ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Computer and Control Engineering (29^{th}cycle)

Network Infrastructures for Highly Distributed Cloud-Computing

By

Francesco Lucrezia

******

Supervisor(s):
Prof. Guido Marchetto

Doctoral Examination Committee:
Prof. Flavio Esposito
Prof. Barbara Martini
Dr. Domenico Siracusa
Dr. Balazs Sonkoly

Politecnico di Torino
2018

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Francesco Lucrezia
2018

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

To my parents,
my partner in life Cinzia and
my newborn son Alessandro

Acknowledgements

Thanks to my supervisor Guido who has been a precious guide in this - sometimes suffered - journey called PhD.

Thanks to Fulvio, my second advisor who has always been capable of giving me the right advises.

Thanks to my family who always supports me, in the good and in the bad.

Thanks to all my colleagues and friends.

Thanks,
Francesco

Abstract

Software-Defined-Network (SDN) is emerging as a solid opportunity for the Network Service Providers (NSP) to reduce costs while at the same time providing better and/or new services. The possibility to flexibly manage and configure highly-available and scalable network services through data model abstractions and easy-to-consume APIs is attractive and the adoption of such technologies is gaining momentum. At the same time, NSPs are planning to innovate their infrastructures through a process of network softwarisation and programmability. The SDN paradigm aims at improving the design, configuration, maintenance and service provisioning agility of the network through a centralised software control. This can be easily achievable in local area networks, typical of data-centers, where the benefits of having programmable access to the entire network is not restricted by latency between the network devices and the SDN controller which is reasonably located in the same LAN of the data path nodes. In Wide Area Networks (WAN), instead, a centralised control plane limits the speed of responsiveness in reaction to time-constrained network events due to unavoidable latencies caused by physical distances. Moreover, an end-to-end control shall involve the participation of multiple, domain-specific, controllers: access devices, data-center fabrics and backbone networks have very different characteristics and their control-plane could hardly coexist in a single centralised entity, unless of very complex solutions which inevitably lead to software bugs, inconsistent states and performance issues.

In recent years, the idea to exploit SDN for WAN infrastructures to connect multiple sites together has spread in both the scientific community and the industry. The former has produced interesting results in terms of framework proposals, complexity and performance analysis for network resource allocation schemes and open-source proof of concept prototypes targeting SDN architectures spanning multiple technological and administrative domains. On the other hand, much of the

work still remains confined to the academy mainly because based on pure Openflow prototype implementation, networks emulated on a single general-purpose machine or on simulations proving algorithms effectiveness. The industry has made SDN a reality via closed-source systems, running on single administrative domain networks with little if no diversification of access and backbone devices.

In this dissertation we present our contributions to the design and the implementation of SDN architectures for the control plane of WAN infrastructures. In particular, we studied and prototyped two SDN platforms to build a programmable, intent-based, control-plane suitable for the today highly distributed cloud infrastructures. Our main contributions are: *(i)* an holistic and architectural description of a distributed SDN control-plane for end-end QoS provisioning; we compare the legacy IntServ RSVP protocol with a novel approach for prioritising application-sensitive flows via centralised vantage points. It is based on a peer-to-peer architecture and could so be suitable for the inter-authoritative domains scenario. *(ii)* An open-source platform based on a two-layer hierarchy of network controllers designed to provision end-to-end connectivity in real networks composed by heterogeneous devices and links within a single authoritative domain. This platform has been integrated in CORD, an open-source project whose goal is to bring data-center economics and cloud agility to the NSP central office infrastructures, combining NFV (Network Function Virtualization), SDN and the elasticity of commodity clouds. Our platform enables the provisioning of connectivity services between multiple CORD sites, up to the customer premises. Thus our system and software contributions in SDN has been combined with a NFV infrastructure for network service automation and orchestration.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

New IT system models are driven by the composition of software services exposed to consumer entities that can benefit from the functionalities such services export transparently, in the most technology and protocol independent way. Composition of very complex system involves synergies of distributed infrastructures of network, storage and compute elements with the objective of achieving very large scale of provisioning and automation.

In this highly dynamic world of Cloud-Computing and Everything-as-a-Service, Network Service Providers (NSPs) are facing important challenges to keep up with the changing, striving to innovate their network infrastructures at the pace content providers do with their services whose proliferation leverages on high-volume standard servers (e.g., x86-based blades), computing/storage virtualisation and distributed applications running on a massive number of heterogeneous devices. Digital contents are consumed by smart phones and sophisticated terminal stations that continuously evolve together with the applications they host. Interestingly enough, the evolution of the Over-The-Top (OTT) services is mainly happening without the aid of the NSPs, within the best-effort data traffic channel in the access networks. OTT providers are widely exploiting the cloud infrastructure to scale in/out without worrying about the infrastructure itself. Connectivity services spanning Wide-Area-Networks (WANs), metro areas and geographically distributed data-centers, instead, are still mainly statically provisioned. This is due to a variety of reasons: standards, protocols and interoperability heavily impact the adoption of dynamic and automated systems; networking problems are complex in nature and networking solutions are

strongly constrained by hardware equipment. Moreover, network management and control tools have been always considered matter for device manufacturers, which offer their solution under closed source code and proprietary platforms inevitably leading to vendor lock-ins and degradation in flexibility, freedom, security, revenues and accountability which are inherent benefits in the adoption of open-source software.

A huge effort is being placed by the Telcos, followed by device vendors, in supporting and financing large open-source projects dedicated to the control plane of the network infrastructures exploiting SDN and NFV [1–5]. In fact these new technologies have radically changed the concept of network architectures decoupling the control plane from the data packet plane, introducing new ways of exploiting the functionalities of network equipments via virtualisation. This enables the network modelling to be dynamic, flexible and scalable, able to guarantee a simpler management and a faster speed of development and deployment of new services and technologies, reducing issues and limitations due to the staticity of hardware components.

Although the early SDN products primarily focus on automation and orchestration within a data-center, the more mature SDN solutions, such as those from Google and Amazon, are designed to take intercloud federation across WAN and software-defined WAN into consideration ([6]). A common factor of innovation growth of these companies is their ability to potentially exploit the benefits of having programmable access to the entire network stack, from the lowest-level hardware to the highest-level software elements for the control of the east-west (data-center to data-center) traffic; rather than being forced to create compromised solutions based on available insertion points, they can design end-to-end secure and performant solutions, by coordinating across the network stack. In this regard, NSPs are in a more disadvantaged position because of vendors, protocols and standards bonds. Many SDN platforms have been conceived and developed by the scientific community, but either they target base platform for *application development environment* to build network applications on top of it, without providing solutions to actual networking problems (e.g. [7, 8]), or they present a broad overview of SDN platforms for distributed and/or multi domain networks focusing on the distribution mechanisms of the network state in the control plane, taking into account solely the forwarding part via the Openflow protocol, leaving aside the prescriptions to implement real network

services on such platforms [9–13]; some proposals focus on theoretical modeling and placement problems for distributed control plane in SDN ([14–18]).

In this context are located the activities of the PhD defining this dissertation. We consider concrete networking problems, specifically the per-flow QoS provisioning and the Carrier-Ethernet virtual circuits provisioning, to present two SDN platforms aimed at overcoming the limitations of a single logically centralised SDN controller when dealing with geographical and heterogeneous networks. The main focus of the studies have been on the architectural perspective and the software design of such platforms. In particular, part of the thesis will cover an SDN platform based on ONOS (Chapter 2), a network controller that targets scalability, high availability, high performance and abstractions to make it easy to create networking apps and services. Thus we leverage on ONOS high availability for the state distribution among a cluster of controller instances. The platform enables:

- A single administrative domain network to be divided into multiple regions piloted by different network controllers to decrease event-to-response delays, increase the overall robustness to geographical network faults and distribute the load among the cluster of network controllers (Chapter 3). An event can be something happening in the data path for which there is a feedback control implemented in ONOS (e.g. port state change, device or link discovery) or something generated in the controller surface, such as a policy update.
- The communication with other administrative domains leveraging on an east-west interface to ensure full control of services and events between domains and enforce configuration policies between domains (Chapters 3, 4).
- Dynamic setup of end-to-end priority paths with guaranteed bandwidth for data packet flows over WANs and/or geographically distributed data-centers and users (Chapter 4). We describe how the flow-based QoS model conceived with RSVP [19] can be revised and implemented in a distributed SDN platform to bring end-to-end QoS provisioning over WAN on a pre-application basis which is still missing in today networks. We present an algorithm for admission control implemented on top of ONOS.

Chapter 5 will focus on the rationale and the insights behind the design and the development of a platform that is part of a wider open-source project: CORD [1];

it was conceived upon the need of an open-source reference platform for connectivity service and bandwidth on-demand, end-to-end control over an heterogeneous network and to enable a unified view/orchestration of the access and transport resources. It enables on-demand mobility and migration of services such as VMs and storage between geographically distributed data-centers by unifying intra and inter data-center network control and management. The ambition of the project is to shift much of the concepts embraced by the SDN paradigm from the academic to the industry world, and from the single data-center premise to geographic networks with particular emphasis for dedicated connectivity services for enterprises such as Carrier Ethernet circuits provisioning on-demand. The system is able to drive a network composed of different commercial devices, from whitebox Openflow switches to proprietary Netconf-enabled CPE devices and disaggregated ROADMs. We believe that being able to properly control real physical devices is essential for a meaningful system and software wise contribution to the SDN control-plane infrastructures, because ultimately, it is the hardware that processes and transmits bits over the wire. This is one major contribution and distinction factor from related proposals, to the best of our knowledge, there is no open-source system able and tested to drive a chain of physical devices from the very access customer premises to the backbone optical transport switches. And it can be easily extended to other different domain technologies such su G/MPLS networks. Moreover, being a sub-project of CORD, it is seamlessly integrated with NVF service chaining provided in CORD. In this open-source project we designed and developed the hierarchical platform of network controllers using ONOS [20].

Chapter 2

ONOS: Open Network Operating System

2.1 ONOS Overview

ONOS (Open Networking Operating System) [7] is a distributed network controller created and maintained by the ONF team [21]. It is an open-source joint community effort with substantial contribution from various partners including AT&T, NEC, Huaweii and Verizon [2]. Most of our contributions have resolved into prototype applications running on it. In this chapter we give a broad description of it and then we will further dig into its architecture in later discussion.

ONOS is an SDN operating system for Service Providers, that is targeting scalability, high availability, high performance and abstractions to make it easy to create apps and services. It implements a distributed architecture using RAFT [22] in which multiple controller instances share multiple distributed data stores with different level of consistency. The entire data plane is managed simultaneously by the whole cluster of instances. However, for each device a single controller instance acts as a master, while the others are ready to step in if a failure occurs. With these mechanisms in place, ONOS achieves scalability and resiliency. Figure 2.1 shows the ONOS internal architecture within a cluster of four instances. ONOS is based on software modules managed by the Apache Karaf suite [23], a set of java OSGi based runtime and applications. It provides a container into which various component can

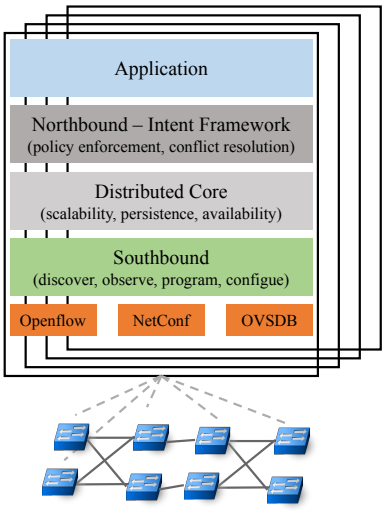


Fig. 2.1 ONOS distributed architecture

be deployed, installed, upgraded, started and stopped at runtime, without interfering other components. The southbound modules manage the physical topology, react to network events and program/configure the devices leveraging on different protocols. The distributed core is responsible to maintain coherent information, to elect the master controller for each network portion and to share information with the adjacent layers. In case of a failure in the data path (switch, link or port down), an ONOS instance becomes aware of the event through the southbound modules, computes alternative paths for all the traffic crossing the failed element, and notifies them to the distributed core; then, each master controller configures accordingly its portion of the network. The northbound subsystem offers an abstraction of the network and the interface for applications to interact and program the NOS. Finally, the Application layer offers a container in which third-party applications can be deployed. Applications on top of ONOS can benefit of the Intent Framework. An intent is an abstraction used by applications to specify their high-level desires in form of policies. The ONOS core accepts the intent specifications and translates them into actionable operations on the network environment. These actions are carried out by the intent installation process, such as flow rules being installed on a switch, or optical lambdas (wavelengths) being reserved.

