all packets in F_N is called the binding function of N and M is called the binder expression of N.

Subnet: Network N_1 is a subnet of Network N_2 if and only if the space of N_1 is a subset of the space of N_2:

$$F_{N_1} \subseteq F_{N_2} \iff N_1 \ll N_2 \tag{4.9}$$

Note that "$\ll$" denotes subnet relation.

Control Policies and Pipeline

In HSA [62] any packet traversal through networking boxes is modeled as transformation over the flow space. We use the same notion to model control policies defined by InPs or VN owners at HyperExchange. A policy in general can be a sequence of OpenFlow actions or steering traffic through a custom VNFs. An OpenFlow header modification transforms the packet in $\mathbb{H}$ while dropping the packet or sending it out of a port is transformation in $\mathbb{P}$. Note that a packet drop is modeled by assigning a logical port to the packet. Based on this notion a control policy can be expressed as chain of transformation functions in $\mathbb{F}$:

$$P(F) = T_1(\dots T_n(F)) = T_1 o \ \dots o\ T_n(F) \tag{4.10}$$

This chain can include filter function as well to apply the policy on a specific slice of traffic. Consider the traffic coming from VN1 in InP1 and going VN2 in InP2 and consider M1.1, M1, M2.2 and M2 as the binder expression of these networks respectively; then the incoming pipeline can be modeled as follows:

$$\rho_{in} = P_{VN1}\ o\ \Psi_{M1.1}\ o\ P_{InP1}\ o\ \Psi_{M1}(\mathbb{F}) \tag{4.11}$$

And the final outgoing traffic will be:

$$\rho_{out} = P_{InP2}\ o\ \Psi_{M2}\ o\ P_{VN2}\ o\ \Psi_{M2.2}(\rho_{in}(\mathbb{F})) \tag{4.12}$$

Policy Authorization

Since a policy is modeled as a transformation in flow space, a policy authorization indicates the set of allowed transformations. It is assumed that all networks without subnet relation are isolated:

$$F_{N_1} \not\subset F_{N_2} \Rightarrow\ F_{N_1} \cap F_{N_2} = \emptyset \tag{4.13}$$

A transformation is allowed if it keeps the packet in the same network space.

$$P(\mathbb{F}) : \begin{cases} F_N \rightarrow F_N \Rightarrow Allowed \\ otherwise \Rightarrow Not\ Allowed \end{cases} \tag{4.14}$$

Transformation along network spaces can be allowed if the principal of the policy has ownership over both networks.

This formal model helps us to present a precise and general definition of VNs. It abstracts away the internal protocols and topologies of participating networks and hence, it is the basis of a pipeline design and can even be extended to include ICNs and name-based routing. We also used the model to drive a logic for authorization. The notion of binding helps us to guarantee the isolation of policies and can be extended to include custom ABAC authorization policies.

4.1.2 Network Specification Data Model

We have defined a data model for network specification at exchange point based on the geometric model described. For simplicity, one level of network hierarchy is considered in this model. Thus, at the top level we have InP networks and at the next level VNs can be defined as subnets of InPs. The data model is JSON structured with following main values:

Network ID: A unique identifier for the network.

Network Domain: If this network is a VN, network ID of InP will be specified as the domain. Network domain will not be specified for InP networks.

Binder Expression: A match expression that filters all packets of this network at exchange point. The binder is a list of lists modeling the expression in form of "sum of product" (i.e. OR of ANDs).

Metadata: A set of key values that describe additional attributes of the network.

The representation of this data model for service provider 2 in Figure 4.3 is:

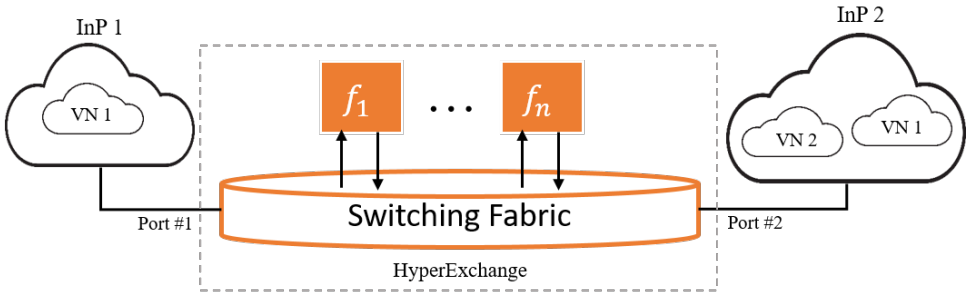


Figure 4.3: Conceptual representation of HyperExchange

```
{ net_id: InP2,
      net_domain: None,
      binder:{{port:2}},
      metadata: {}
}
```

Consider VN2 in the Figure 4.3 as private IP network with address range of 192.168.0.0/16. Then network data structure for this network will be:

```
{ net_id: VN2,
      Net_domain: InP2,
      binder: {{ip:"192.168.0.0/16"}}
      metadata: {}
}
```

4.2 Data-Path Design

HyperExchange is architectured with a three-layer data plane that is inspired by our SDI reference model. The bottom-most layer is the hardware switching fabric connected to a set of server racks on top. Each rack hosts a software switch (i.e. OVS) and an OpenStack [100] agent on top of that. The set of Open vSwitches forms the second layer of a data plane which is the software switching fabric. These software switches

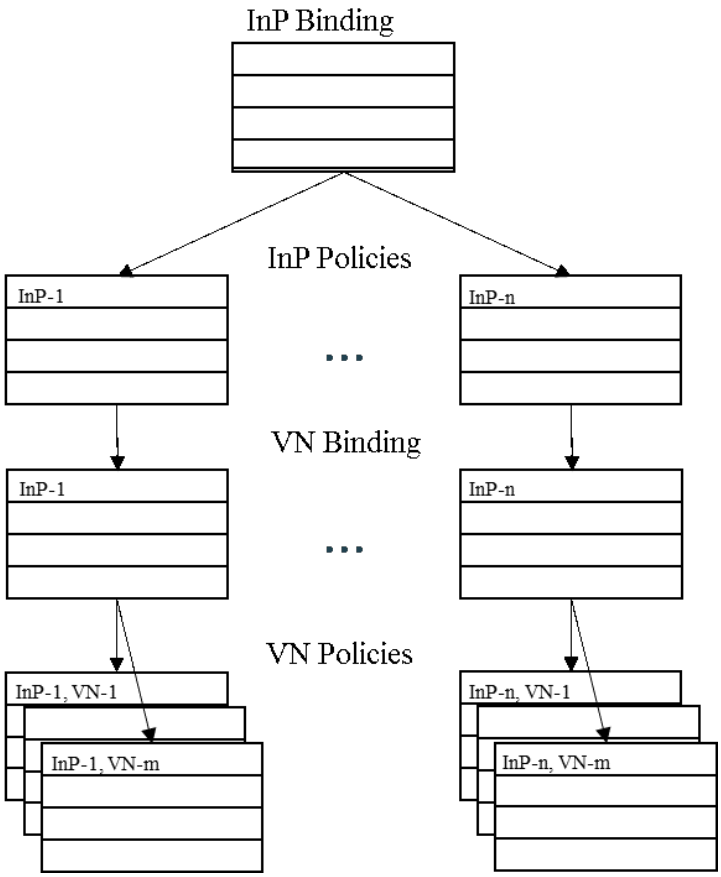


Figure 4.4: Main steps of the switching pipeline

are mainly used to establish steering circuits through Virtual Network Functions (VNF) and also as secondary flow store for swapping flow tables with hardware switches. The upper-most layer is the VNF-plane which is a set of Virtual Machines (VMs) hosting standard (validated) software network functions. VMs in each agent are connected to the OVS through Virtual Ethernets (Veth pairs).

4.2.1 Traffic Switching Pipeline

A packet processing pipeline is designed to realize the concept of formal model described in the previous section. To address the exclusion of incoming and outgoing flow spaces, the pipeline is designed with two separate phases. Since the outgoing phase has technically the same steps (in reverse direction) as the incoming phase, we only describe the

incoming phase and we refer to the incoming phase as the pipeline.

The pipeline should enforce policies for InPs and their tenants separately. A logical priority is considered for InPs over their hosted VNs and the control policies of the provider should be applied on the traffic prior to applying tenant policies. To make a clear separation between packets from different participating networks (including provider network and tenant network) our design has benefited from the multi-table feature in OpenFlow 1.3. A separate flow table is used for each of the participating networks.

Figure 4 demonstrates the overall life cycle of a packet in the pipeline which has the following four main steps.

Traffic Binding to InP Networks

Packets coming to the exchange point will be matched by the binder of InP networks at the very first step. This is the technical realization of binder function described in previous section, using OpenFlow match. the binding process is actually done by matching all packets by a flow-table containing binder flow-entries of all InP networks. In case of match, the packet will be sent to the policy table of the matched InP. A packet in this step must be bound to at most one InP network (is const). If a packet does not match to any InP network binder, it will be dropped.

InP Policy Enforcement

There is a dedicated flow table per each service provider. Filters and actions defined by an InP will be stored as flow entries in its dedicated policy table. It is common that a packet does not match to any entry in the policy table of InP; that means the InP has not defined any policy. In this case, the packet will be sent directly to next table in the pipeline line. This can simply be done by defining flow entry with the least priority to match on all packets and send them to the next table as the action.

Traffic Binding to VNs

VN network binding is similar to InP binding. There is a separate tenant binding (i.e. a flow table for tenant network binding) dedicated to each InP. There is a flow table for tenant network binding per each service provider. After enforcement of policies defined by associated InP, packets will be matched by the VN binding table of the same InP. Once the packet is matched to a VN in the InP, it will be sent to the policy table of that VN.

Tenant Policy Enforcement

Similar to InP policy tables, there is a dedicated table for each tenant of each InP. These tables contain flow-entries associated with the policies defined for each VN. In contrast with InP policy tables, a packet cannot continue the pipeline without matching any entry in the VN policy table and in that case the packet will be dropped.

4.2.2 Design Challenges

Here we mention and address two important challenges of the design described above, in terms of scalability and feasibility.

Virtual Flow Tables

The primary approach of dedicating a flow table for each InP network as well as each VN network can cause a severe growth in the number of flow tables needed. Due to the limited number of flow tables supported in a real OpenFlow switch, this can cause an important scalability problem for our design. To address this issue we introduced a novel approach called Virtual Flow Table (VFT) with which we can store multiple flow tables in a single real flow table. Metadata in OpenFlow 1.3 is used to realize this. Each VFT is assigned with a Virtual Table ID (VTID). In the primary case once a packet matches to a network binder the packet will be sent to the table associated with that network.

In this case, once the packet is matched in the binding table VTID of the designated network will be set as the Metadata and packet will be sent to the next table which stores all tables of a group (for example policy tables of all service provider networks). In this table, all flow entries of a Virtual Table have metadata = VTID as an additional match criterion. As an example, consider a table containing all of the policies of all InPs. The policies of each InP will be differentiated by an extra match criterion that is VTID of that InP. By using this approach, the total number of flow tables can be greatly decreased to a fixed number.

Circuit Switched VNFs

HyperExchange allows users to steer traffic through Virtual Network Functions (VNF). However, the steering mechanism is a challenge in this design. OpenFlow logical ports can be used to establish a dedicated circuit to each middlebox. This will allow packets to be forwarded only based on incoming port in a service chain. A logical port is a standard type of OpenFlow ports that can be used to create logical links. The concept of logical ports in OpenFlow protocol is neutral to the technology being used to realize it. For example, VXLAN tunnels can be used to establish logical links. If a user specifies VNF steering in the control policy, the SDI manager (described in next section) will create a dedicated logical port in the main switch and stablish a tunnel between the logical port and the VM running the VNF. This design abstracts away the topology details of HyperExchange for users.

4.3 Prototype Implementation

We have implemented a prototype of HyperExchange to show the feasibility of our modeling and design. The prototype is an extension of our SDI-manager reference model called Janus written in Python. The control plane of HyperExchange is an extension of our ref-

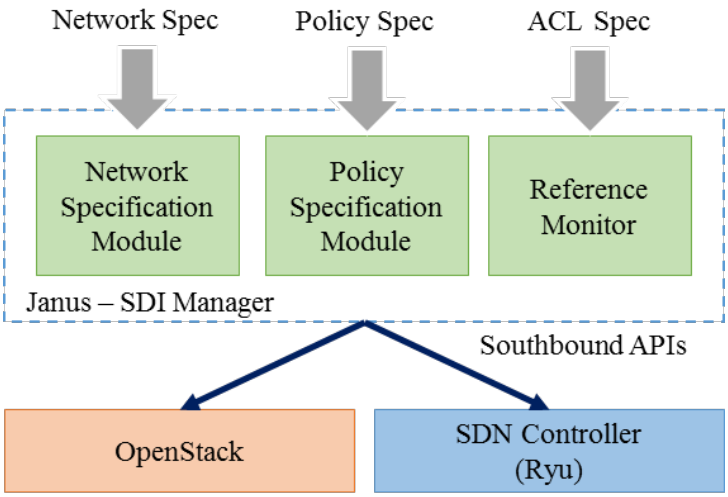


Figure 4.5: Overall architecture of HyperExchange control-plane

erence SDI manager design [88]. Figure 5 shows the overall architecture of control plane which includes SDI-manager and the modules added specifically for HyperExchange. The SDI-manager has southbound APIs to the SDN controller (Ryu) which controls Hardware and Software OpenFlow switches. The other southbound API of SDI-manager is to the Cloud controller (OpenStack). Through this API VMs will be provisioned on demand to host requested VNFs based on user policies.

4.3.1 Main Modules

The control-plane of HyperExchange includes three main modules and each provides its own associated northbound APIs.

Network Specification Module

Any participating network must be specified through this module. InP networks will be defined statically and cannot be defined by user requests through API, but tenants of predefined InPs can define arbitrary VNs through the API. The API specification is the JSON data-model described in section 2. Once a network is defined, this module extracts attributes from the binder list of the specified VN and sends an authorization request

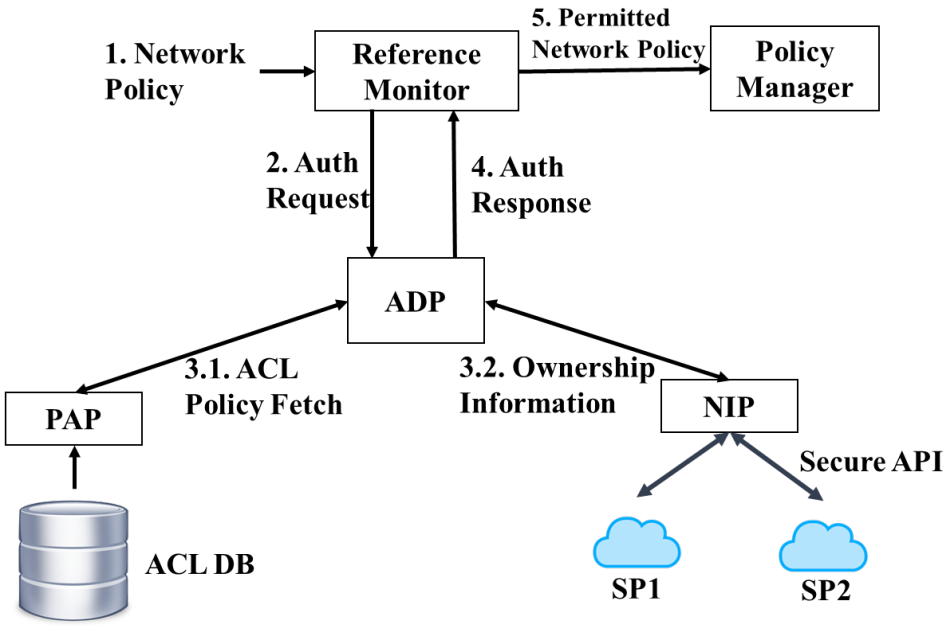


Figure 4.6: Authorization process in the Reference Monitor Module

to the Reference Monitor to authorize the VN. In case of successful authorization, the network specification module creates binding flow-entries from the specifies binder list and installs them to the switch.

Policy Specification Module

The user defined policies for network control will be received by Policy Manager (PM). Upon receiving a policy, the PM module extracts policy attributes and creates an authorization request to the Reference Monitor. If the response is "permitted", the policy will be stored in database and the policy flow table in switch.

Reference Monitor

This module is the reference monitor for authorizing network control policies at HyperExchange. The authorization system relies on ownership information that indicates the subset of authorized flow space of each user. Since VNs from different InPs can participate in HyperExchange, a single static root of trust cannot be used to indicate

ownership information. To address this issue and provide more dynamic and extensible authorization, we have designed an Attribute Based Access Control (ABAC) system for HyperExchange. The overall architecture of this system is shown in Figure 6. This architecture is inspired by XACML [41] and includes three main points. Authorization Decision Point (ADP) receives authorization requests from other control-plane modules. It then retrieves Access Control Policies from Policy Administration Point (PAP) and network ownership information from Network Information Point (NIP). NIP can get the ownership information of tenant VNs from their InP through API on a secure channel.

4.3.2 Technical Issues

Here we mention and address two important technical issues regarding the implementation of the control-plane.

Unified Identity

The authorization system of HyperExchange requires getting ownership information of a tenant VNs from InPs. However, each of the InPs and the HyperExchange itself have a local identity management system. A single user can have an identity in each of these systems. Unifying different identities of a single real user is a fundamental challenge for authorization at HyperExchange. Our primary solution for this problem is to use a centralized identity provider for all of the participating InPs and exchange point. In this case all of the service providers and the exchange point will authenticate tenants through the central identity provider. A single Shibboleth identity server can be used as the identity provider for all parties. In the general, all of the participating InPs may not be joined in the same identity provider. In case of multiple identity providers, an identity peering mechanism is needed which is neutral to the design of HyperExchange.

Network Resource Deallocation

A common feature of multi-tenant environments is the high frequency of deallocation and reallocation of the resources. An example of networking resources can be a VLAN tag or a floating IP address. Since HyperExchange relies on ownership information of networking resources from InPs to authorize tenant policies at exchange point, a resource deallocation in an InP can invalidate a previously authorized tenant policy at exchange point. For example, if Alice as a tenant allocates IP1 = 142.150.208.235 in SAVI as the InP, her policies over IP1 will be authorized and enforced in flow tables of the exchange point. However, once she releases IP1 in SAVI, her previously defined policies that include IP1 are no longer valid at the exchange point. This example shows that exchange point must be aware of resource deallocation/reallocation in the participating InPs to make sure that tenants policies are always valid. An ideal solution for this challenge could be a notification of deallocation from the InPs. However, not all of the participating InPs can provide such capability. For those service providers, a timeout and revalidation mechanism can be used for all of the specified VNs.

In the current implementation the user can specify a VN using a network specification API. The authorization module uses a private API to get VN attributes and to authorize the specification. By this step, the policy specification API is the primary structure of OpenFlow flow-entries. Our current implementation supports OpenFlow actions and traffic steering through a single function as the policies described by user and traffic steering through a chain of VMs is under development. The authorization module uses remote APIs for network specification requests but for policy flow entries, it uses local specifications of network domains.

We have deployed our prototype of HyperExchange between SAVI and GENI testbeds. Our prototype deployment is built on a hardware switch between ORION and Internet2 networks, the underlying interconnection of SAVI and GENI testbeds respectively. Each of the SAVI and GENI networks are connected to the exchange point via a single dedi-

cated physical port. The Experimental results are covered in Chapter 6.

4.4 Discussion and Remarks

How does HyperExchange compare with existing Software Defined Exchanges? Richter *et al.* [96] presented that how a central route server can facilitate peering at IXPs. The concept of Software Defined Exchanges (SDX) and use of SDN for flexible inter-domain networking is introduced in [36]. The primary implementation of SDX [47] enables participating autonomous systems to overwrite basic BGP route selection at IXPs. The Cardigan project [106] was another primary attempt to enhance inter-domain networking by use of SDN. Based on the measurements provided in [45] none of these primary implementations can address scalability requirement of a large scale exchange point. Industrial Scale SDX (iSDX) [45] is a more recent implementation of SDX based on Ryu controller which has addressed the scalability issues of primary implementations by use multi-table feature of higher version of OpenFlow. All of these implementations have some features in common. Only IP networks can exchange traffic in current SDX architectures. Moreover, these implementations have not targeted exchange between multi-tenant environments so there no clear division between service provider networks and tenant networks.

Why is it beneficial to peer Virtual Networks? Expanding a Virtual Network over multiple Infrastructure Providers is a requirement for future services; because of the geographical distribution of the network (e.g. Akamai global CDN that is expanded over many multiple ISPs) and cost optimization strategies and federation (e.g. Apple iCloud that uses Amazon EC2 in addition to their own data-centers). In these cases, the network needs to be deployed in multiple InPs and it is required to federate these InPs and their networks. In fact the main motivation of this project came out of a real problem that is network federation of two research testbeds (SAVI GENI). The only

current solution is encapsulation over IP (i.e. overlays) which inherits the ossification of the current Internet and provides a narrow solution for specific use cases, not a holistic approach for network federation of multiple SDI domains.

How does control authorization in HyperExchange compare with other network flow authorization frameworks? Authorization of network control policies from tenants of different service provider poses new challenges in the design of exchange points. Even though the authorization of network control is discussed in FLANC [43] and used in iSDX implementation, a static root of trust is considered to have all ownership information of networking resources and it is not mentioned that how this information can dynamically be gathered from participating service providers. Particularly, in a Virtual Networking Environment, network domains of tenants may change rapidly and the Reference Monitor of the exchange point must be able to recollect these ownership information as they change in participating Infrastructure Providers. HyperExchange uses secure APIs to collect and update tenant ownership information from participating InPs.