2.2 Control vs Configuration

The basic idea of SDN is to achieve dynamic control over forwarding plane behaviour from a logically centralised vantage point. Telco operators have been doing configuration and management for quite a long time, from a logical centralised vantage point and for very large network, but the increased demand for performance, agility and reliability due to the advent of cloud-based applications pushed the interest for new automated platform from human time-scales to machine time-scales and with reduced tolerance for control plane failures. Performance and scalability are important for configuration as well, but the time-scale is different w.r.t to control: configuration and management requires around 1000 operations per day, while control requires around 1000 if not 1000000 operations per second.

Operators want to create and sell customised services with agility and minimal human intervention and create automated ways to instantiate such network services. These services comprise both configuration and control of forwarding devices, (e.g. setting-up lambdas and Openflow rules, provision NFV service chains and steer traffic through them). For this reason operators need a resilient and scalable platform capable of both control and configuration.

ONOS was originally designed to be an SDN platform focusing solely on the control of the forwarding behaviour and for this reason it adopts an API-driven approach that stems from the forwarding rules' abstractions of the Openflow protocol [24]. Before ONOS, since 2008, multiple controllers were developed with this approach: Beacon [25], Floodlight[26], NOX [27], POX [28] and Ryu [29]. Each controller implemented Openflow as the sole southbound protocol towards the packet forwarding function, and provided access to its control plane functions through northbound REST APIs. Over the years, ONOS has introduced the support for other important protocols such as Netconf and Restconf to add configuration capabilities. ONOS is an API-driven platform because the control abstractions and APIs are semi-fixed, borne out of following the Openflow standard, while configuration abstractions are harder to fix, though standards exist, vendors want to expose their unique features (this explains the explosion of SNMP MIB variables). The pros of the API-driven approach are:

- Application developers are presented with a solid surface.

- The platform is not tied to a closed set of protocols: device drivers can use YANG/NETCONF, SNMP, REST, Openflow and any other to interact with physical devices.
- Applications portability and stable evolution is facilitated.

While the cons of are:

- Limit access to new or differentiating device features (unless API is sufficiently open-eneded).
- Enhancements and new features require development resources.

A pure configuration and management platform (see OpenDaylight [8]) would adopt the model-driven approach instead. It consists of a framework based on consistent relationships between (different) models, standardised mappings and patterns that enable model generation and, by extension, code/API generation from models. This generalisation can overlay any specific modelling language although YANG has emerged as the data modelling language for the networking domain. The model driven approach is being increasingly used in the networking domain to describe the functionality of network devices, services, policies and network APIs [30]. Here pros and cons are listed together given their intertwined nature:

- Code-generation avoids manual boilerplate code. Consideration must be given to versioning and to the impact of a model change on the (re)generated API.
- Applications have access to nuanced features, not limited by fixed APIs and they are presented with a fluid surface on the platform. But this implies that applications must be model-aware, that is, they must know the model semantics. This has a strong impact on application portability.
- Enhancements require fewer development resources since much code is auto generated by the models.

The ONOS releases used in our prototypes, 1.9.0 and 1.10.0, adopt an hybrid approach that is a combination of, and a compromise of, the stability given by the semi-fixed APIs used by the NB applications and the elasticity and rich capabilities

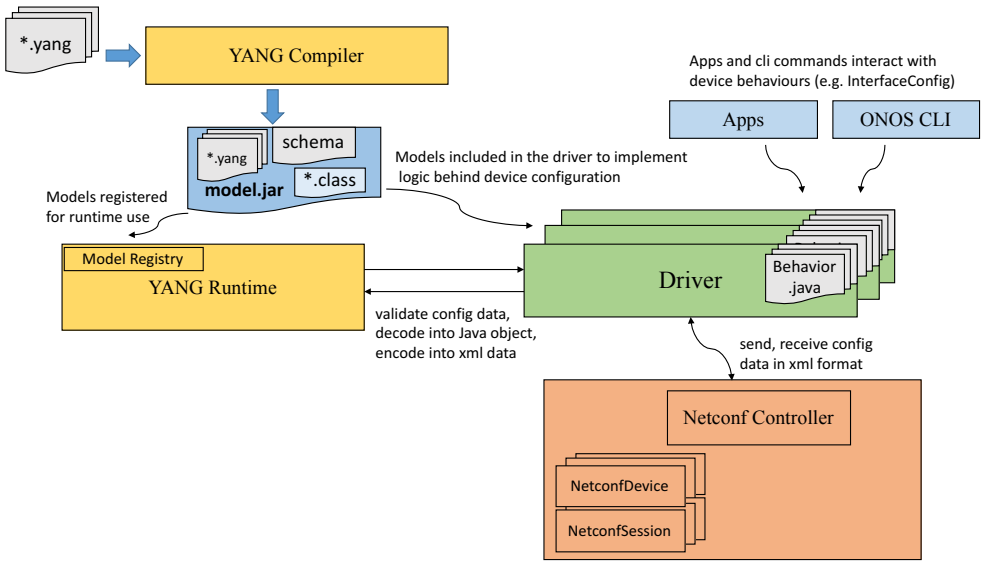


Fig. 2.2 Model-based device drivers implementation via Yang/Netconf

enabled by YANG/Netconf within the device drivers (Fig. 2.2). Micro-APIs abstracting many facets of configuration are expressed in ONOS by extending the Behaviour interface to implement effectively an API per feature (e.g. VXLAN, VLAN, Queues configuration). The implementation encapsulates specific logic and code of the exported capability and within it one can use whatever modelling language (YAML, YANG, SNMP schema etc.) to define semantics and syntax for interacting with the device. Apps and other ONOS subsystems instead are completely independent of such modelling languages and refers to the device capability through such Behaviour abstractions. A collection of behaviours describes the capabilities of a device in ONOS and they are accessible via class projections and casting, while macro APIs remain unchanged (stable) to the northbound applications.

In Chapter 4 we make use of the driver subsystem to implement the logic behind the QoS configuration of the virtual switch OpenVSwitch [31]. In Chapter 5 we use multiple ONOS drivers to control and configure the data path nodes. One of these devices is entirely described via YANG models. The drivers for the Openflow-enabled switches instead are implemented after the ONOS core APIs written in Java. The rest of our system contributions to ONOS touches the application, the core and the provider layers which are device agnostic.

Chapter 3

ICONA: A Peer-to-Peer Approach for Software Defined Wide Area Networks Using ONOS

Part of the work described in this chapter has been published in [32]

3.1 Motivation

As mentioned in the introduction, WAN networks are composed by a huge number of distributed network devices and terminal stations. Reliability, scalability and availability are among the major elements of attention expressed by Service and Cloud Providers. Existing deployments show that standard IP/MPLS networks natively offer fast recovery in case of failures. Their main limitation lies in the complexity of the distributed control plane, implemented in the forwarding devices. IP/MPLS networks fall short when it comes to designing and implementing new services that require changes to the distributed control protocols and service logic.

The SDN architecture, that splits data and control planes, simplifies and speeds up the introduction of new services, by moving the intelligence and most of network state from the physical devices to a logically centralised Network Operating System (NOS), also known as controller, in charge of all the forwarding decisions. It is also

clear, as described in the work of Heller et al. [5], that even if a single controller may suffice to guarantee round-trip latencies on the scale of a typical mesh restoration delays (200 msec), this is not enough for all network topologies. Furthermore, ensuring an adequate level of fault tolerance (i.e., avoiding excessive packet loss and session termination) can be guaranteed only if controllers are spaced apart in different locations of the network. A logical step towards robustness in SDN is to distribute the load of the control plane between entities, each taking care of a portion of the entire geographical network and each providing an east-west communication interface to enable programmability of the entire network. To guarantee the proper level of redundancy in the control plane, several distributed NOS architectures have been proposed in the last years: ONIX [33], Kandoo [13], HyperFlow [9] to name a few. Mainly, these architectures fall into two categories: *(i)* hierarchy of controllers and *(ii)* peer-to-peer interconnections between controllers. While the former gives adequate scalability for resources under the control of the same domain, the latter offers more benefits in case of a multi administrative domain solution, removing a top-level entity, possibly managed by a third party, controlling the interconnections between networks belonging to different providers.

In the ONOS architecture, a cluster of controllers shares a logically centralised network view: network resources are partitioned and controlled by different ONOS instances in the cluster. Resilience to faults is guaranteed by design, with automatic traffic rerouting in case of node or link failure. However, despite the distributed architecture, ONOS is designed to be placed in a single geographical location, because its distributed architecture requires negligible communication delays between instances. Given this consideration, we engineered an open-source ONOS application called ICONA (Inter Cluster Onos Network Application). ICONA is designed to work in a single administrative WAN network scenario, increasing the robustness to network faults by redounding ONOS clusters in several geographical locations and decreasing event-to-response delays, as well as in a multi administrative domain scenario. To better support the latter use-case, ICONA is based on a peer-to-peer architecture, and implements configuration policies between clusters (i.e., domains belonging to different owners), that ensure the full control of services and events between domains.

3.2 ICONA Architecture

ICONA is a new southbound ONOS *provider* that offers an east-west interface and a powerful abstraction layer to allow a single ONOS cluster to be interconnected with several other clusters, both in the same and in different administrative domains. *Providers* in ONOS are standalone ONOS applications based on OSGi components that can be dynamically activated and deactivated at runtime. The main purpose of *providers* is to abstract the configuration, control and management operations of a specific family of devices (e.g. OpenFlow, SNMP, Netconf, etc.). ONOS interacts with the underlying network with the help of these components. Being a *provider*, ICONA is completely transparent to the ONOS core systems and to other applications, thus offering the same functionalities of ONOS, but extended to a geographically distributed environment, including multiple administrative domains. From an application perspective, all the features offered by ONOS are then available in an multi administrative domain composed of several ICONA clusters. The main architectural goals of ICONA are to:

- *Enable east-west communication between ONOS clusters*. In a single-domain, this implies partitioning the Service Providers network into several geographical regions, each one managed by a different cluster of ONOS instances. The network architect can select the number of clusters and their geographical dimension depending on requirements (e.g., leveraging on some of the tools being suggested within the aforementioned work [5]), without losing any features offered by ONOS, neither worsening the system performance. In a multi-domain scenario, several ONOS clusters, belonging to different administrative domains, can exchange network services based on respective policies and network abstractions.

- *Provide an abstraction to ONOS*, able to: *(i)* abstract and communicate external topologies (i.e., devices, links and ports not directly managed by the local cluster), *(ii)* configure these external devices from local applications, leveraging on the Intent Framework and *(iii)* enforcing policies between clusters.

Clusters, policies and topology abstractions can be easily injected in ICONA through the ONOS configuration service. ICONA extrapolates the local topology from the ONOS core, abstracts it based on the configuration and finally exposes it to

remote clusters; likewise, it receives the external topologies from the remote clusters and notifies them to ONOS. In case of a multi domains, this external topology is exposed as a single big switch. Moreover, it takes care of reporting relevant updates to the remote clusters about changes affecting the abstracted topology by listening to events reported by the ONOS subsystems (e.g., devices, links, ports and edge hosts). The east-west communication between clusters is not bounded to a single peer-to-peer mechanism, but it allows different implementations, leveraging on the ICONA Southbound Interface (ISBI).

Figure 3.1a depicts two clusters, A and B, sharing their topologies through ICONA. In Figure 3.1b, *cluster A* exposes its 4 switches topology as a single big switch, that is abstracted and communicated to the local ONOS core by the ICONA provider of *cluster B*.