How is HyperExchange related to Virtual Network Embeding? Virtual Network Embedding is concerned about the mapping between virtual topology and physical topology and is well studied in the literature [113] [29] [93]. In case where multiple InPs are involved, an end-to-end VNE platform can used to slice and map slices of VNs to each InP [27]. However, these InPs can use different logic and technologies for network virtualization. For instance, there are multiple protocols available to create a virtual Layer-2 network including GRE, VLAN, VXLAN and MPLS and each of them might be used in one of the InPs. However, the nature of a Layer-2 network is the same and the exchange point must be able to peer Layer-2 networks independent of the underlying protocol. In these cases the VNE platform can employ a HyperExchange to enable traffic exchange between dissimilar InPs.

Chapter 5

MD-IDN

Intent-Driven Networking (IDN) promises to provide a simple, yet expressive high-level abstraction over the network controller [54]. This abstraction hides the unnecessary details of the underlying infrastructure from users and allows them to customize network configuration using human readable intents.

Current IDN frameworks [6–8] have implemented intents with a specific design and compilation process. Therefore, there is no uniform notion of network intents among different control platforms. Additionally, in these frameworks user intents are not clearly differentiated from provider intents. Furthermore, current intent NBIs compile intents over a flat non-abstracted topology, which is not scalable and feasible in multi-domain scenarios.

In this chapter, we introduce MD-IDN [14], a framework for end-to-end Multi-Domain Intent-Driven Networking over multiple SDI deployments. To enable intent compilation over heterogeneous network domains, we based our design on the general graph representation of network intents presented in Chapter 3. MD-IDN framework first compiles the user-requested intent in the form of a intent graph over an abstracted multi-graph of domains and their interconnections. This process results in a set of local intent graphs that are to be submitted to the local intent framework of each domain. MD-IDN introduces

the following particular contributions:

- A generic and extensible graph representation for user-defined network policies and intents. This intent graph abstracts away details of the network topology from the users' perspective.

- We introduce and evaluate a set of algorithms to automatically distribute and scale the compilation and installation of intents in the form of a intent graph over heterogeneous and multi-domain networks.

- Our proposal for MD-IDN goes beyond a paper design. It is deployed and available as a public service for SAVI Testbed users, and has been under continuous improvement and development over the past year.

5.1 End-to-End Network Intents

As mentioned before, enabling multi-domain network intents is not possible by simply adding an IDN framework for the entire global network. The challenge originates mainly from scalability issues and the fact that different domains may use various control platforms and consequently not allow direct control over their network to an external entity. To address these issues, based on the proposed intent graph model in Chapter 3, a hierarchical and distributed IDN design is presented in this Chapter. In this design, a global IDN first processes the requested intents over an abstracted topology to generate a set of local intent graphs.

5.1.1 Topology Abstraction

We proceed by briefly describing the topology abstractions that are required for feasible and scalable compilation of multi-domain intents.

Local Topology

The internal network topology of each domain is referred to as local topology, shown in the collections within Figure 5.1. In a real multi-domain scenario where different domains could potentially have autonomous control, the domain provider would not provide the detailed local topology view to the external entities. However, the local IDN deployed over the internal control platform could have access to this topology.

Global Topology

The global topology is the actual detailed graph including all the nodes in each domain and the interconnections between the domains as depicted in Figure 5.1. It is clear that this graph grows drastically as the number or the average size of domains increases. However, this topology is not usually available in practical multi-domain environments since different domains keep their internal topology private. In addition, no single central entity would be able to directly define control policies over such topology.

Inter-domain Multigraph

The Domain Multigraph is an abstracted graph where each domain is represented as a node and the domain interconnections as edges. This graph abstracts away the internal topology of the domain. This multigraph is built with minimal amount of information (only the interconnection points) provided by each domain, and thus it is much smaller than the real global topology.

5.1.2 Multi-domain Intent Compilation

The general multi-domain compilation is a two-step process. In the first step, the input intent graph gets projected over the domain multigraph resulting with the domain intent graph (Algorithm 2) which then gets processed into separate local intent graphs for each of the involved domains (Algorithm 3).

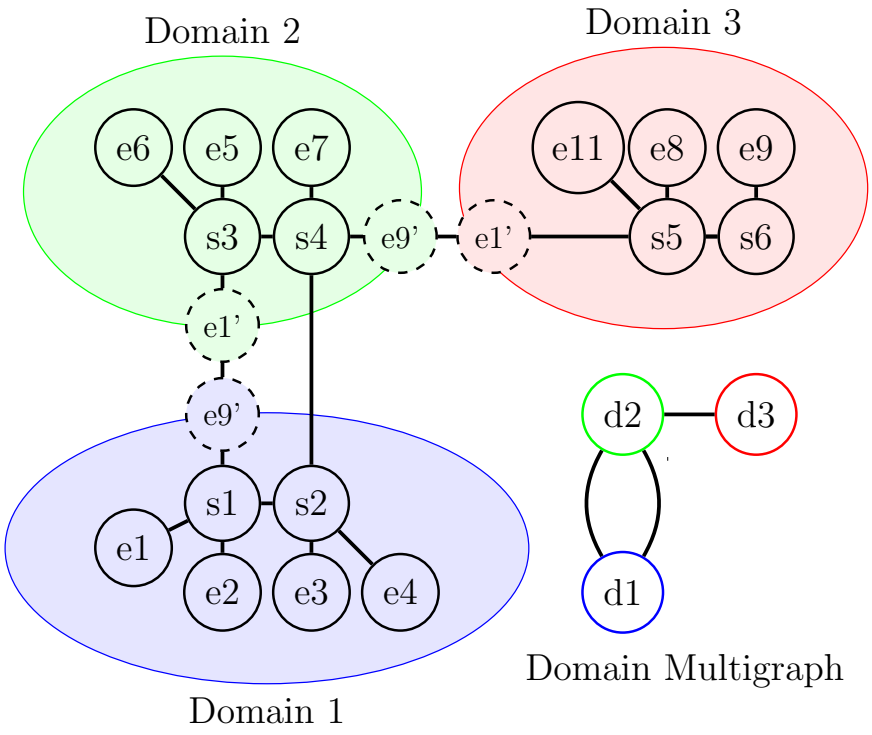


Figure 5.1: A sample multi-domain topology

Algorithm 2 receives a intent graph G_P and begins by removing all the edges whose both source and destination belong to the same domain and directly places that policy edge in the local intent graph of that domain. After this process the remaining edges in the input G_P are the ones that are between two endpoints from different domains. Next, it starts to construct the domain intent graph. To achieve that, it maps every node to its domain in the domain intent graph and for each edge in the input intent graph, an edge in the domain intent graph gets created. Figure 5.2 exemplifies this process, where Figure5.2.a shows an end-to-end Layer2 connectivity intent between e1 from d1 and e9 from d3. Since d1 and d3 are not directly peered, a transit domain like d2 is required. Accordingly, Algorithm 2 results with a domain intent graph as shown in Figure 5.2.b.

The resulting domain intent graph is delivered to Algorithm 3 which generates the adequate local intent graphs for each domain involved in the domain intent graph. Each domain can process intent graphs that only contains its internal endpoints. To identify external nodes to the internal topology of another domain, we use the notion of shadow

Algorithm 2: Projection of multi-domain policy graph over domain multigraph

```
CompileMDPolicyGraph (G_P, G_D, M)
    inputs  : G_P = (V_P, E_P)
              // multi-domain policy graph
              G_D = (V_D, E_D)
              // domain multigraph
              D
              // mapping of nodes to domains
    outputs : G_dict
              // Dict of local policy graphs per domain
              G_I = (V_I, E_I)
              // Projected policy graph over domain multigraph
    initialization();
    foreach policy edge e ∈ E_P do
        // check if the edge is intra-domain or inter-domain
        if D[e.src] = D[e.dst] then
            d ← D[e.src]
            G_dict[d].addEdge(e)
            G_P.removeEdge(e)
        else
            path_i ← getPath(G_D, D[e.src], D[e.dst])
            foreach d_n, d_{n+1} ∈ path_i do
                f_i ← "e.src : e.f : e.dst" e_i ← newedge(d_n, f_i, d_{n+1})
                // create a new edge with lable of e.src:e.f:e.dst
                G_I.addEdge(e_i)
    return G_dict, G_I;
```

nodes which represent mock nodes registered in the internal topology at the interconnection point. Following our example, e9 is not known for d1, thus it gets registered as e9´ in d1 and e1 gets registered as e1´ in d3. Now, both nodes are external to d2, so both e9´ and e1´ get registered in d2. Consequently, each domain has an updated topology view and can receive a local intent graph. Figure 5.2.c shows the three resulting local intent graphs for each domain.

The second row of Figure 5.2.d shows a multi-domain service chaining intent graph in which the source is in d1, two middleboxes are in d2 and the destination is in d3. Upon running Algorithm 2, the domain intent graph shown in Figure 5.2.e is obtained and it gets passed to Algorithm 3 in order to generate the three local intent graphs as shown in Figure 5.2.f, including the required shadow nodes.

Algorithm 3: Decomposition of domain projected policy graph into local policy graphs

CompileMDPolicyGraph (G_I, G_dict, D)

inputs : $G_I = (V_I, E_I)$
// Projected policy graph over domain multigraph
G_dict
// Dict of local policy graphs per domain
D
// mapping of nodes to domains

outputs : G_dict
// Final dict of local policy graphs per domain

$initialization()$;
foreach *inter-domain policy edge* $e_i \in E_I$ **do**
  $(src, f, dst) \leftarrow e_i.label$
  $dst' \leftarrow registerShadowNode(D[src], dst)$
  $e_{scr\_domain} \leftarrow new\ edge(src, f, dst')$
  $G\_dict[D[src]].addEdge(e_{scr\_domain})$
  $src' \leftarrow registerShadowNode(D[dst], src)$
  $e_{dst\_domain} \leftarrow new\ edge(src, f, dst')$
  $G\_dict[D[dst]].addEdge(e_{dst\_domain})$
return G_dict, G_I;

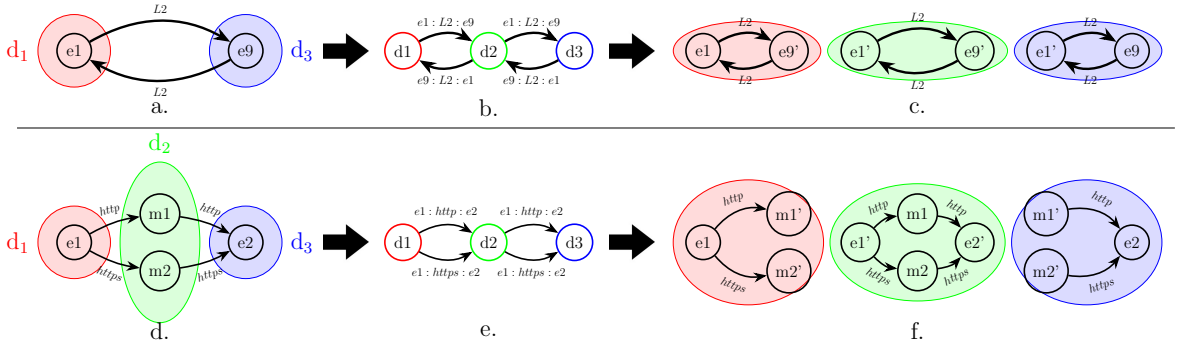


Figure 5.2: Decomposition of sample multi-domain intent graphs into local intent graphs for each domain

5.2 Implementation

We now describe an implementation of MD-IDN as shown in Figure 5.3. The architecture has two main software components: the local IDN and the global IDN. Our Python-based implementation of the main platform components has over 6000 lines of code and has been under continuous refinement since last year. We proceed by briefly describing the implementation of the main components.