The communication between clusters relies on the ISBI interface, that can be implemented by different mechanisms. To make ICONA as flexible as possible, its structure is vertically split in two logical layers:

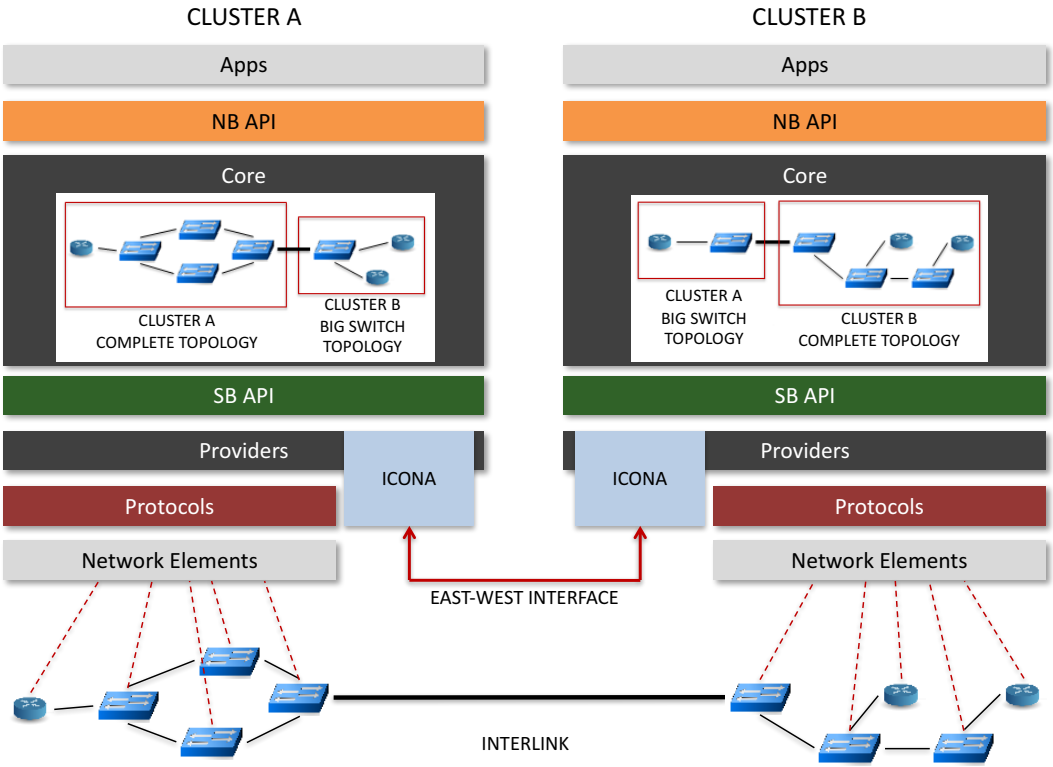
- the ICONA Provider which contains the main logic and lays between the ONOS core and the ISBIs.
- multiple ISBI drivers, each one tied to a specific for a communication mechanism. For each remote cluster, it's possible to specify a different mechanism, thus allowing several ISBI implementations to be used simultaneously.

This architecture allows to integrate several communication mechanisms, just by implementing the ISBI logic, without any modifications in the ICONA provider.

3.2.1 ICONA Provider

The provider contains the main logic behind ICONA, and performs various functionalities. As a **Topology Manager** (see 3.2.1), it builds an abstraction of the local topology to be exposed to remote clusters. Currently ICONA supports two topology abstractions: *BigSwitch* (single switch representing the entire network with edge ports) and *FullMesh* (network topology in which there is a direct link between all pairs of edge nodes). While the former shrinks the entire topology in a single switch, the latter builds a full virtual mesh topology between all the edge switches (e.g.

(a) ICONA as a peering provider



(b) Topology exchange

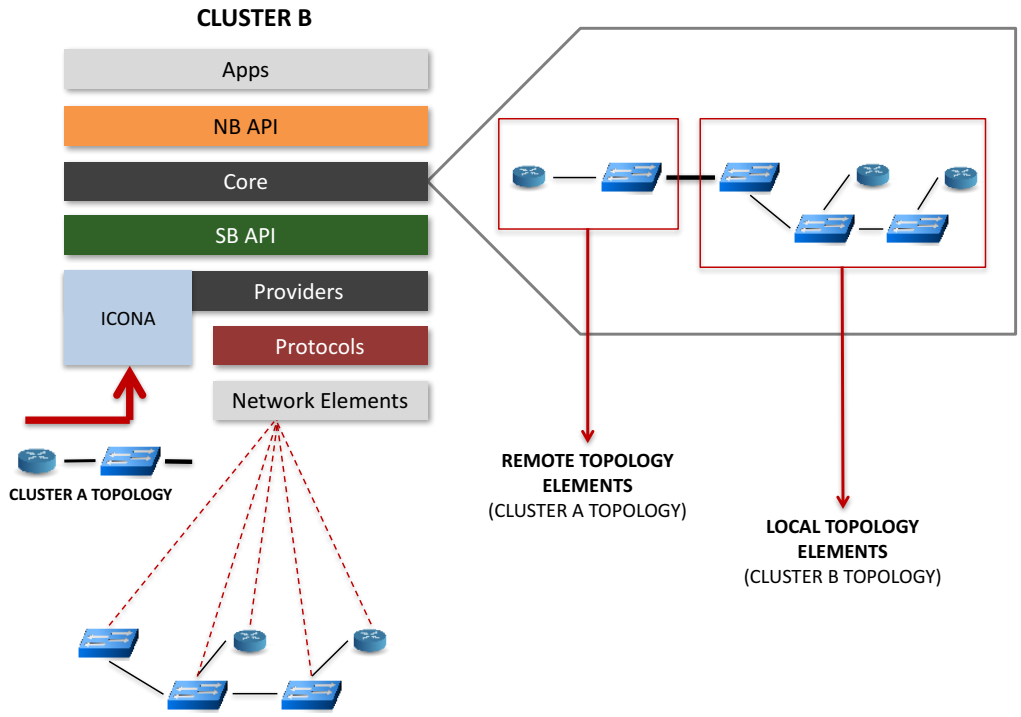


Fig. 3.1 ICONA topology abstraction.

the ones with edge ports) of the network. The provider, after receiveing remote topologies from the southbound mechanisms, notifies them to the local ONOS core, and reacts to network events which could reflect a change in the topology exposed to/from the remotes. As a **Service Manager** (see 3.2.1), it manages service requests coming from the local applications to remote clusters and vice versa. Finally, as a **Policy Manager** (see 3.2.1), it enforces configuration policies between clusters. The next sections detail the features offered by the three provider modules, depicted in Figure 3.3.

Topology Manager

The Topology Manager (TM) is responsible to *(i)* analyse the remote topologies and install them in ONOS with the relevant metric and annotations, and *(ii)* reacts to network events, both local and remote. Each TM shares with the other clusters an abstraction of the local topology, that may vary from cluster to cluster, based on the configuration. The topology is composed of:

- Inter-links (IL): links belonging to different ONOS clusters. Each IL is provided trough the ONOS configuration subsystem and it's tagged by some metrics, such as the link delay, available bandwidth and number of flows crossing the link.
- Virtual devices and intra-links: links within the local cluster/domain.
- End-points (EP): interconnection ports between the customer's gateway router/switch and the ONOS network.

Service Manager

The Service Manager (SM) is the ICONA component, that provides inter-cluster path computation whenever a northbound application, on-top of ONOS, requires connectivity between two or more EPs crossing multiple clusters. The SM intercepts an intent requests from the ONOS core targeting one or more virtual devices of the remote topology and sends it to the target remote cluster (Fig. 3.2). The recipient translates the remote intent request into a local intent request and submits it to the ONOS Core. The translation consists in resolving the mapping between the abstract

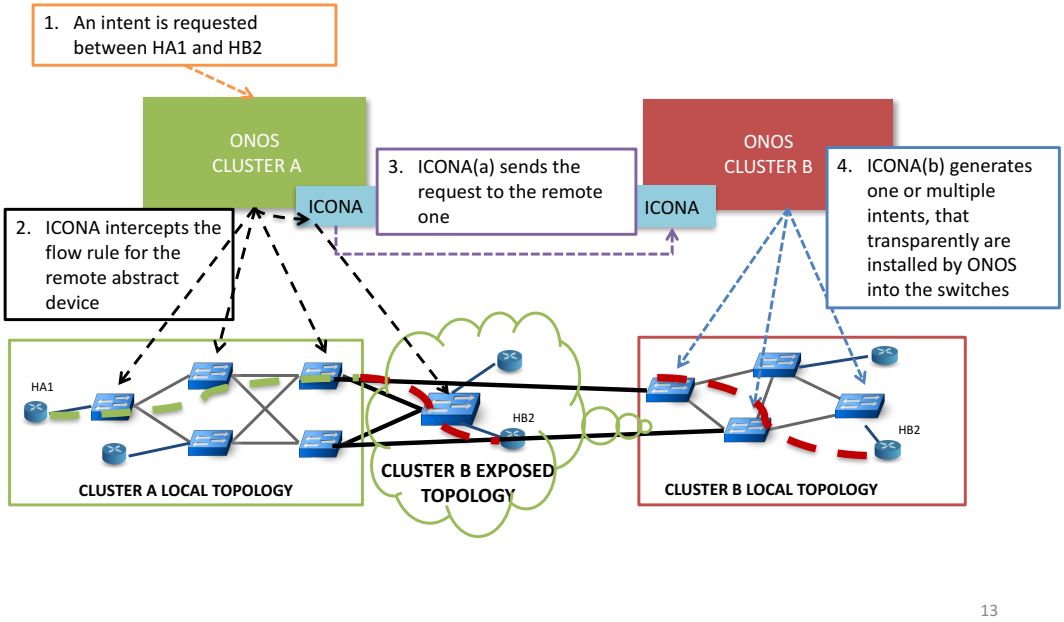


Fig. 3.2 Service request accomplished with ICONA.

ingress and egress points of the original request into local ingress and egress points of the underlying network. The evaluation on the feasibility of the intent installation is also performed, both in terms of policy and capabilities. If the request is accepted, the SM waits for the ONOS core to accomplish the task of installing the intent and then notifies the requester about the outcome.

Policy Manager

A multi administrative domain scenario is characterized by the presence of networks under control of different authorities. Usually, the mutual trust between these domains is limited to specific agreements, which identify a list of constraints to be applied at the edge of the network. To support such use-case, ICONA is policy-oriented and enforces, through configuration, the compliance to those agreements. Currently ICONA allows to set at runtime several parameters, such as *(i)* the external peering clusters information, *(ii)* the topology abstraction exposed to each domain, *(iii)* the list of EPs, *(iv)* the type and number of intents installable by a remote cluster,

(v) the ILs and their metrics (i.e., bandwidth, delay and type), *(vi)* the preferred path for specific classes of traffic (based on L2 and L3 fields). However, we are currently analysing the common design patterns of BGP policies, that are typically used by ISPs [34], to implement innovative policy mechanisms in the future releases.

3.2.2 ICONA Southbound Mechanisms

A southbound mechanism is an implementation of the communication system between clusters (Fig. 3.3). Basically, it's a software component in charge of translating the provider's requests into protocol-specific, network operations and the remote clusters messages into abstracted notifications via the ISBI. This component performs the exchange of the information with message encoding and decoding, and does not retain any system state except the status of the remote clusters. Currently ICONA supports two different mechanisms:

- BGP: an extension of the BGP protocol with a new Type-Length-Value (TLV) field to offload the communication system to an external router (e.g. Quagga based) and to use BGP as a pure transport protocol for data exchange.
- REST: a REST client/server peer-to-peer architecture has been implemented between clusters. The client is in charge of sending local topology elements and to request service installation, while the server is responsible to receive remote topology elements and service requests from the other clusters. We implemented a distributed architecture, in which every ICONA instance is responsible to interconnect the local cluster with a set of remote clusters and the communication is balanced among multiple end-points providing resiliency.