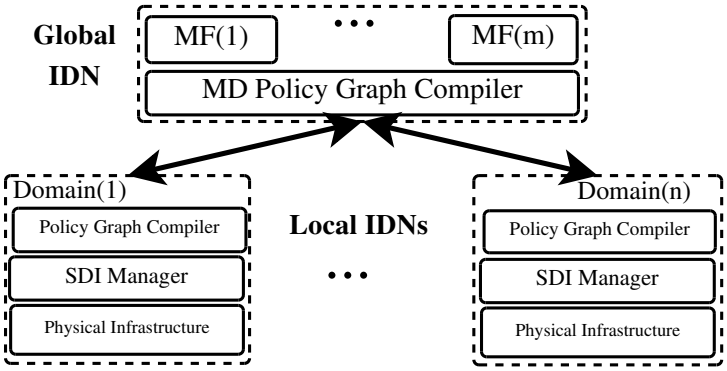


Figure 5.3: MD-IDN architecture and main components

5.2.1 Local IDN

The local IDN provides the basic intent graph compilation for each domain. Upon receiving an input intent graph, it performs the local compilation process using Algorithm 1. To find paths between endpoints, it uses the southbound APIs provided by the local SDI manager. In addition, it may receive requests to add shadow nodes, upon which it adds the node to the current topology view. The local IDN also tracks events in the local network, thus if certain network events that might affect the existing intent intent graphs occur, it will trigger a recompilation based on the latest network changes and will essentially re-install the intent, if possible. Within the local IDN, each intent might have different states including pre-compilation, compilation, installation, installed and failed.

5.2.2 Global IDN

The global IDN receives multi-domain intent requests from users and maps each request to an adequate intent graph using mapping functions. It then performs a two-step compilation provided by Algorithm 2 and Algorithm 3. The resulting set of local intent graphs will be submitted to the local IDN for the involved domains. To construct and maintain the domain multi-graph, a simple domain discovery mechanism is implemented which enables the global IDN to receive domain interconnection points from each local IDN. After the compilation process, all the local intent graphs will be submitted to the local

IDNs using asynchronous non-blocking RESTful calls in parallel. Once the results are received, if a failure had happened within the domains hosting an endpoint, the intent setup will fail; otherwise, if the failure happened in a transit domain, the global IDN will try to find an alternative transit domain, if one exists. Multi-domain intents can have different states similar to the local IDN states. In our primary implementation the global IDN is a single centralized platform. However, in the case of many multiple local IDNs, the global IDN could be implemented in a distributed fashion where each instance is connected to a set of local IDNs.

5.3 Evaluation

In order to evaluate the design and implementation of the platform we targeted the performance of the multi-domain intent compilation process to evaluate the feasibility of the MD-IDN design and the effect of the proposed algorithms in a multi-domain scale. To evaluate our design, we measured the compilation time of a policy graph for a WAN intent over a large multi-domain topology. In these experiments two setups are compared: a flat compilation over the global topology, similar to the available alternative practices; and a hierarchical distributed compilation process of MD-IDN. To focus on the intent compilation time, we have eliminated the communication round trip times and the flow installation times.

Using the described scenario, two different experiments are performed to analyze how the MD-IDN design can improve the intent compilation performance which represents the major bottleneck as the network size grows in a multi-domain scale. In terms of path selection, CSPF [78] (Constrained Shortest Path First) algorithm is used in both cases, similar to the one used in ONOS controller [7]. In case of using optimal path selection algorithms, compilation time over the flat global topology considerably will be increased.

In the first experiment the compilation time is measured against an increasing number

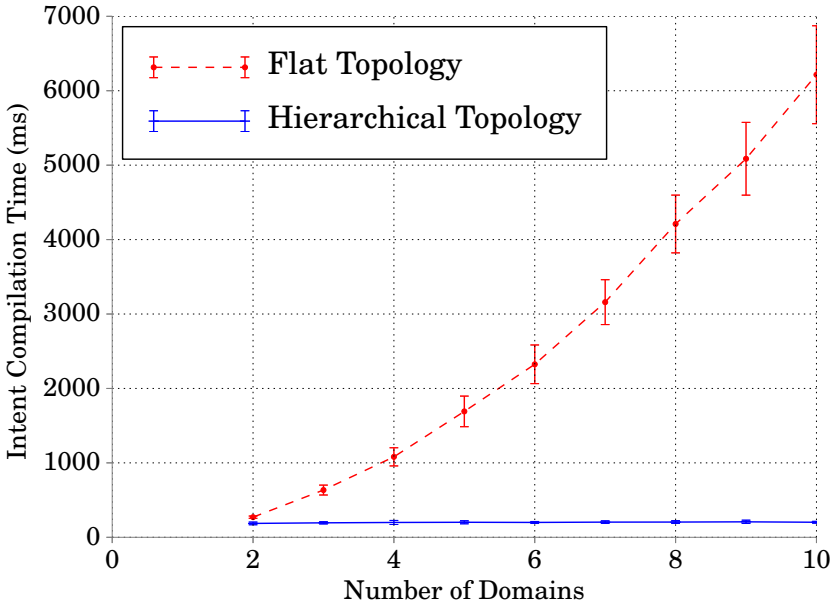


Figure 5.4: Intent compilation time vs global network size with increasing number of domains

of network domains ranging from two to ten while keeping the topology size of each domain fixed with an average of 1K nodes per domain. Figure 5.4 shows that as the number of domains increases, the case of naive compilation over the global topology results with considerable increase, whereas the MD-IDN approach keeps the compilation time almost fixed. In case of having ten network domains, MD-IDN performs 30x faster than the naive approach. This result stems from the fact that in MD-IDN the primary compilation over the abstracted domain multi-graph happens discreetly fast, thus not contributing much to the overall compilation time. The rest of the compilation process happens in parallel in different domains. Therefore, the multi-domain intent compilation time remains almost equal to the maximum time elapsed in the local domains.

In the second experiment the number of network domains is kept fixed to four, while the size of each domain is increased from an average of 0.25K to 32K nodes per domain. As depicted in Figure 5.5 the MD-IDN shows a gradual increase, whereas the performance of the naive approach falls much faster.

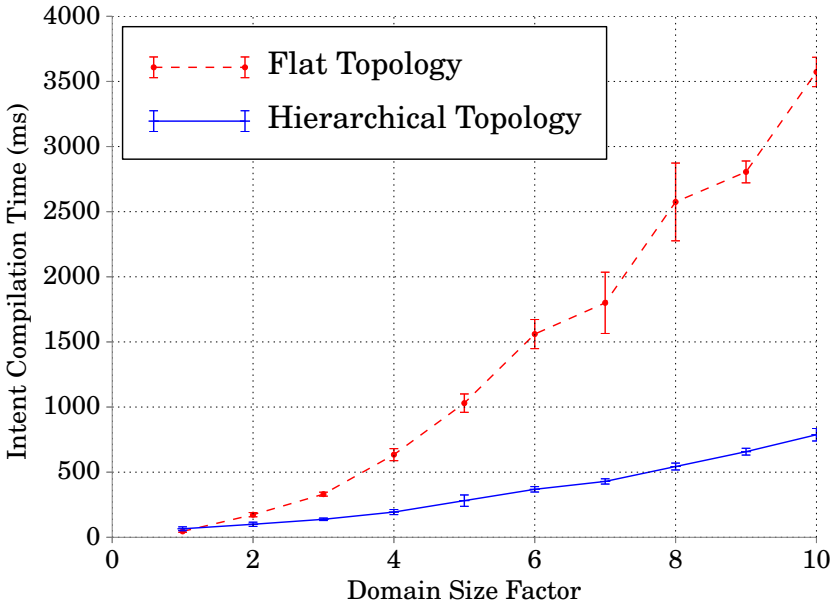


Figure 5.5: Intent compilation time vs global network size with increasing size of domains

5.4 Discussion and Remarks

What is the definition of Intent and Intent Driven Network (IDN) in the context of this paper? Policy-based management is a broad area with a long history of work and proposals on network policy mapping and policy refinements. Our use of the term Intent applies specifically to the Northbound Interface (NBI) in Software Defined Networks (SDN). In this context, high level management policies are defined as a set of Intents enforced through an Intent NBI which is a declarative middleware between network applications and the controller. This definition follows Intent frameworks implemented in industry-leading SDN controllers (ONOS and OpenDaylight) and is established by the Open Networking Foundation [54]. our work follows the same concept as network intents in SDN and is not proposing an alternative definition. There are additional recent white papers and industry initiatives from Huawei [6] and Cisco on intent-based networking following the same principles and definitions.

How MD-IDN is different from the current Intent-based frameworks? In the current IDN solutions [6–8] intents are implemented as a set of programming modules

to automate the mapping between user inputs and low-level flow rules in the network. This mapping procedure (i.e., intent compilation) may vary based on the control platform, infrastructure vendor and even the intent type within the same platform. In contrast, we proposed a precise and uniform intent abstraction model based on intent graphs that can be integrated within any underlying control platform. Among the policy abstraction models that have been used in prior works [31,32,84,94], PGA [91] has the most similar abstraction model, where policy graphs are defined for endpoint groups in the network and are used as an intermediate abstraction for network intents. PGA targets scenarios where multiple parties should have control over one enterprise network, whereas MD-IDN mainly focuses on multi-domain and widely distributed networks. Network topology abstraction in Software-Defined Networking has been previously proposed and used [11, 60,79,104]. However, the unique part of our work is the use of this abstraction to simplify and scale the intent compilation process. There are distributed or hierarchical controller designs for SDN [18, 49, 66, 90], the SDI manager in the SAVI testbed [87] also follows the same approach. However, this work is not about proposing another distributed controller design. Instead, MD-IDN uses hierarchy and distribution to improve intent compilation scalability over geographically distributed network domains. This work adds the following main contributions to the recent efforts in IDN: - Enabling End-to-End network intents over Multiple domains/providers - Algorithms that scale to the extent of large and geographically distributed SDNs - A very first implementation of IDN in a testbed that supports real users and evaluations, and that shows significant performance gains. In fact, the motivation of this work came from the real experience of operating the testbed and witnessing the limitations and difficulties of users in using the SDN capabilities in real scenarios.

What are the roles of User and Provider in the context of this paper? The roles of User/Tenant and Provider are considered in the context of Software-Defined Infrastructures (SDIs) and an Infrastructure Provider. In this context the provider provides

the virtualized infrastructure and the user/tenant uses the infrastructure to deliver an application service to the end users. Thus a content provider like Netflix or any application service provider like Skype are considered to be users/tenants of the Infrastructure Provider.

What is the relation between the compilation phase and topology discovery protocols? The compilation phase relies on the topology information provided by the underlying SDI manager in each domain. The SDI manager itself would employ discovery protocols (such as LLDP or BDDP) to gather topology information and to build the topology graph.

Why do the authors need to define intent classes? Although the graph model is capable of defining any customized intents, the classes act as an additional abstraction that provides users with a set of commonly used advanced network functionalities in the form of an intent class.

What are the Intent states handled by each of the Local and Global IDN? The Global IDN keeps the state of end-to-end intents over the domain multigraph while local IDNs handle the state of local intents over the local topology. A change in the local topology would trigger a state change in that local IDN and if the change makes the local intent uninstallable the state of the end-to-end intent will change in the global IDN.

Chapter 6

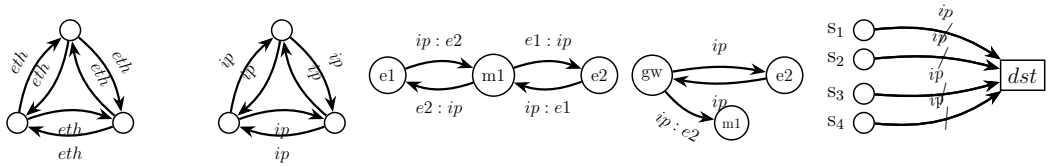
Use cases and Experiments

In this chapter we first introduce the available usecases for MD-IDN followed by a set of experiments. We also provide an experiment of multi-provider setup between SAVI and GENI testbeds using HyperExchange.

6.1 MD-IDN Usecases

To deploy MD-IDN, we have added a local IDN to each region which uses southbound APIs offered by the SDI manager to retrieve the required information and to install flow-entries to the underlying network. The required information includes the interconnection points to the other domains and the available paths between the internal endpoints based on the provided topology view by the SDI manager. The local IDN provides northbound APIs for submitting policy graphs and registering shadow nodes. Apart from local IDNs in each region, a deployed global IDN receives end-to-end intent requests from users, and it generates and submits local policy graphs for each of the involved domains.

Over the above described deployment, we have provided five intent classes and mapping functions from intent inputs to policy graphs accordingly. Each of these intent classes enables a unique network service beyond the common networking services available in OpenStack. The intents can be easily provisioned by the Testbed users from an



a. Virtual L2 WAN b. Distributed Virtual Router c. Service Function Chaining d. Tapping e. Distributed Firewall

Figure 6.1: Sample policy graphs of the five Intent classes that are currently available in the SAVI Testbed

extended web UI portal or by using RESTful APIs. Figure 6.1 demonstrates sample policy graphs for each intent class, where the endpoints could reside within different network domains/regions. We proceed by briefly summarizing each of these classes.

6.1.1 Virtual L2 WAN

This intent class provisions a virtual WAN between a set of endpoints across the Testbed. Originally, there is no such service provided by an out of box installation of OpenStack. Figure 6.1.a shows a sample policy graph for this service between three endpoints. After compilation, flow-entries related to Ethernet traffic will be installed along the selected path between the instances in order to establish the distributed WAN without the involvement of a central switch. The process is transparent from users' perspective as they only need to instantiate a WAN and add endpoints to it.

6.1.2 Distributed Virtual Router

The DVR intent provisions a distributed virtual router between a selected set of endpoints from different IP subnets in different regions or projects. This intent enables a layer 3 connectivity between the existing IP addresses of the selected instances without the involvement of a central router. The intent eliminates the need of flows passing centralized regional routers; which is the default OpenStack approach causing additional delay and bandwidth bottleneck for inter-domain traffic.

6.1.3 Service Function Chaining

The service chaining intent enables redirection of specific subset of the traffic through a pre-defined SFC. This intent enables multi-domain service chaining for which the current alternative approach is using overlay tunnels over the legacy IP inter-networking. With the legacy approach users have no control over the underlying network and in addition, tunneling causes considerable overhead, rendering the SFC unusable for practical high-bandwidth scenarios. Whereas this intent class provides an actual underlay SFC experience intuitively to the user over multiple network domains, while the provider of each domain preserves the control and privacy of the internal topology details.

6.1.4 Tapping

The tapping intent class enables network traffic to be duplicated to an additionally specified instance taking a monitor role. The instance receives copies of network traffic for potential further processing, while allowing the default traffic path. In case of detecting a malicious flow, it could be blocked or redirected to a middlebox.

6.1.5 Distributed Firewall

This intent class enables multi-domain security policies in addition to the default OpenStack security groups. It operates based on a black-list approach to block malicious flows at the source domain, leaving the destination domain network unutilized, as opposed to the OpenStack security groups enforcing a white-list approach right behind the destination endpoint. The distributed firewall intent could be leveraged for DDoS defense by easily updating and managing the black-list. There has been a huge body of research for DDoS detection [21] [76] [61] [103], however in a practical case where attacking flows are coming from different external domains, the main challenge is the effective mitigation upon the primary detection as the victim domain cannot block the flows outside of its

network domain. This intent class spans the effective defence footprint to a multi-domain scale.

6.2 Performance results and Evaluation

To evaluate the credibility of MD-IDN, we designed and performed a second set of experiments targeting the real use cases. In these experiments we have compared the new network functionalities enabled by MD-IDN with the available alternative solutions; in terms of network delay and bandwidth capacity. The data plane performance is evaluated through experiments based on the MD-IDN use cases that are currently available in the SAVI Testbed. In these experiments, three Intent classes (Service Function Chaining, Distributed Virtual Router and Virtual L2 WAN) are tested and compared with the available alternative solutions for Testbed users.

To test the SFC intent class, we created a scenario where traffic from a source passes through two middleboxes prior to reaching the destination. As an alternative, we created the same chain using VXLAN overlays and performed delay and bandwidth tests on both. In Figure 6.2a, the delays incurred from the chaining using VXLAN overlay, the underlay approach using MD-IDN, and the default case with no chaining involved are measured. As expected, the default one has the least delay, while with chaining included, the MD-IDN setup adds insignificant overhead, as opposed to the VXLAN chaining overhead being 3x to 4x more.

In terms of the bandwidth, Figure 6.3 shows a comparison of the TCP bandwidth performance between the chaining intent class and the direct communication with no middlebox involved. While the default case with no middlebox saturates up to 25 Gbps, the chaining intent with two middleboxes can easily saturate up to 15 Gbps. The performance of the VXLAN case is not included in the graph, as its performance was far below the other two numbers. In fact, once the sequence of the two middleboxes is added,

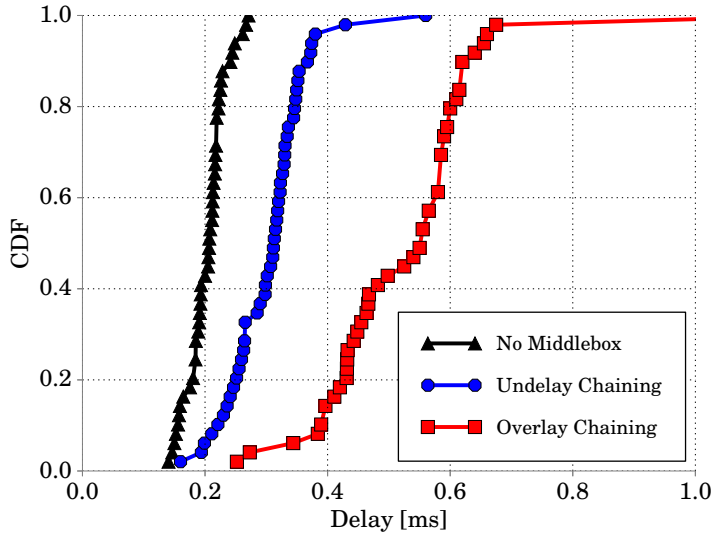


Figure 6.2: End-to-End delay, direct chaining using MD-IDN vs overlay chaining using VXLAN vs no middlebox

the TCP bandwidth drops to less than 100 Mbps. The main reason behind this drastic decrease is related partly to VXLAN, as it encapsulates overlay TCP packets in underlay UDP packets, which in turn affects KVM's networking [5], as it does not perform well with UDP packets. We used other encapsulation methods like GRE and received almost identical results.

In another experiment, a virtual L2 WAN was created between instances from Toronto Edge data center and Toronto Core data center. To compare with an alternative approach, a virtual L2 connection was created using VXLAN ports. Figures 6.4 show that MD-IDN provides less network delay, since it does not have the encapsulation overhead.

In the last experiment, a DVR instance was used to route traffic between two endpoints with different subnets. By default the flow should pass software routers inside the controller machines, however DVR eliminated this redirection. Figure 6.5 shows the comparison of the delay incurred by the DVR intent class and the default approach, and it can be concluded that the DVR intent class incurs much less delay.