3.3 Evaluation

The purpose of the experimental tests described in this section is to compare ICONA with a standard ONOS setup, and evaluate the performances of the two solutions in an emulated environment. It is important to highlight that these evaluation has been achieved with a preliminary version of ICONA, working on an old ONOS version (Blackbird release). For these reasons, the results presented in this section should

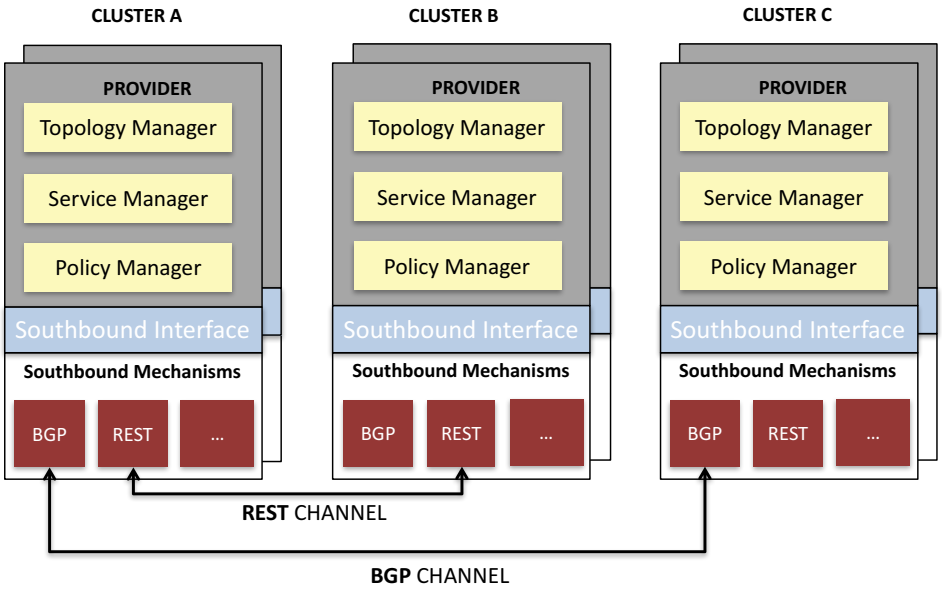


Fig. 3.3 High level view of the ICONA components.

not be considered as benchmark, but they can offer a comparison between the fully centralized solution versus the peer-to-peer architecture offered by ICONA.

The control plane is composed of several virtual machines, each configured to use 4 Intel Core i7-2600 CPUs @ 3.40GHz and 8GB of RAM. For the ONOS tests we used 8 instances, while with ICONA we created 2, 4 and 8 clusters, respectively with 8, 5 and 3 instances each. The data plane is emulated by Mininet [35] and Netem [36]: the former creates and manages the network based on OpenFlow 1.3, while the latter emulates the properties of wide area networks, introducing variable delays, throughput and packet loss. Both solutions (ONOS and ICONA) have been tested on top of "grid" networks and of the GÉANT [37] topology.

3.3.1 Reaction to Network Events

Grid network

The first measured performance metric is the overall latency of the system for updating the network state in response to events; examples include rerouting traffic in response to link failure or moving traffic in response to congestion. To evaluate how the system performs when the forwarding plane scales-out, few standard grid topologies (from 4*4 to 10*10) have been chosen, with a fixed link delay of 5ms (one-way) and the latency needed to reroute a certain number of installed paths when an inter-cluster link fails is compared between ONOS and ICONA with various clustering settings.

The total latency is defined as the amount of time that ONOS or ICONA requires to react to the failure. It is computed as the sum of: *(i)* the amount of time taken by the OpenFlow messages (`PORT_STATUS` and `FLOW_MOD`) to traverse the control network, *(ii)* the alternative path computation, *(iii)* the installation of new flows in the network devices and *(iv)* the deletion of the pre-existing flows. In particular, we have been running several simulations by installing 10^3 paths in the network and then causing failure of an inter-cluster link carrying al least 10^2 flows.

Figure 3.4 shows the latency (avg, min, max) required for the different case to execute the four tasks previously mentioned. Each test has been repeated 10^3 times. Despite the same mechanism used by ICONA to compute and install the new paths, the difference is mainly due to the following reasons: *(i)* each ICONA cluster

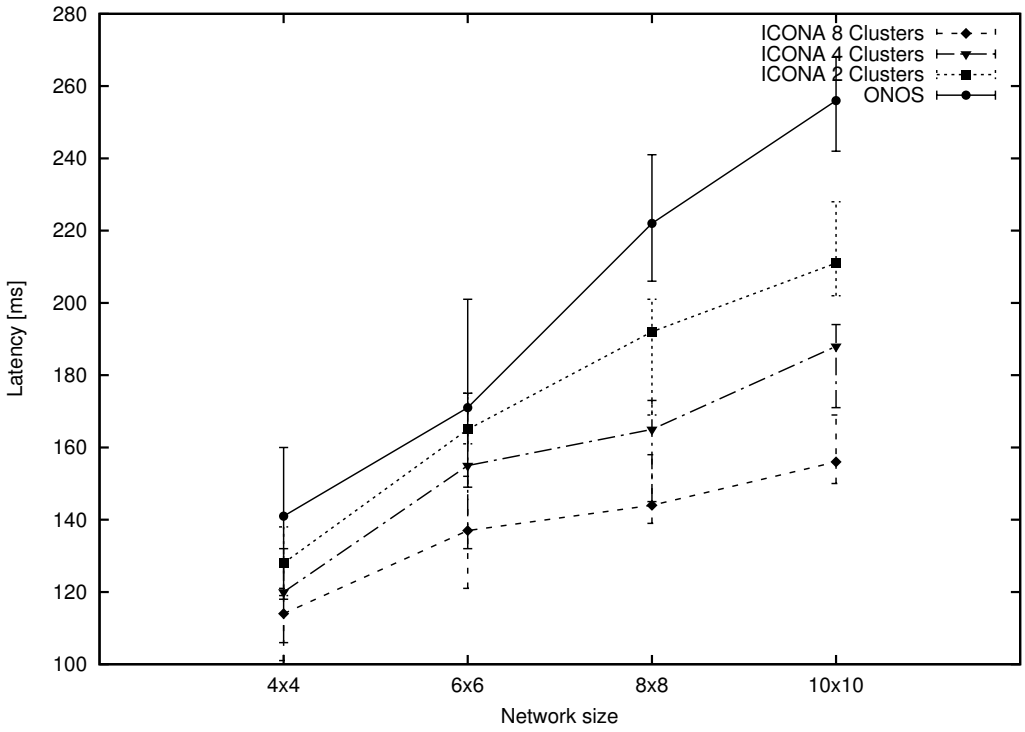


Fig. 3.4 Average, maximum and minimum latency to reroute 100 paths in case of link failure for ONOS and ICONA (2, 4 and 8 clusters)

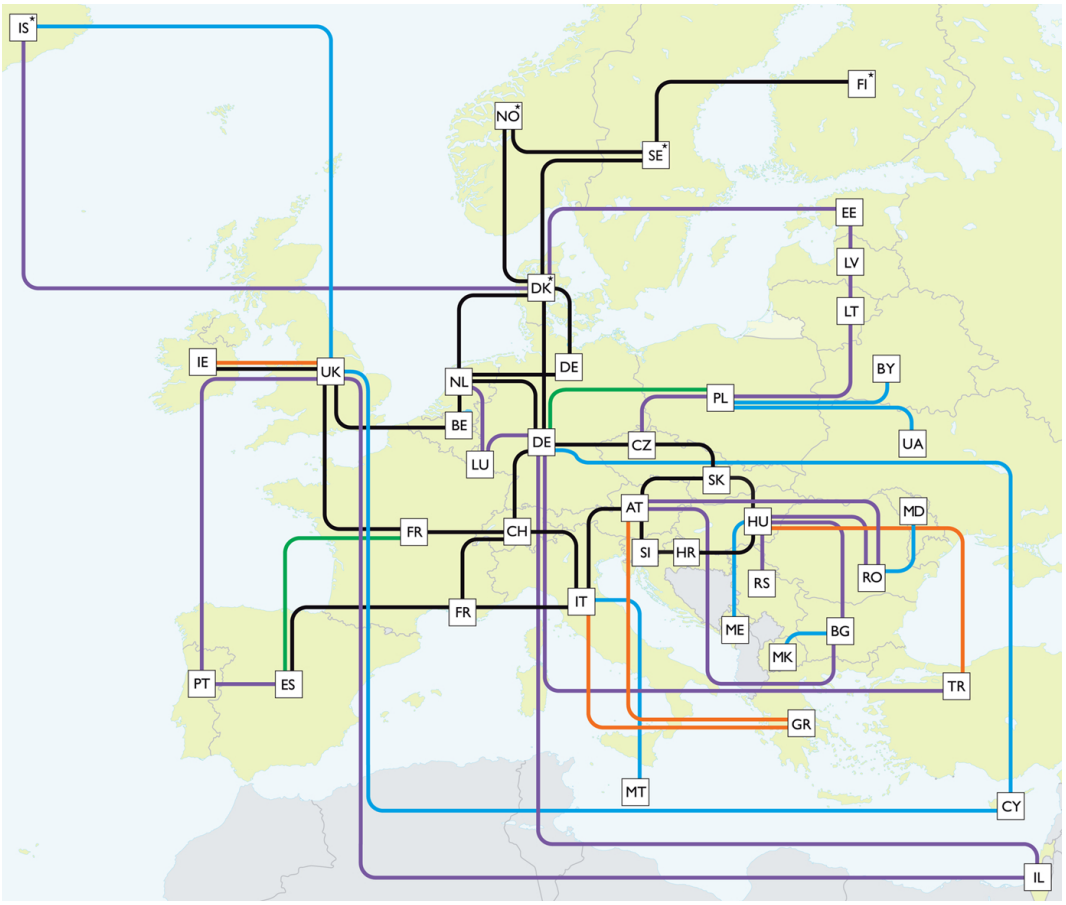


Fig. 3.5 GÉANT pan-European network

is closer to the devices, thus reducing the amount of time required for OpenFlow messages to cross the control channel and *(ii)* the ICONA clusters are smaller, with fewer links and devices, thus decreasing the time used for computation and the overall numbers of flows to be installed and removed from the data plane.

GEANT network

The same metrics have been evaluated on the GÉANT topology (see Figure 3.5). Circuits have various one-way delays (from 10 to 50ms) and throughputs (from 1 to 100Gbps).

Table 3.1 depicts similar results as the previous test. While the GÉANT network is smaller than the grid topology, with 41 switches and 58 bi-directional links, the

Control plane	Avg latency [ms]	Min latency [ms]	Max latency [ms]
ONOS	297	284	308
ICONA2	272	261	296
ICONA4	246	232	257
ICONA8	221	199	243

Table 3.1 GÉANT network: average, maximum and minimum latency to reroute 100 paths in case of link failure for ONOS and ICONA (2, 4 and 8 clusters)

higher delay in the data plane adds an additional time before convergence to a stable state.

3.3.2 Startup Convergence Interval

This second experiment measures the overall amount of time required for both solutions to re-converge after a complete disconnection between the control and data planes. The tests have been performed over the GÉANT topology, and replicated 10^3 times. Table 3.2 shows the average, maximum and minimum values in seconds.

Control plane	Average Time [s]	Minimum Time [s]	Maximum Time [s]
ONOS	6,98	6,95	7,06
ICONA	6,96	6,88	7,02

Table 3.2 Amount of time required to obtain the network convergence after disconnection for ONOS and ICONA

The result shows that ICONA and ONOS require comparable time intervals to return to a stable state, in case of a complete shutdown or a failure of the control plane.

In the next chapter we are going to describe a QoS framework architecture on top of ONOS which exploit the functionalities provided by ICONA for the inter domain control over WAN.

Chapter 4

A Proposal for End-to-End QoS Provisioning in Software-Defined Networks

Part of the work described in this chapter has been published in [38].

4.1 Motivation

Cloud-based, real-time applications are likely to spread over the next years with the advent of IoT and smart mobility systems. Recently, a new plethora of applications requiring a RTT delay of around 1ms have been grouped under the hat of tactile-Internet applications: a tactile sensor reads information and a connected system reacts with actuators seen by a human within 1 ms [39]. Although we are still far from achieving end-to-end RTT of around 1ms with wireless communications, ISPs need to be ready to re-architect their software control-plane in order to fully exploit the enormous potentials offered by their infrastructures.

Current adoption of distributed control algorithms forces the use of the same signaling protocol (e.g. RSVP, BGP-LS) in all the data-path nodes, not taking into account the resistances inevitably present between device vendors and between administrative domains. For this reason the Service-Level-Agreements (SLAs) be-

tween a service provider and its customers or between providers are still mainly static. Moreover, the experience has shown that the scalability issue of the core network in maintaining per-flow state for resource reservation in each node along a path prevented the diffusion of RSVP and integrated services in general. As discussed in [40], per-flow service treatment does not scale in the Internet core; backbone routers must be fast and only an *aggregate behaviour* is feasible. Instead, it is important to enable such treatment at the edge of the network where mass of users enjoying a mixture of heterogeneous applications share indistinctly the same portion of the network as in the case of cellular access networks; performance degradation is likely to happen in the access links where an increasing number of traffic sources and sinks can introduce a significant amount of queueing delay. Given these considerations, we believe that an hybrid combination of flow-based and class-based traffic treatment, respectively at the edge and in the core of the network, could enable guaranteed services for current and future real-time applications. Since these applications could have terminals deployed all around the globe, the end-to-end provisioning will have to span a chain of administrative domains, requiring an east-west communication interface to convey data that vary from classic inter-domain routing information exchanged via BGP. If we make the assumption that QoS requirements requested by the customer applications are satisfied in the core network, at least for what concerns a delay bound, and up to a maximum bandwidth allocation, then we can think to overlay an integrated service scheme on top of the current deployments where class-based treatment is applied, as long as the resource admission control system is able to map the dynamic service requests to the statically allocated resources in the core.

In this Chapter we present the design and a prototype implementation, partially based on ICONA (Chapter 3), of a control-plane network application for provisioning dynamic end-to-end QoS profiles to end-user applications. Our proposal is a signaling scheme for path reservation and configuration whose implementation does not require the involved data-path devices to be bound to a single control protocol. The aim is to solve the interoperability problem in provisioning end-to-end guaranteed services, in a multi-vendor, multi-technology and multi-domain environment by exploiting current software technologies advances; in particular, the decoupling between functional intents and the way they are accomplished is crucial.

The next sections are organised as follows: Section 4.2 gives a high level description of the complete system workflow, the first part of Section 4.3 is dedicated to a general introduction to the QoS and to the Internet technologies adopted to achieve it. Here is where the most of the related works are considered. Then in Section 4.3.2 the critical issues of RSVP are presented and in Section 4.3.3 the end-to-end behaviour of the QoS system is taken into account. In Section 4.4 we discuss the communication interface between clusters or domains of networks required to achieve the end-to-end provisioning, while in Section 4.5 a brief contextualisation of our work into a policy management system is presented. Section 4.6 contains the details of our prototype implementation while Section 4.7 presents some benchmark results on the service request computation time.

4.2 System Workflow

The overall process is activated upon the occurrence of an event triggering the dispatch of a request sent by an Over-The-Top (OTT) application. The request contains a user authentication token, an application identifier, information about an endpoint to contact together with additional flow specifications, and a QoS profile containing delay and bandwidth requirements, plus an amount of time (or an estimate of it) for which the profile is required. A previous agreement between the user and the network operator is made in order to convey to a traffic plan based on its dynamism, amount of data, QoS parameters, number of requests and possibly other parameters. The request is sent to a manager application running on top of the local domain controller that is listening for incoming connections from registered users. The authentication token, previously generated in a hand-shake phase is verified and the content of the payload is parsed and elaborated as follows.

The endpoint information, either a destination address (L2 or L3, depending on the use-case) or the hostname of the machine to be contacted, is used as look-up key to get the collection of candidate paths existing between the end terminals of the user application. Then an *admission control* routine runs to check the availability of a suitable path where resources are to be reserved for the subject QoS profile. Once a path is selected, the network application has to instruct the core controller to setup a priority flow between the endpoints of the user application; for the sake of simplicity,

we will refer to point-to-point paths, although the solution is equally applicable to paths with multiple destinations (> 2).

It may be necessary to establish connectivity between the endpoints, other than traffic control's rules. For example, in a pure Openflow network where none of the routing protocol suite runs in the data-path devices, the controller would prescribe a set of flow rules containing forwarding instructions, together with QoS constraints. If forwarding rules have been previously installed on the data-path devices, then only traffic classification and shaping is to be done through device-specific configurations.

To setup a priority flow, the manager application issues the setup of custom queues in the devices along the selected path and sends an intent request to the controller's core with the specification of the target flow. An intent is an abstraction used by the applications to specify their high-level desires in form of policies. The ONOS network controller (Chapter 2), used in our prototype implementation, is the first open-source controller that provide such feature to its applications. The intent is then split by the core into device-specific flow rule requests dispatched to the proper software drivers of the underlying devices. Queues are also configured by means of device-specific drivers. Unfortunately the ONOS *Intent subsystem* does not support the configuration of queues and its southbound level yet, and so we had to implement the configuration in our application.

As mentioned before, the endpoints can normally span multiple domains and, clearly, each domain has to take care of its portion of the network. A key point of the system is the topology abstraction (Figure 4.2): all forwarding devices of the local domain are exposed to the controller with the same abstraction model, as is an entire remote domain topology, viewed as a single device (a big switch) whose ports are connected to terminal endpoints or to other domain topologies. When the endpoint of the application requesting a priority path resides on a different domain, the admission control routine recognises the presence of a virtual device associated to the remote domain and as a consequence queries the network manager application of the remote domain in order to establish an end-to-end resource reservation along the path between the endpoints (Figure 4.1). Once the process converges, the domain controllers can simultaneously setup the path in their own portion of the network,

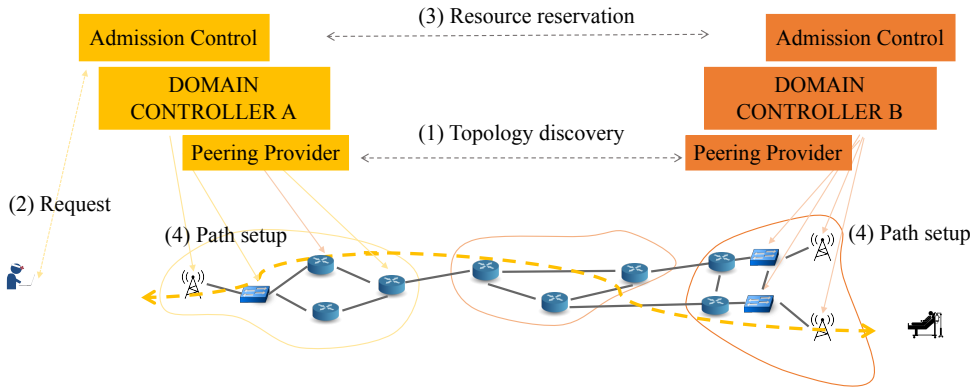


Fig. 4.1 Reference scenario

the manager application that received the original request sends a response back to the user application that can start sending priority data over the network.

4.3 QoS Provisioning

4.3.1 General Discussion

QoS is introduced in packet-switched networks in order to apply a special treatment to specific data packet flows. At the device level, QoS is achieved through traffic classification, shaping, and scheduling at the egress ports; at the ingress ports packets are filtered for policing. Classification determines which treatment each data packet has to undergo, shaping is used to control customer input data rate to conform to the SLAs, while scheduling affects delay and throughput of the data packet flows. Given conformance to a SLA, scheduling at the network interfaces is the lowest level operation gearing QoS provisioning, also known as *service discipline*.

At the path level, QoS is achieved through resource reservation over the whole set of nodes along the path. The reservation must be secured end-to-end and the resource allocation in one node must be consistent with the others along the path. A path selection with the end-to-end delay constraint is subject to inaccurate information

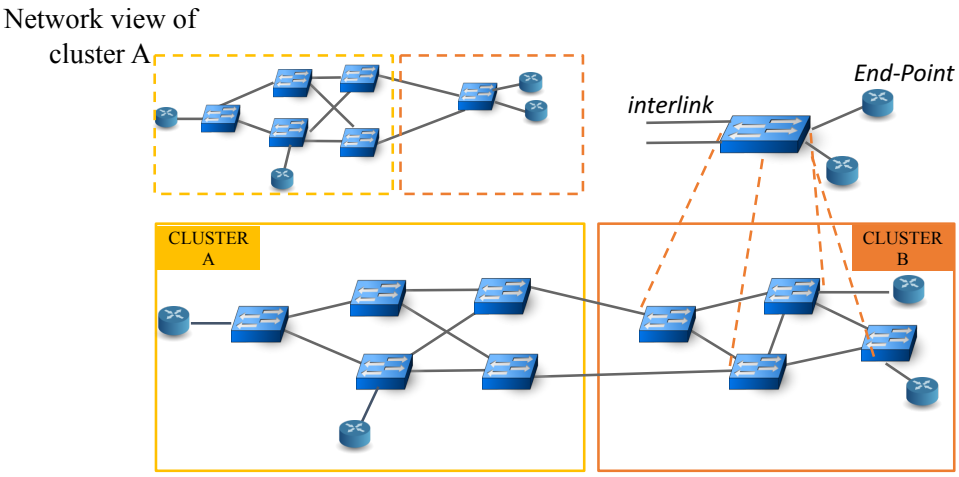


Fig. 4.2 Domain topology abstraction

due to the dynamic nature of the delay and hence subject to theoretical intractability ([41] [42]). However, end-to-end delay and delay jitter bounds have been computed by means of network calculus applied to queueing systems modelling the networks ([43–45]). Academic and industrial communities have been very active in the last decades in investigating network models and algorithms to solve the *QoS routing* problem, also known as the *multi-constrained path computation* problem ([46–52]). In [40] they cover all the important components of the QoS provisioning in Internet as it has been conceived in the last decades: integrated services, RSVP ([19, 53]), Diff-Serv [54], Multi Protocol Label Switching (MPLS [55]) and constraint-based routing.

Today network operators employ MPLS mainly for layer 2 and Layer 3 Virtual-Private-Network (VPN) services ([56]), while constrain-based routing for Traffic Engineering (TE) operations is complex to achieve in a complete distributed control-plane. Moreover, TE is not useful in presence of congestion. This happens at the bottleneck links that typically reside in the last mile towards the customers. DiffServ model within a single AS is employed by ISPs for class-based treatment of the data packets; but the validity of the Type of Service (ToS) field in the packet IP header may lose completely meaning when traversing multiple administrative domains. In other words, within a single administrative domain the class-based QoS provisioning is theoretically easy to achieve and technology is not the hurdle, while policy and economic factors have the major impact on the fate of the QoS provisioning in the

multi-domain scenario. IntService and the RSVP signaling protocol instead did not take off even within the single administrative domain. RSVP is used for labels distribution in G/MPLS but it failed in its primordial intent. In order to accomplish the process described in Section 4.2 an end-to-end guaranteed service must be provided. In the next section we discuss the critical issues of RSVP and in what our proposal differs from it.

4.3.2 Comparison with RSVP

Path Computation and Routing. RSVP uses a combination of Constrained Shrotest Path First (CSPF) algorithm and Explicit Route Objects (EROs) to determine how reserved traffic and signaling messages are routed over the network; RSVP is a distributed signaling protocol and it needs routing to work. EROs are a mean to explicit indicate some nodes that must belong to the reserved path; in order to force a specific path through a set of nodes you should enter and configure each node with specific EROs instructions. If the total bandwidth reservation exceeds the available bandwidth specified across the link for a particular path segment, the path must be recomputed through another route. If no segments can support the bandwidth reservation, path setup fails and the RSVP session is not established.

In our solution the path computation is independent from the routing protocol. Different routing and forwarding scheme can be used to build the path in the underlying network: flow-rules, labels, tunnels. But no routing of the signaling scheme is needed; centralised path computation is clearly much faster and protocol-independent. The controller only needs the view of the topology as a connected graph. To force the reservation in a specific path, you can directly reserve resources and install forwarding rules into the proper devices at once. The candidate paths are collected from the store and a suitable one is found before injecting resource reservation rules into the network. This occurs in the centralised controller within a single software process.

RSVP is simplex. In RSVP the reservation process is applied in a single direction of the path. To have full duplex reservation, the number of operations and the

messages exchanged are doubled.

In a centralised network controller, you can equally provide simplex or duplex reservation via a single software entity.