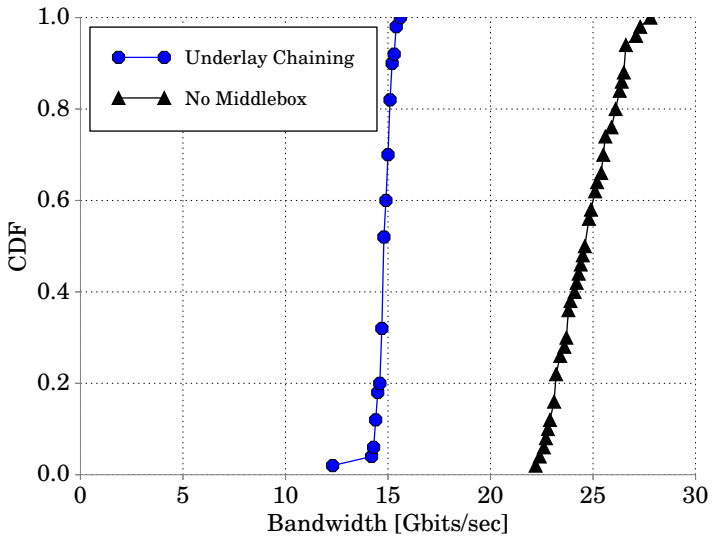


Figure 6.3: Service chaining bandwidth degradation of direct chaining using MD-IDN vs no middlebox

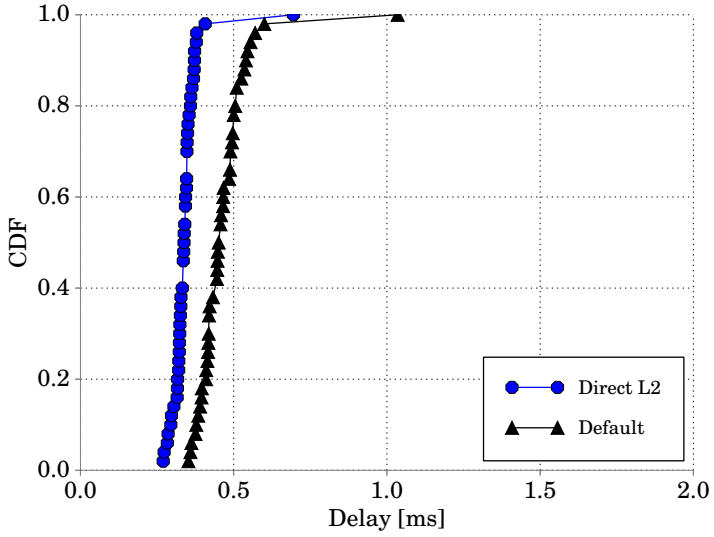


Figure 6.4: End-to-End delay, direct Layer-2 intent vs VXLAN over IP

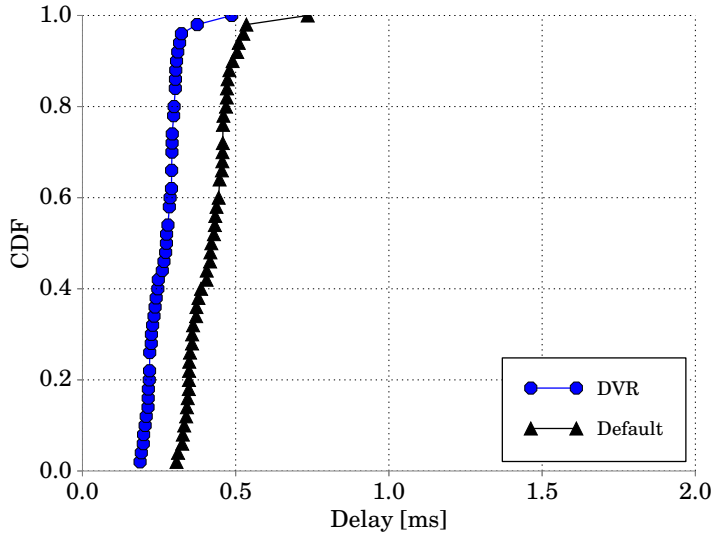


Figure 6.5: End-to-End delay, DVR vs Default OpenStack router

6.3 Multi-Provider Experiment: Peering of Layer-2 Networks [13]

By use of HyperExchange it is possible to setup layer-2 networks over multi-region clouds without encapsulation. A generic virtual layer-2 network is key for any further innovation in upper layers such as IP alternatives. Also it makes it possible to simply define custom and private IP networks on a Wide Area layer-2 Network. These features make inter-domain layer-2 peering a beneficial usecase of HyperExchange. Layer2 networks in SAVI are established by end-to-end path stitching on OpenFlow switches based on MAC addresses of endpoints. On the other side GENI uses VLAN tags to create a layer-2 network between arbitrary set of VMs. Each of these InPs has a different logic to realize a Layer2 network. Thus peering of two virtual Layer2 networks on both sides as single end-to-end layer-2 network is the challenge we addressed by use of HyperExchange. Our flexible model for networks at exchange point allows us to simply define and peer these networks in a uniform manner. In our test case we had two VMs in SAVI testbed connected in a Layer2 network in Toronto with the following network data structure at exchange point:

```
{ net_id: SAVI_L2,
      Net_domain: SAVI_NET,
      binder: {{mac = fa:16:3e:65:ac:52},
              {mac = fa:16:3e:5d:33:db}}
      metadata: {}
}
```

In GENI side we defined a VLAN including a VM in Chicago with the following network data structure at exchange point:

```
{ net_id: GENI_L2,
      Net_domain: GENI_NET,
      binder: {{VLAN_TAG = 7273}}
      metadata: {}
}
```

As can be seen, the network attributes of a Layer2 network in GENI is not related to the number of nodes. However, in SAVI as the number of nodes in a Layer2 network increases, the network attributes to be authorized will also increase. Figure 6.6 demonstrates the relation of number of nodes and the time it takes at HyperExchange to query each InP to verify network specifications. We have emulated authorization API to GENI by users own credentials. However, an speaksfor API is under development in GENI that can be used for remote authorization on behalf of the user.

Figure 6.7 shows the comparison of cumulative distribution of time over 10 trials for specification of the GENI side VN. The overall time is the time of processing VN specification, authorization through remote API and creation and installation of binding flow-entries to the switch. In Figure 6.7 the black line is the time excluding remote authorization time and the dotted line is the entire time. The figure shows that a large amount of time is spent on remote authorization. Thus our technique to authorize only

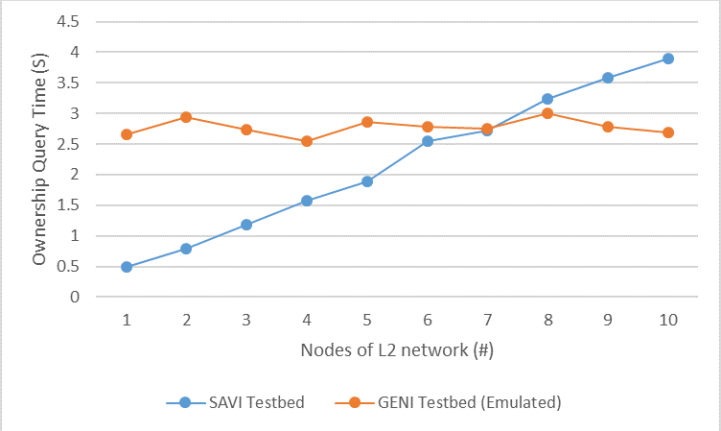


Figure 6.6: Remote attribute authorization time based on the number of nodes in VN

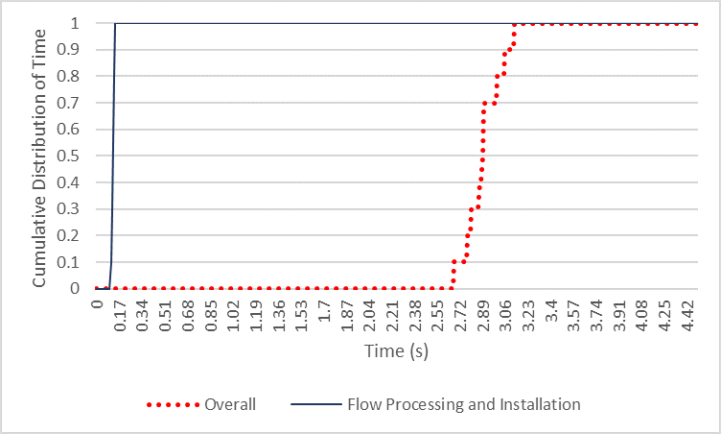


Figure 6.7: Time analysis of network specification in HyperExchange (GENI Side)

the network specification using remote API and authorizing later policies by local and pre-authorized domains can effectively reduce the policy installation time.

We defined the following policies to peer these VNs at HyperExchange:

```
bind(SAVI_L2).incoming()
        .modify({"type":"SET_VLAN",
        "value":7273})
        .output(GENI_L2)

bind(GENI_L2).incoming()
    .modify({"type":"STRIP_VLAN"})
    .output(SAVI_L2)
```

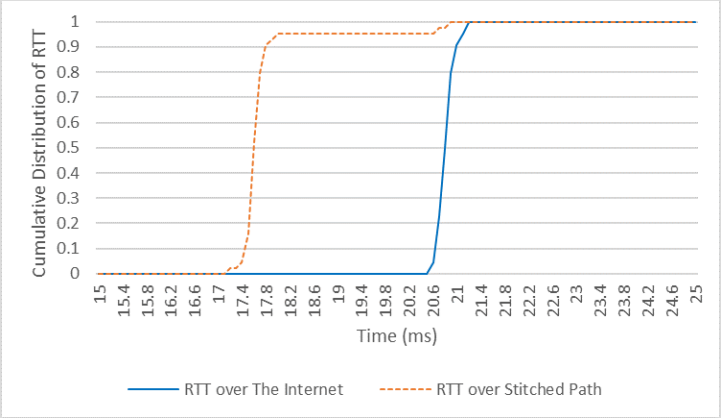


Figure 6.8: RTT comparison of the regular path over the Internet vs the directed path through the exchange point

As mentioned earlier, policy specification must be installed as flow entries at this state of our implementation and the above code is the pseudo representation of the peering policies. As depicted in Figure 6.8, we have measured Round Trip Time from a VM in Chicago to a VM in Toronto for the regular path over the Internet and Layer2 directed path through exchange point. Our experiment shows that HyperExchange flexibility helps InP tenants to setup arbitrary end-to-end paths between VNs over autonomous Infrastructures.

Chapter 7

Conclusion and Future Directions

In this thesis we presented abstractions as well as an architectural model for end-to-end orchestration over multiple autonomous service providers. We started by defining a uniform intent abstraction model named application intent graph which encompass the communication model of a distributed cloud application and is able to express new network functionalities. Using the intent graph model, cloud application developers can provision advanced networking configurations without requiring to program the network.

Based on the model, a uniform compilation algorithm is introduced and we presented a straw-man architecture to realize the proposed model in a multi-domain cloud environments. We mentioned the practical limitations of the initial design mainly for scalability and inter-provider network slicing.

In order to address the multi-provider network slicing problem, we presented HyperExchange which enables traffic exchange between Infrastructure Providers and their hosted Virtual Networks. We built a formal model for traffic classification at exchange point and extended it to design the traffic switching pipeline of HyperExchange. Our formalism allowed us to define a protocol-agnostic network model that satisfies the feature of protocol customizability of Virtual Networking Environemnts. Based on the formal specifications, we proposed an extensible architecture for the switching fabric of the ex-

change point. Our current design for authorization system is inspired by XACML but does not fully support its standards.

we presented MD-IDN to enable end-to-end intents over multi-domain, multi-tenant networks. This uniformity allowed us to extend the intent framework to cover multi-domain intents, where each domain could have autonomous network control and would only require receiving and installing local policy graphs. A set of algorithms are provided to map a policy graph spanning multiple network domains to a set of local policy graphs. Our evaluations show that MD-IDN can easily scale to numerous multi-domain networks, each containing hundreds of nodes.

The proposed architecture is implemented and deployed in the SAVI Testbed over multiple regional network domains. Also, a propototype of Hyperexchage is deployed between SAVI and GENI testbeds. To demonstrate the value of MD-IDN, we developed and deployed five intent classes available to the Testbed users. Our use case experiments confirm that network functionalities enabled by MD-IDN provide better networking performance and experience to the Testbed users than the available alternatives, while in addition it provides an intuitive and easy to use interface for cloud application developers.

7.1 Future Directions

This work represents an initial step and provides a feasible and practical direction towards an Intent-Driven Internet, where users can request end-to-end network intents over multiple providers. Here we mention the potential research directions for this thesis:

- **Inter-domain traffic engineering:** Currently infrastructure providers have many options to optimize bandwidth allocations and guarantee certain quality of service levels in their internal networks. However, inter-domain traffic engineering has seen very limited success. SDXs create an opportunity to provide fair and optimal bandwidth allocation at the inter-domain level. Also, given the end-to-end pro-

grammability, many cost optimization strategies for cloud service deployment can be studied.

- **Integration with orchestration tool:** There has been extensive standardization efforts for cloud service orchestration and many modeling languages have been introduced including TOSCA. As a future direction, we can extend current widely-used orchestration tools and modeling languages to include our application intent graph modeling and enable advanced networking configuration.

- **Improving conflict detection and resolution** The intent graph abstraction simplifies the task of sanity checks and conflict detection between different intent requests. However, when different tenants are competing for a specific resource or bandwidth, a more intelligent and fair conflict resolution approach is requierd.

Bibliography

[1] Cloudify, http://www.cloudify.co.

[2] Openstack heat orchestration, https://wiki.openstack.org/wiki/heat.

[3] Ubuntu juju, https://www.ubuntu.com/cloud/juju.

[4] Geni testbed wiki. 2017.

[5] Kvm virtualization. 2017.

[6] Nemo: An applications interface to intent based networks. 2017.

[7] Onos intent framework. 2017.

[8] Opendaylight network intent composition. 2017.

[9] Bernhard Ager, Nikolaos Chatzis, Anja Feldmann, Nadi Sarrar, Steve Uhlig, and Walter Willinger. Anatomy of a large european IXP. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM.

[10] T. Anderson. Networking as a Service, HOTI-21 Keynote, `https://www.youtube.com/watch?v=bEtq_4arFz0`, (2013).

[11] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing.

In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM.

[12] Saeed Arezoumand, Hadi Bannazadeh, and Alberto Leon-Garcia. Hyperexchange: A protocol-agnostic exchange fabric enabling peering of virtual networks. In *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*, pages 204–212. IEEE, 2017.

[13] Saeed Arezoumand, Hadi Bannazadeh, and Alberto Leon-Garcia. Layer-two peering across savi and geni testbeds using hyperexchange. In *Integrated Network and Service Management (IM), 2017 IFIP/IEEE Symposium on*, pages 907–908. IEEE, 2017.

[14] Saeed Arezoumand, Hadi Bannazadeh, and Alberto Leon-Garcia. Md-idn: Multi-domain intent-driven networking in software-defined infrastructures. In *Network and Service Management (CNSM), 2017 13th International Conference on*. IEEE, 2017.

[15] Jonathan Baier. *Getting Started with Kubernetes*. Packt Publishing Ltd, 2015.

[16] Ilia Baldine, Yufeng Xin, Anirban Mandal, Chris Heermann Renci, Jeff Chase, Varun Marupadi, Aydan Yumerefendi, and David Irwin. Networked cloud orchestration: a geni perspective. In *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, pages 573–578. IEEE, 2010.

[17] Abdul Basit, Saad Bin Qaisar, Hamid Rasool Syed, and Mudassar Ali. Sdn orchestration for next generation inter-networking: A multipath forwarding approach. *IEEE Access*, 2017.

[18] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, et al.

Onos: towards an open, distributed sdn os. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[19] Mark Berman, Jeffrey S. Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. GENI: A federated testbed for innovative network experiments. 61:5–23.

[20] Mark Berman, Jeffrey S Chase, Lawrence Landweber, Akihiro Nakao, Max Ott, Dipankar Raychaudhuri, Robert Ricci, and Ivan Seskar. Geni: A federated testbed for innovative network experiments. *Computer Networks*, 61:5–23, 2014.

[21] Monowar H Bhuyan, DK Bhattacharyya, and Jugal K Kalita. An empirical evaluation of information metrics for low-rate and high-rate ddos attack detection. *Pattern Recognition Letters*, 51:1–7, 2015.

[22] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and others. P4: Programming protocol-independent packet processors. 44(3):87–95.

[23] Kevin RB Butler, Toni R. Farley, Patrick McDaniel, and Jennifer Rexford. A survey of BGP security issues and solutions. 98(1):100–122.

[24] Ignacio Castro, Juan Camilo Cardona, Sergey Gorinsky, and Pierre Francois. Remote peering: More peering without internet flattening. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 185–198. ACM, 2014.

[25] Walter Cerroni, Chiara Buratti, Simone Cerboni, Gianluca Davoli, Chiara Contoli, Francesco Foresta, Franco Callegati, and Roberto Verdone. Intent-based management and orchestration of heterogeneous openflow/iot sdn domains. *NetSoft17*, 2017.

[26] Nikolaos Chatzis, Georgios Smaragdakis, Anja Feldmann, and Walter Willinger. There is more to ixps than meets the eye. *ACM SIGCOMM Computer Communication Review*, 43(5):19–28, 2013.

[27] Mosharaf Chowdhury, Fady Samuel, and Raouf Boutaba. Polyvine: policy-based virtual network embedding across multiple domains. In *Proceedings of the second ACM SIGCOMM workshop on Virtualized infrastructure systems and architectures*, pages 49–56. ACM, 2010.

[28] NM Mosharaf Kabir Chowdhury and Raouf Boutaba. Network virtualization: state of the art and research challenges. *IEEE Communications magazine*, 47(7):20–26, 2009.

[29] NM Mosharaf Kabir Chowdhury, Muntasir Raihan Rahman, and Raouf Boutaba. Virtual network embedding with coordinated node and link mapping. In *INFOCOM 2009, IEEE*, pages 783–791. IEEE, 2009.

[30] Rami Cohen, Katherine Barabash, Benny Rochwerger, Liran Schour, Daniel Crisan, Robert Birke, Cyriel Minkenberg, Mitchell Gusat, Renato Recio, and Vinit Jain. An intent-based approach for network virtualization. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 42–50. IEEE, 2013.

[31] Lin Cui, Fung Po Tso, and Weijia Jia. Heterogeneous network policy enforcement in data centers. 2017.

[32] Lin Cui, Fung Po Tso, Dimitrios P Pezaros, Weijia Jia, and Wei Zhao. Plan: Joint policy-and network-aware vm management for cloud data centers. *IEEE Transactions on Parallel and Distributed Systems*, 28(4):1163–1175, 2017.

[33] Amogh Dhamdhere and Constantine Dovrolis. The internet is flat: modeling the transition from a transit hierarchy to a peering mesh. In *Proceedings of the 6th International COnference*, page 21. ACM, 2010.

[34] David Dietrich, Ahmed Abujoda, Amr Rizk, and Panagiotis Papadimitriou. Multi-provider service chain embedding with nestor. *IEEE Transactions on Network and Service Management*, 14(1):91–105, 2017.

[35] Nick Feamster. Revealing utilization at internet interconnection points. 2016.

[36] Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, Ron Hutchins, Dave Levin, and Josh Bailey. Sdx: A software defined internet exchange. page 1.

[37] Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, Ron Hutchins, Dave Levin, and Josh Bailey. Sdx: A software defined internet exchange. *Open Networking Summit*, page 1, 2013.

[38] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. In *ACM Sigplan Notices*, volume 46, pages 279–291. ACM, 2011.

[39] Lixin Gao and Jennifer Rexford. Stable internet routing without global coordination. *IEEE/ACM Transactions on Networking (TON)*, 9(6):681–692, 2001.

[40] Vasileios Giotsas, Matthew Luckie, Bradley Huffaker, et al. Inferring complex as relationships. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 23–30. ACM, 2014.

[41] Simon Godik, Anne Anderson, Bill Parducci, P. Humenn, and S. Vajjhala. OASIS eXtensible access control 2 markup language (XACML) 3.

[42] Enrico Gregori, Alessandro Improta, Luciano Lenzini, and Chiara Orsini. The impact of ixps on the as-level topology structure of the internet. *Computer Communications*, 34(1):68–82, 2011.

[43] Arpit Gupta, Nick Feamster, and Laurent Vanbever. Authorizing network control at software defined internet exchange points.

[44] Arpit Gupta, Nick Feamster, and Laurent Vanbever. Authorizing network control at software defined internet exchange points. In *Proceedings of the Symposium on SDN Research*, page 16. ACM, 2016.

[45] Arpit Gupta, Robert MacDavid, Rudiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Laurent Vanbever. An industrial-scale software defined internet exchange point. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 1–14.

[46] Arpit Gupta, Robert MacDavid, Rudiger Birkner, Marco Canini, Nick Feamster, Jennifer Rexford, and Laurent Vanbever. An industrial-scale software defined internet exchange point. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 1–14, 2016.

[47] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P. Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. SDX: a software defined internet exchange. 44(4):551–562.

[48] Arpit Gupta, Laurent Vanbever, Muhammad Shahbaz, Sean P Donovan, Brandon Schlinker, Nick Feamster, Jennifer Rexford, Scott Shenker, Russ Clark, and Ethan Katz-Bassett. Sdx: A software defined internet exchange. *ACM SIGCOMM Computer Communication Review*, 44(4):551–562, 2015.

[49] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 19–24. ACM, 2012.

[50] Lorin Hochstein and Rene Moser. *Ansible: Up and Running: Automating Configuration Management and Deployment the Easy Way.* " O'Reilly Media, Inc.", 2017.

[51] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 15–26. ACM, 2013.

[52] DPDK Intel. Intel dpdk, programmers guide, 2014.

[53] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4):3–14, 2013.

[54] C Janz. Intent nbi–definition and principles. *Open Networking Foundation, Version*, 2, 2015.

[55] Joon-Myung Kang, Hadi Bannazadeh, and Alberto Leon-Garcia. SAVI testbed: Control and management of converged virtual ICT resources. In *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, pages 664–667. IEEE.

[56] Joon-Myung Kang, Hadi Bannazadeh, and Alberto Leon-Garcia. Savi testbed: Control and management of converged virtual ict resources. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 664–667. IEEE, 2013.