Admission and Policy Control. RSVP Admission and policy control is applied to each node. So RSVP implementation must be integrated with each node's traffic and policy control module, thus increasing the chance of interoperability. RSVP must provide QoS service characterisation within opaque objects parsed by each network node.

In our proposal the QoS service characterisation is completely decoupled by the signaling protocol/scheme and must be embedded only into the front-end APIs consumed by the customer applications. The admission and policy control is applied only once, in each domain, in the central controller. An RPC-based mechanism is used for configuring the traffic control module given the possibility to fulfil the request. Our prototype relies on ONOS. ONOS provides common abstracted behaviours for traffic selection and treatments that are translated into device-specific rules.

First speaker. RSVP session initiator is the inbound router running RSVP in conjuction with other protocols (e.g. MPLS or GMPLS in case of label distribution). If RSVP is embedded in a host application, then the first network node should speak RSVP, otherwise a tunnel between the application and the first RSVP-capable device shall be created thus increasing the number of operation required for RSVP signaling to work.

In the presented solution the session initiator is a user application featured with proper APIs for contacting the controller. The idea is to keep the API consumer implementation as simple as a REST client that has the capability to create, remove, update and delete a priority path. Such client would be provided for different software environments.

Scalability. As per [57], the scaling problems of RSVP are linked to the resource requirements (in terms of processing and memory) of running RSVP. The resource requirements increase proportionally with the number of sessions. Each session requires the generation, transmission, reception and processing of RSVP Path and Resv messages per refresh period. Supporting a large number of sessions, and the corresponding volume of refresh messages, presents a scaling problem.

A centralised control plane presents the same scalability issues concerning the state maintenance of an increasing number of sessions in the data-path nodes. But it only matters the traffic control and flow rules, while processing and singnaling overhead are significantly lowered.

Complexity. Finally, and maybe the most relevant obstacle to success, RSVP is complex because it was designed with IP multicast in mind, intermediate nodes have to merge resource reservation requests coming from the receiver nodes. Moreover, the basic RSVP reservation model is "one pass": a receiver sends a reservation request upstream, and each node in the path either accepts or rejects the request. This scheme provides no easy way for a receiver to find out the resulting end-to-end service. To solve this issue an enhancement was proposed [58], introducing further complexity in the concretization of RSVP and the integrated services in general.

Orchestrating the data-path nodes from within a central controller avoids the issues related to the exchange of asynchronous signaling messages. The decision process not being distributed decreases the complexity in maintaining a single state of the system.

4.3.3 End-to-End Behaviour

The end-to-end QoS profile model shall follow the one described in the Specification of Guaranteed Quality of Service [59]. As per [59], "*the end-to-end behaviour provided by a series of network elements is an assured level of bandwidth that, when used by a policed flow, produces a delay-bounded service with no queueing loss for all conforming datagrams*". We invite to refer to the specifications for further clarification about the QoS model taken into account. Each network node must

provide a service that matches, with some error bounds, the fluid model ([60, 61]) through the token bucket scheme with paramters (b, r, p), respectively the bucket size, the token rate and the peak rate. The QoS request includes a maximum end-to-end delay bound, d_{req}, that shall be guaranteed between the application terminals.

In the centralised controller, the link providers are responsible for notifying the presence of links they are provider for and their characteristics (propagation delay, transmission capacity and the maximum transmission unit); these information are stored in the controller database upon discovery of the link itself. Likewise, the device providers in the southbound must export other relevant information, such as the delay error terms representing how the device's implementation of the guaranteed service deviates from the fluid model in each network interface (the C_{tot} and D_{tot} in the formula 4.2 below).

On a link l with capacity c_l, we define a minimum bandwidth reserved to the best effort traffic, R_{be_l}. Let R_i be the allocated bandwidth for a flow i. On each link, the total number of accepted profiles N is subject to:

$$N : \; c_l \geq R_{be_l} + \sum_{i}^{N} R_i \tag{4.1}$$

The end-to-end delay bound as defined in [59] is:

$$[(b-M)/R * (p-R)/(p-r)] + (M + C_{tot})/R + D_{tot} + \sum_{l} d_{p_l} \tag{4.2}$$

With $r <= p <= R$, M being the path Maximum-Transmission-Unit and $\sum_l d_{p_l}$ the propagation delay sum.

Statement 4.1 imposes that the sum of the allocated bandwidths for N flows must not exceed the capacity of the link. Flows requesting a maximum delay bound are assigned to higher priority queues w.r.t. to the best effort traffic. It is possible to assign the same high priority queue to more distinct flows, as long as statement 4.1 holds. R shall be chosen such that d_{req} is greater or equal to the value computed in equation 4.2, provided that d_{req} is greater than the fixed delay terms D_{tot} and $\sum_l d_{p_l}$. These constraints must apply on all links of a candidate path between the end

terminals of the customer application; a new queue is created for a new flow if they are satisfied. As mentioned in the previous section, the *admission control* routine occurs only once per domain (more details in Sec. 4.4), in the central controller. Network elements must export the proper information, while the drivers have to translate a service request into device-specific traffic control rules.

The provisioning of a guaranteed service along a path of several devices and links is possible only through a cross-vendor and cross-technology solution. This leads to the adoption of software driver modules installed into the centralised controller. These drivers converts the protocol-agnostic rules into device-specific instructions and are essential to solve the interoperability problem derived by the presence of devices from multiple vendors and technologies. For example, in a LTE cellular network, the high-level profile is mapped to a standardised QoS Class Identifier (QCI) by the proper software driver; the mapping would be followed by the setup of the packet data network gateway and the mobile station with some scheduling rules applied to the target data flow [62]. The same high-level profile has to be translated into a specific setup on the backhaul that provides the connectivity towards the core network consisting of all the required switches to aggregate the traffic from the access cellular network [63]. These switches could be, for instance, pure Linux devices in which case the driver would execute a remote procedure call configuring the involved interfaces with the well-known commands suite *tc qdisc*, *tc filter* and *tc class* for setting up the queues. If instead a switch is an Openflow-enabled device, *classification* and *priority* come within the forwarding rules, while the queue configuration for the *service discipline* must be supplied on a separate communication channel, for example, through OVSDB protocol in Open-vSwitch [31] (Fig 4.3).
Together with the local domain (or local cluster in the single administrative domain) our framework adds the reflection of such operations into the remote domain (cluster) where the endpoint of the customer application requesting the service resides.

Note that we control the edge devices on each side of the communication while leaving aside the backbone routers where per-flow service treatment does not scale. While rfc-2212 states that all the nodes of a path should take part of the resource reservation process for equation 4.2 to hold, we argue that the dynamic resource reservation at the edge of the network can occur transparently w.r.t. the statically allocated resources of the core where the QoS exists only for classes of traffic,

rather than flows, in form of virtual circuits created with protocols such as MPLS or GMPLS (Fig. 4.4). If the backbone is treated as a composition of these circuits rather than a composition of nodes and links, then a resource mapping between the dynamic and the static portions of the network resolves in representing these circuits as aggregate elements into the topology view of the controller. Path-Computation-Element (PCE) describes a model to address the problem of constrain-based path computation in conjunction with a label switched protocol ([64, 65]). An all in one orchestration framework for the complete set of the network elements is presented in Chapter 5.

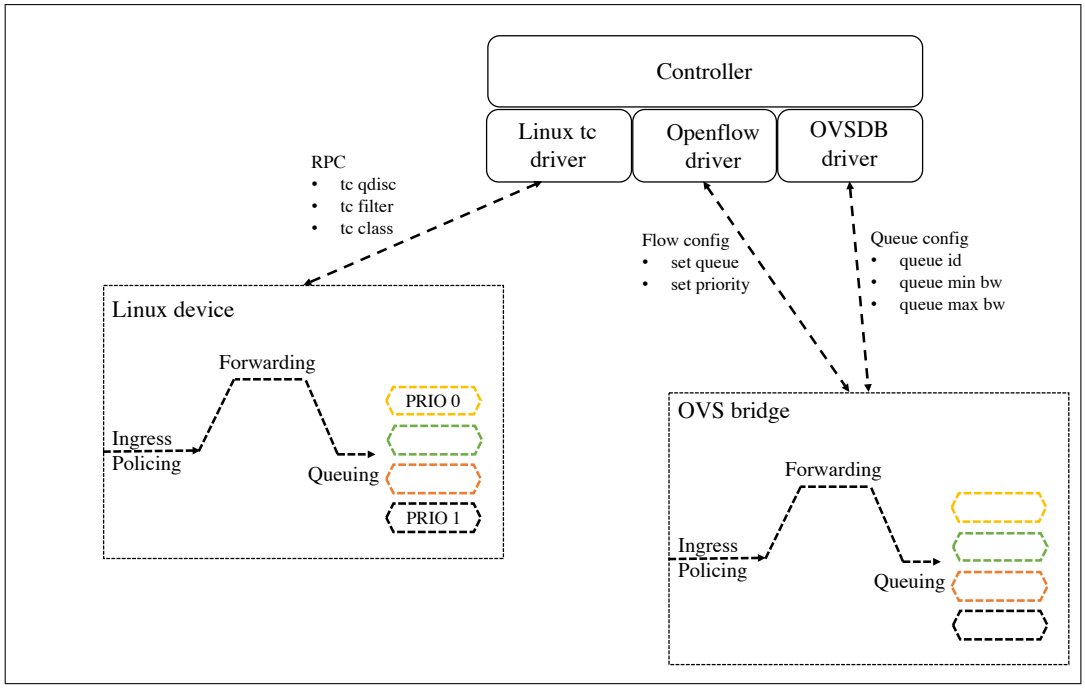


Fig. 4.3 Driver modules in the controller are the means for device-level configuration of the queues

4.4 East-West Resource Exchange

When the terminals of the application requiring a priority path are located in two portion of the network piloted by distinct controllers, a communication mechanism between these controllers is necessary in order to exchange the proper information during the reservation process. From hereafter, we will use the term domain and cluster interchangeably to indicate distinct portions of the network.

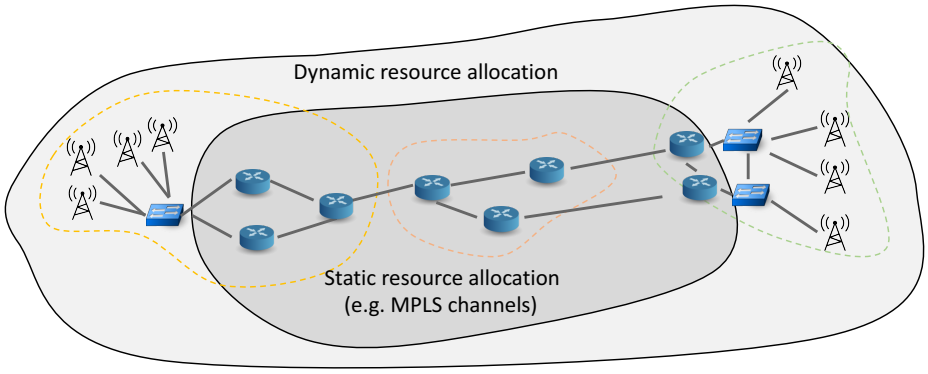


Fig. 4.4 Two allocation schemes for two layers of the network

At the origin of the communication, there is the route discovery phase to share the endpoints of each domain. A design choice is to be made on how and when to expose the network element parameters related to the topology itself. There are two options: **i)** sharing these information during the route discovery phase, or **ii)** avoid to pre-share the parameters and collect them during the resource reservation process on-demand, that is, every time a new request arrives.
Suppose we have a portion of the network under domain A, and another portion under domain B, then suppose that at some point in time an application connected to A asks for a priority path that includes an endpoint in domain B. The two different approaches are described in the following sections.

4.4.1 Pre-Shared Network Parameters and Bandwidth Resource

With option **(i)**, the controller A has already all the information required for the end-to-end admission control routine when the request arrives. This means that the topology exposed by B to A shall include all the necessary network infrastructure parameters, $(c_l, MTU, C_{tot}, D_{tot}, d_{tot}^{prop})_p$ for all p in the set of paths that B is willing to expose to A, which in turns implies that the exposed topology shall be detailed enough for A to select a suitable path. This requires a more complex topology abstraction than the single big switch depicted in Fig. 4.2, because clearly with the single node abstraction you cannot have as many path properties as you would have with a network of nodes.