[57] Joon-Myung Kang, Hadi Bannazadeh, Hesam Rahimi, Thomas Lin, Mohammad Faraji, and Alberto Leon-Garcia. Software-defined infrastructure and the future central office. In *2013 IEEE International Conference on Communications Workshops (ICC)*, pages 225–229. IEEE.

[58] Joon-Myung Kang, Hadi Bannazadeh, Hesam Rahimi, Thomas Lin, Mohammad Faraji, and Alberto Leon-Garcia. Software-defined infrastructure and the future central office. In *Communications Workshops (ICC), 2013 IEEE International Conference on*, pages 225–229. IEEE, 2013.

[59] Joon-Myung Kang, Thomas Lin, Hadi Bannazadeh, and Alberto Leon-Garcia. Software-defined infrastructure and the savi testbed. In *Testbeds and Research Infrastructure: Development of Networks and Communities*, pages 3–13. Springer, 2014.

[60] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the one big switch abstraction in software-defined networks. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 13–24. ACM, 2013.

[61] Reyhaneh Karimazad and Ahmad Faraahi. An anomaly-based method for ddos attacks detection using rbf neural networks. In *Proceedings of the International Conference on Network and Electronics Engineering*, pages 16–18, 2011.

[62] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126.

[63] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russell J Clark. Kinetic: Verifiable dynamic network control. In *NSDI*, pages 59–72, 2015.

[64] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M Frans Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[65] Bikash Koley. The google zero touch network. 2016.

[66] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, et al. Onix: A distributed control platform for large-scale production networks. In *OSDI*, volume 10, pages 1–6, 2010.

[67] Vasileios Kotronis, Adrian Gämperli, and Xenofontas Dimitropoulos. Routing centralization across domains via sdn: A model and emulation framework for bgp evolution. *Computer Networks*, 92:227–239, 2015.

[68] Vasileios Kotronis, Rowan Klöti, Matthias Rost, Panagiotis Georgopoulos, Bernhard Ager, Stefan Schmid, and Xenofontas Dimitropoulos. Stitching inter-domain paths over ixps. In *Proceedings of the Symposium on SDN Research*, page 17. ACM, 2016.

[69] Alok Kumar, Sushant Jain, Uday Naik, Anand Raghuraman, Nikhil Kasinadhuni, Enrique Cauich Zermeno, C Stephen Gunn, Jing Ai, Björn Carlin, Mihai Amarandei-Stavila, et al. Bwe: Flexible, hierarchical bandwidth allocation for wan distributed computing. In *ACM SIGCOMM Computer Communication Review*, volume 45, pages 1–14. ACM, 2015.

[70] Josh Kunz, Christopher Becker, Mohamed Jamshidy, Sneha Kasera, Robert Ricci, and Jacobus Van der Merwe. OpenEdge: A dynamic and secure open service edge network. In *IEEE/IFIP NOMS*.

[71] Josh Kunz, Christopher Becker, Mohamed Jamshidy, Sneha Kasera, Robert Ricci, and Jacobus Van der Merwe. Openedge: A dynamic and secure open service edge network. In *IEEE/IFIP NOMS*, 2016.

[72] Craig Labovitz, Scott Iekel-Johnson, Danny McPherson, Jon Oberheide, and Farnam Jahanian. Internet inter-domain traffic. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 75–86. ACM, 2010.

[73] Dong Lin, David Hui, Weijie Wu, Tingwei Liu, Yating Yang, Yi Wang, John CS Lui, Gong Zhang, and Yingtao Li. On the feasibility of inter-domain routing via a small broker set. In *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*, pages 2031–2038. IEEE, 2017.

[74] Thomas Lin, Joon-Myung Kang, Hadi Bannazadeh, and Alberto Leon-Garcia. Enabling sdn applications on software-defined infrastructure. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–7. IEEE, 2014.

[75] Changbin Liu, Yun Mao, Jacobus Van der Merwe, and Mary Fernandez. Cloud resource orchestration: A data-centric approach. In *Proceedings of the biennial Conference on Innovative Data Systems Research (CIDR)*, pages 1–8. Citeseer, 2011.

[76] Xinlei Ma and Yonghong Chen. Ddos detection method based on chaos analysis of network traffic entropy. *IEEE Communications Letters*, 18(1):114–117, 2014.

[77] J Mambretti, J Chen, and F Yeh. Software-defined network exchanges (sdxs) and infrastructure (sdi): Emerging innovations in sdn and sdi interdomain multilayer services and capabilities. In *Science and Technology Conference (Modern Networking Technologies)(MoNeTeC), 2014 International*, pages 1–6. IEEE, 2014.

[78] KB Manayya. Constrained shortest path first. 2010.

[79] James McCauley, Zhi Liu, Aurojit Panda, Teemu Koponen, Barath Raghavan, Jennifer Rexford, and Scott Shenker. Recursive sdn for carrier networks. *ACM SIGCOMM Computer Communication Review*, 46(4):1–7, 2016.

[80] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. 38(2):69–74.

[81] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[82] Jan Medved, Robert Varga, Anton Tkacik, and Ken Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a*, pages 1–6. IEEE, 2014.

[83] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing software defined networks. In *NSDI*, volume 13, pages 1–13, 2013.

[84] Hung Xuan Nguyen, Michael R Webb, and Sanjeev Naguleswaran. Achieving policy defined networking for military operations. In *Military Communications and Information Systems Conference (MilCIS), 2016*, pages 1–6. IEEE, 2016.

[85] Aurojit Panda, James Murphy McCauley, Amin Tootoonchian, Justine Sherry, Teemu Koponen, Syliva Ratnasamy, and Scott Shenker. Open network interfaces for carrier networks. 46(1):5–11.

[86] Aurojit Panda, James Murphy McCauley, Amin Tootoonchian, Justine Sherry, Teemu Koponen, Syliva Ratnasamy, and Scott Shenker. Open network interfaces

for carrier networks. *ACM SIGCOMM Computer Communication Review*, 46(1):5–11, 2016.

[87] B. Park, T. Lin, H. Bannazadeh, and A. Leon-Garcia. Janus: Design of a software-defined infrastructure manager and its network control architecture. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 93–97, June 2016.

[88] Byungchul Park, Thomas Lin, Hadi Bannazadeh, and Alberto Leon-Garcia. Janus: design of a software-defined infrastructure manager and its network control architecture. In *2016 IEEE NetSoft Conference and Workshops (NetSoft)*, pages 93–97. IEEE.

[89] Przemyslaw Pawluk, Bradley Simmons, Michael Smit, Marin Litoiu, and Serge Mankovski. Introducing stratos: A cloud broker service. In *2012 IEEE Fifth International Conference on Cloud Computing*, pages 891–898. IEEE, 2012.

[90] Kevin Phemius, Mathieu Bouet, and Jérémie Leguay. Disco: Distributed multi-domain sdn controllers. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–4. IEEE, 2014.

[91] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. Pga: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM Computer Communication Review*, 45(4):29–42, 2015.

[92] Haiyang Qian, Xin Huang, and Ci Chen. Swan: End-to-end orchestration for cloud network and wan. In *Cloud Networking (CloudNet), 2013 IEEE 2nd International Conference on*, pages 236–242. IEEE, 2013.

[93] Muntasir Raihan Rahman, Issam Aib, and Raouf Boutaba. Survivable virtual network embedding. In *International Conference on Research in Networking*, pages 40–52. Springer, 2010.

[94] Dinesha Ranathunga, Matthew Roughan, Phil Kernick, and Nick Falkner. The mathematical foundations for mapping policies to network devices. In *SECRYPT*, pages 197–206, 2016.

[95] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular sdn programming with pyretic. *Technical Reprot of USENIX*, 2013.

[96] Philipp Richter, Georgios Smaragdakis, Anja Feldmann, Nikolaos Chatzis, Jan Boettger, and Walter Willinger. Peering at peerings: On the role of IXP route servers. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 31–44. ACM.

[97] Philipp Richter, Georgios Smaragdakis, Anja Feldmann, Nikolaos Chatzis, Jan Boettger, and Walter Willinger. Peering at peerings: On the role of ixp route servers. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 31–44. ACM, 2014.

[98] M Rouse. Docker swarm, https://docs.docker.com/engine/swarm/, 2016.

[99] SDN Ryu. Ryu sdn framework, https://osrg.github.io/ryu/, 2016.

[100] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. OpenStack: toward an open-source solution for cloud computing. 55(3).

[101] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3), 2012.

[102] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network

processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[103] Stavros N Shiaeles, Vasilios Katos, Alexandros S Karakos, and Basil K Papadopoulos. Real time ddos detection using fuzzy estimators. *computers & security*, 31(6):782–790, 2012.

[104] Thomas Soenen, Sahel Sahhaf, Wouter Tavernier, Pontus Sköldström, Didier Colle, and Mario Pickavet. A model to select the right infrastructure abstraction for service function chaining. In *Network Function Virtualization and Software Defined Networks (NFV-SDN), IEEE Conference on*, pages 233–239. IEEE, 2016.

[105] Jonathan Stringer, Dean Pemberton, Qiang Fu, Christopher Lorier, Robert Nelson, James Bailey, Carlos NA Corrêa, and Christian Esteve Rothenberg. Cardigan: Sdn distributed routing fabric going live at an internet exchange. In *Computers and Communication (ISCC), 2014 IEEE Symposium on*, pages 1–7. IEEE, 2014.

[106] Jonathan Philip Stringer, Qiang Fu, Christopher Lorier, Richard Nelson, and Christian Esteve Rothenberg. Cardigan: Deploying a distributed routing fabric. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 169–170. ACM.

[107] Tejas Subramanya, Roberto Riggio, and Tinku Rasheed. Intent-based mobile backhauling for 5g networks. In *Network and Service Management (CNSM), 2016 12th International Conference on*, pages 348–352. IEEE, 2016.

[108] Andreas Voellmy and Paul Hudak. Nettle: Taking the sting out of programming network routers. *Practical Aspects of Declarative Languages*, pages 235–249, 2011.

[109] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: a language for high-level reactive network control. In *Proceedings of the first workshop on Hot topics in software defined networks*, pages 43–48. ACM, 2012.

[110] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying sdn programming using algorithmic policies. *ACM SIGCOMM Computer Communication Review*, 43(4):87–98, 2013.

[111] Sami Yangui, Iain-James Marshall, Jean-Pierre Laisne, and Samir Tata. Compatibleone: The open source cloud broker. *Journal of Grid Computing*, 12(1):93–109, 2014.

[112] Soheil Hassas Yeganeh and Yashar Ganjali. Beehive: Simple distributed programming in software-defined networks. In *Proceedings of the Symposium on SDN Research*, page 4. ACM, 2016.

[113] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: substrate support for path splitting and migration. *ACM SIGCOMM Computer Communication Review*, 38(2):17–29, 2008.