You could achieve a certain level of aggregation by hiding elements of the B topology, by computing the aggregate parameters for some paths towards the destinations and then exposing a virtual topology composed by only these paths to A. In this case, each domain controller should maintain a mapping between the local physical devices and links and the virtual ones exposed to the remote domains which adds further complexity.

Other than the topology abstraction, there is one major issue with this approach: the computation of the aggregate, rate-related delay error term, C_{tot}, which is theoretically not feasible when the computation occurs, for instance, in the domain controller A for some devices of domain B, because C depends on the parameter r, the rate requested by a user application. So either C is expressed as a function of r for each device and shared during the topology discovery phase, which means exposing the entire topology, or it must be computed on-demand, by the domain controller B as described in the next section.

4.4.2 On-demand Network Parameters and Bandwidth Resource

In this case, the path parameters are collected and exchanged during the admission control routine. The domain B is requested to run the resource reservation process in its own domain. Controller A forwards the application request parameters (i.e. d_{req}, the tuple (b, r, p), the domain ingress point and the target destination) to B, which replies with a tuple $(M, C_{tot}, D_{tot}, d_p)^B$ and an upper bound on R, chosen based on a proper selected path, if available, so that A can compute the end-to-end delay bound before proceeding with the actual reservation and path setup. This way the topology abstraction can be kept as simple as a single big switch whose function is to merely offer a point of connection to the remote destinations to any domain controllers with which there is a peering. The network parameters and the resource selection comes directly from an up-to-date decision process made within the concerned domain. The computation of C_{tot} can be actually computed while masking the details of the local topology. This approach is the choice of our implementation prototype described in the rest of the article.

4.4.3 Inter-Domain Resource Scope

The network parameters used for the delay bound computation in equation 4.2 are mainly static, except for the bandwidth R, the dynamic network resource under consideration. We assume unlimited buffer space for the queues, or at least enough to support the traffic bursts in any link of the network. Within each domain a resource management system should track the allocated and the available bandwidth in each link of the underlying network. From the inter-domain communication perspective, the bandwidth resource management unfolds three cases depending on the use-case scenario:

- **Full share**. The bandwidth is completely shared between applications, regardless of the domain they reside. This can be the case where the control plane of a single administrative domain is split into multiple regions for performance reason (see ICONA, Chapter 3) or because of different underlying physical networks. In this scenario, it is necessary to update the exposed resource each time an application obtains or releases a portion of the bandwidth. Upon a new allocation or release in one domain, a message is sent in broadcast to all the other domains with which there is a peering. The message is read by the remote controllers and their local resource stores are updated accordingly. In the on-demand mode of communication, Section 4.4.2, each domain manages its bandwidth resource independently and only during a reservation process the resource availability in a remote domain is determined.

- **Partial share**. This case is equal to the previous one but only a portion of the bandwidth is shared with the remote domains.

- **Partial static share**. A domain controller advertises to each remote controllers a virtual value of the bandwidth resource during the route discovery phase and no further update messages are exchanged between controllers. This value is the static portion of the bandwidth allocated to each remote domain. In this case, also with the on-demand mode of communication a controller can determine the availability of bandwidth even before contacting a remote domain. This is the case of the multi administrative domains scenario.

4.5 Policy Enforcement

Policy-based QoS management is of primary importance for network operators. If the *service discipline* at the network interface is the lowest level operation gearing QoS, at the top level we have the SLAs expressed in terms of policies. The SLAs consist of a set of specifications that are translated by the network manager into device level primitives (e.g., forwarding rules, queue configurations, traffic shaping policies, etc.). In [66] [67] the authors propose automatic policy based management system in the Internet DiffServ architectures. They present a framework for policy management that reacts to network state changes or customer users requests to dynamically re-adapt the policy enforcement. In [68] a management framework for automatic policy enforcement is introduced in a network controller based on Openflow; they describe all the necessary functional components of the system without entering in the implementation details of any of them thus avoiding to discuss how do they actually interact between each other.

The focus of the present article is on the QoS provisioning in the economic context of dynamic SLAs; this framework foresees the possibility to be integrated with an existing policy-based management system. The concerned SLAs are between a service provider and its customers and between service providers who cooperate to provide an overall service that can span multiple administrative domains. Between the customer and the provider, a set of APIs can be embedded directly into the customer applications and layered on top of an existing policy management system. The network operator could also provide ready-to-use applications for specific services (e.g. a remote health control system). Here we limit the discussion by listing the additional information needed by the policy manager in order to conform the ingress traffic to the dynamic SLAs.

Policy to regulate the interaction with customer applications:

- List of user and application IDs that are allowed to request a service.
- Upper bound on the amount of bandwidth each user could request.
- Maximum amount of time each user is allowed to retain a priority path.
- Amount of bandwidth reserved to the best effort traffic.

- Set of destinations for which a user could request a priority path.
- Set of network elements that cannot be part of the reservation process.
- Pre-configured queues for selected customers or applications.

Policy to regulate the interaction between providers:

- List of peer domains that are allowed to interact with the local domain.
- List of destinations to expose to the remote domains.
- Topology abstraction to expose to the remote domains.
- Virtual bandwidth resource associated to the exposed destinations.
- Aggregate parameters selection, $MTU, C_{tot}, D_{tot}, d_p$.
- Number of total service requests that a remote cluster is able to perform.
- Pre-configured queues for selected peers.

4.6 Architecture

4.6.1 High-level System Components

The main functional modules in the control plane are *protocol agnostic* thanks to the separation of concerns given by the network controller architecture of ONOS that is partitioned into:

- Protocol-aware network-facing modules that interact with the network.
- Protocol-agnostic system core that tracks and serves information about network state.
- Applications that consume and act upon the information provided by the core.

At the application layer resides the *admission control* and *resource allocation* routine. It guarantees the correctness of the QoS provisioning to the end-user applications; it

has to dynamically setup and teardown multiple and concurrent QoS profile sessions and verify that everything in the underlying network is up-to-date and in a consistent state. In the core controller there are several components required to accomplish the complete reservation process, see Figure 4.5, while in the network-facing layer we have as many drivers as the number of different devices in the underlying network and a communication interface used to exchange data between the domain network controllers. Such interface has to take into account several aspects of the system: routes to destinations discovery, network topology elements exposition, resource reservation parameters and a policy-driven mechanism to abide to the SLA made between the involved administrative domains. We leverage on ICONA to fulfil the remote topology and destinations discovery function. The resource reservation parameters are currently exchanged during the *admission control* routine at the application layer, using a prototype REST channel interface. The integration with a policy manager is left as a future work.

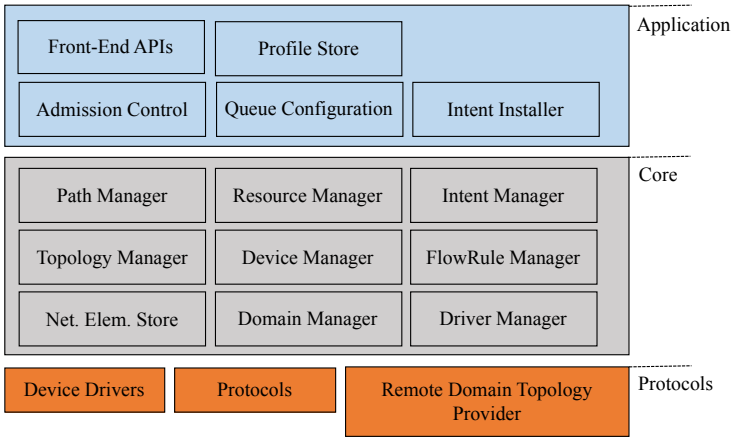


Fig. 4.5 Controller's Main Components

4.6.2 The Manager Application

The manager application is a standard, on-platform ONOS application. Its main function is the *admission control* routine. We separate the routing and resource assignment process into two steps: the collection of candidate paths between the application terminals through the ONOS *PathManager* and the selection of the one that

satisfies the constraints imposed by the basic *admission control* scheme described in Section 4.3.3. The state of the underlying network resources is tracked by the *ResourceManager*, backed by a distributed store, which is queried during this process to reserve the bandwidth resource. The local path parameters are collected by the internal store populated with the information of the underlying network elements, while in presence of virtual domain devices, the parameters are queried to the remote controllers and then merged with the local ones. If a suitable path is found using Algorithm 1, the bandwidth in each link is temporary reserved (locked), priorities queues are setup in each device through the *QueueManager* module and an intent is submitted by the *IntentInstaller* (Fig. 4.6). If the installation is successful, the *ResourceManager* is requested to allocate the previously locked resource, otherwise a rollback is performed on all the previous actions, bandwidth reservation and queues configuration. The collection of the candidate paths is subject to the constraint imposed by the eq. 4.1; other criterions may be applied, taken, for instance, by a configured policy. A REST applet that implements the APIs consumed by the end-user applications and by the remote domain controllers to request and terminate the setup of priority paths is also part of the manager bundle. All these components exploit the service-based OSGi model to communicate each other within the platform.

Algorithm 1 Pseudo-code of the admission control routine. *Assuming $p >> r, R$

1: (b, r, p, mtu, d_{req}): `user app request`

2: $R \leftarrow r$
3: `collect all paths whose spare capacity on each link is` $\geq R$
4: **for all** `p in paths` **do**
5: **if** `p contains a domain device` **then**
6: `collect` $(R,MTU,C,D,d_prop)_p$ `from the remote domain`
7: `merge local and remote path parameters`
8: **end if**
9: $R \leftarrow \max\{r,\ (M + C_{tot})/(d_{req} - D_{tot} - d_{tot}^{prop})\}$*
10: **if** `R can be allocated along the path` **then**
11: `allocate R and setup queues`
12: **else**
13: `rollback and try next path`
14: **end if**
15: **end for**

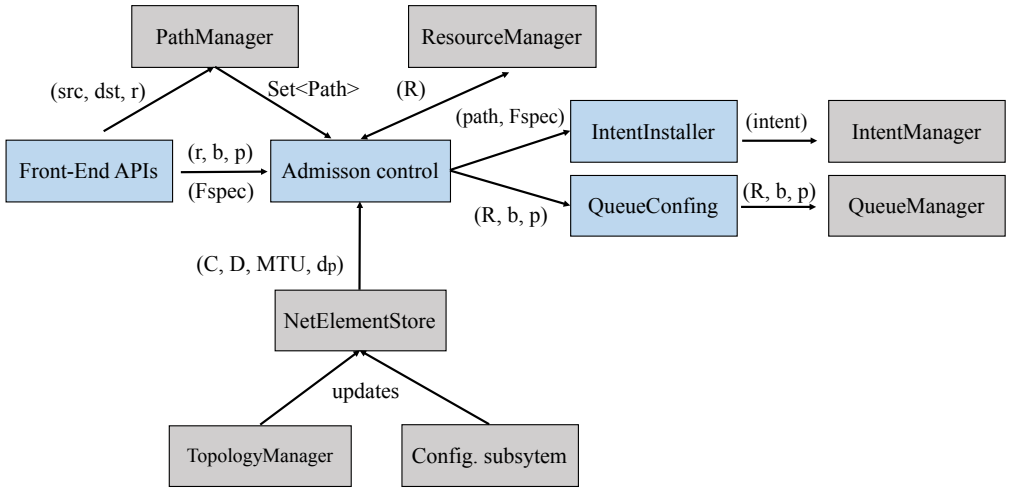


Fig. 4.6 Admission control workflow.

4.6.3 Routing and Scalability

With a complete view of the network topology, routing becomes a problem of graph searching; the ONOS *PathManager* exports the proper APIs to the northbound applications. Destinations addressing is a matter of use-case scenario; in our prototype implementation the endpoints are identified by L2 addresses and the infrastructure devices are Openflow devices, but this does not affect the generality of the system because, as mentioned in 4.6.1, the main functional modules like the admission control routine are protocol-agnostic. Currently the topology discovery is implemented by a full-mesh communication between clusters of ONOS and each cluster only advertises the destinations that are directly connected to the local devices, thus avoiding to implement a distance vector or link state routing protocol. Every instance of a cluster handles the peering with a subset of all the other clusters through a leader election process implemented in ONOS in order to load-balance the number of peering connections among the ONOS instances.

4.7 Algorithm Computation Time Evaluation

The purpose of this section is to report some benchmark results on the scalability performance of the control plane of our prototype implementation.

Table 4.1 Single-request computation time

	$N = 50, p_l = 0.75$	$N = 100, p_l = 0.75$	$N = 450, p_l = 0.05$	$N = 500, p_l = 0.05$
Response time (ms)	6.392	9.109	201.842	331.055

The resource reservation process overhead within a single domain is proportional to:

$$D_{app,ctl} + T_{algo} + \max_{n \in N} \{D_{ctl,n}\}$$

where:

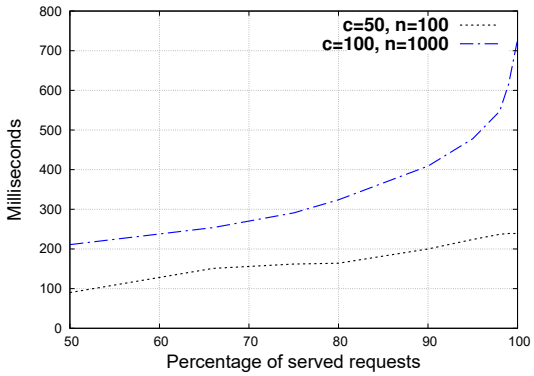
- $D_{app,ctl}$: latency between the application and the controller.
- T_{algo}: admission control computation time.
- $D_{ctl,n}$: latency between the controller and the n_{th} device.

The overhead when considering the endpoints placed in M domains is proportional to:

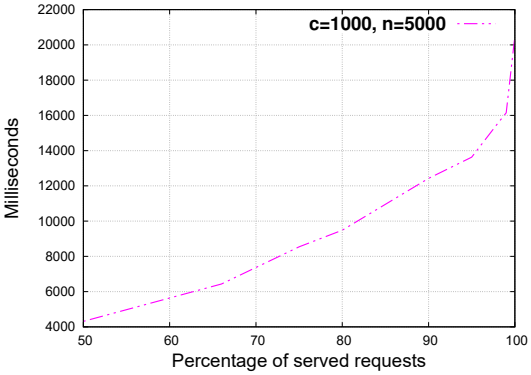
$$D^i_{app,ctl} + T^i_{algo} + \max_{j \in M \setminus i} \{D_{ij} + T^j_{algo}\} + \max_{n \in N, j \in M} \{D^j_{ctl,n}\}$$

where D_{ij} is the latency between domain controller i and j. In [32] we show that splitting the control plane into multiple clusters improves the event-to-response reactivity by decreasing the $D_{ctl,n}$ term at the expense of adding communication overhead between clusters. Splitting the control-plane into multiple regions also decrease the algorithm computation time that here we benchmark within a single region.

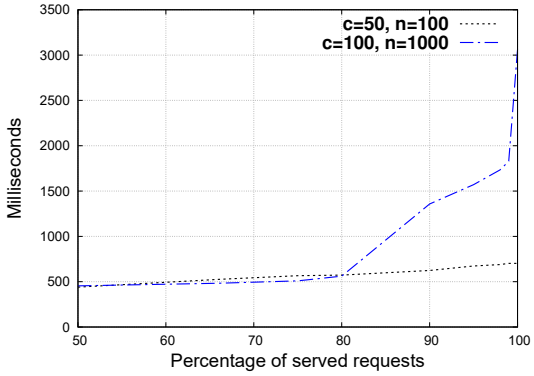
The overhead is reported against increasing number of concurrent HTTP requests and increasing topology dimension for a single instance of an ONOS cluster running on a bare metal HP EliteDesk 800 G1 SFF with Intel Core i7-4770 CPU @ 3.40GHz, 16 GB RAM. Random topologies are generated by assigning a probability p_l of link existence between any two nodes (Bernoulli model) and injected into the ONOS core database. N is the number of infrastructure devices, c the concurrency level and n the total number of requests per experiment. For $N = 450$ and $N = 500$ we exploit the limit theorem according to which the probability that a Bernoulli random graph is fully connected is distributed as $1 - N(1 - p_l)^{(N-1)}$. The thread pool is configured to use up to a maximum of 300 threads. The computation time T_{algo} is assumed to



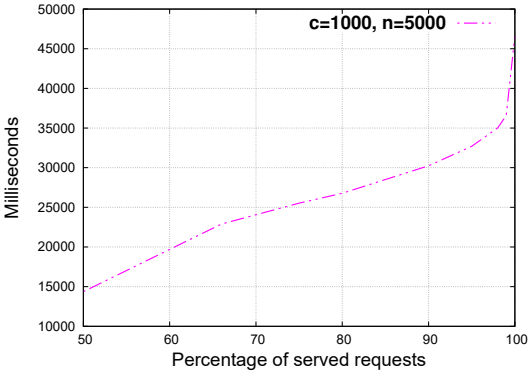
(a) $N = 50, p_l = 0.75$



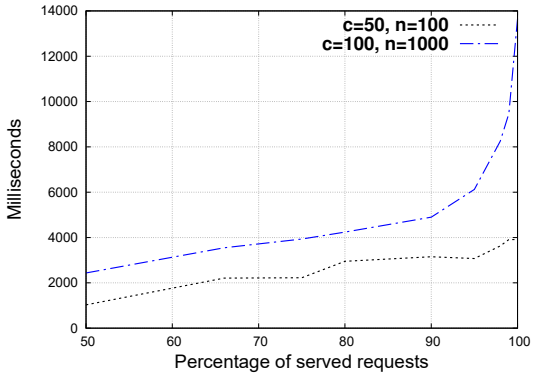
(b) $N = 50, p_l = 0.75$



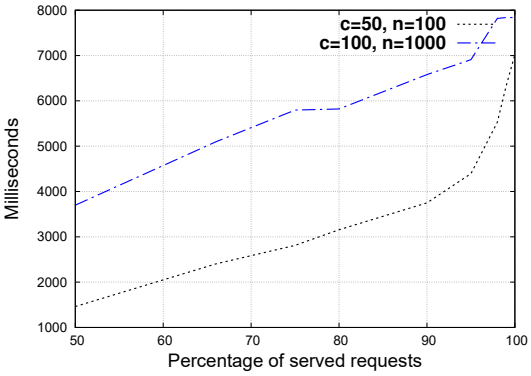
(c) $N = 100, p_l = 0.75$



(d) $N = 100, p_l = 0.75$



(e) $N = 450, p_l = 0.05$



(f) $N = 500, p_l = 0.05$

Fig. 4.7 Percentage of requests served within a certain amount of time expressed in milliseconds. With a concurrency level c=100 (Fig. *a, c*) 1000 thousands requests are processed in less than 1sec. (Fig. *a*) and in about 3 sec. (Fig. *c*). With a c=1000, the processing time increases exponentially w.r.t. the total number of served requests n (Fig. *b, f*). With hundreds of nodes (Fig. *e, f*), even with c=1000 we are in the order of tens of seconds for 1000 requests. The goodness of these results is relative to the type of service and network involved. For example, does a HQ streaming video on-demand for premium customers expect a request rate of thousands of requests per second? Is the QoS applied to all the network nodes along the path or on a small subset of them?

start as soon as the request arrives to the REST applet of the manager application until the drivers for setting up the queues are called. We use a modified version of algorithm 1 in which the set of candidate paths is chosen with the Dijkstra algorithm for a maximum of five shortest paths, using a link cost function that forces to infinite the cost of the direct link between two endpoints, if present, while the cost of all the other links is uniformly distributed between zero and one. The endpoints for each single request are also randomly generated so are the network parameters for the admission control formula. The T_{algo} overhead even for networks with hundreds of nodes (and links and hosts) is of the order of milliseconds (Table 4.1); in Figure 4.7e 50% of the requests, at the origin point, are served within one second. However, we encountered serious problems in processing requests with a concurrency level of 1000 connections with N equal to 450 and 500 so much so that we decided to not report the numbers. This inefficiency is intrinsic to the ONOS controller that showed a greedy cpu usage of some hundreds percentage during the tests, due to possibly unnecessary operations on the simulated network elements. Nevertheless, note that the overhead T_{algo} includes the read time to get the collection of candidate paths, the transactional allocation process on the bandwidth resource on each link of any scanned path, the collection and the access to the device drivers for setting-up the queues. The high values reported in 4.7e 4.7f with a concurrency level of one hundred are given by the failure in the transactional operation for bandwidth allocation due to possible collisions among the requests. This is certainly a point of investigation and the prototype application and control framework would need an engineering effort in terms of scalability and performance for a production-ready application.

Chapter 5

Hierarchical End-to-End Network Control with ONOS

The work described in this chapter has not been published, neither it is under submission to a conference or a journal. This work describes part of the open-source platform maintained by the ONF [69, 21]

5.1 Overview

In Chapters 3 and 4 we presented a peer-to-peer architecture for SDN controllers. Here we discuss a hierarchical platform of ONOS controllers which is partially based on the codebase developed for the peer-to-peer model. This platform is currently one of the building block of the ECORD project (Section 5.3.1 and 5.3) the code is open-source [20]. One key motivation behind this work resides in the lack of an open-source reference platform for unified network resource orchestration from a centralised vantage point and fine-grained bandwidth and connectivity service on-demand. End-to-end connectivity involves the control of multiple heterogeneous underlying networks. ONOS already provides separation of concerns by separating northbound and southbound APIs, but it falls short when heterogeneity of the physical layer becomes dense; in fact the northbound APIs of ONOS do not fully express the capabilities that many device drivers support, nor it would make sense to run all network applications on a single controller because virtual network isolation and resource multiplexing is hard to achieve within a single process platform, in the

machine hosting the JVM. Moreover, it is much more likely that different portion of the networks are driven by different controllers to fully exploit the potential of the infrastructure. For example, in data-center networks, a pure Openflow controller is the best candidate while in the transport network it is likely that you shall interface with proprietary Network Management System (NMS) platforms. We want to separate the controllers into a domain-agnostic one, the global ONOS orchestrator, and multiple domain-specific controllers. This way it is easier to maintain the platform up and running during temporary down times due to failures or software releases. By using ONOS as global orchestrator for network services, we are presented with a platform that has well-defined structures for topology and network elements, stable primitives for HA and network applications. We also address scalability for WAN networks by delegating full routing and forwarding control to the leafs controllers. At the time of writing, the state of the art of hierarchical SDN controllers mentioned in Chapter. 6 were confined within the academia and they did not have any successful feedback from the industry. ECORD is under field trial of an handful of service providers such as China Mobile and TIM and undergoing continuous enhancements by the ONF community.

5.2 Architecture

The platform is composed by a two layer hierarchy of controllers, the root global controller and the leafs local controllers. They all leverage on the ONOS core to store and distribute the state among the instances of a cluster. For the rest of the chapter we will use the term domain and network interchangeably since it is the expression to refer to a local site or network under the control of a single ONOS cluster. Clearly, it is not required that all the local domains are controlled by ONOS, as long as a different choice implements the APIs defined in the communication channel between the local and the global.
The global node has three main logical components:

- **A service orchestrator** exports northbound APIs to off-platform orchestrators and splits service requests into instructions targeting the virtual devices the global node is aware of.

- **A virtual topology provider** receives notifications form the underlying domains about devices, ports and inter-connection links between devices.
- **A communication channel** to talk with the underlying domain controllers. The platform is not bound to a single transport protocol very much like we thought the communication mechanism for ICONA (see 3.2.2).

While in each local controller we have:

- **A topology aggregation mechanism** to aggregate topology elements into virtual topology data structures.
- **A communication channel** to talk with the global controller to notify topology elements and receives network provisioning requests.
- **A domain-specific network provisioning** application implements the network provisioning (forwarding, filtering rules, policing etc.) interacting with physical devices.

The latter component is different in each domain while the topology aggregation mechanism and the communication channel are common to all local controllers. The discussion follows by considering each single logical component of the platform.

5.2.1 Topology Abstraction

The global node maintains an abstract view of the underlying topology for sake of scalability and to separate domain-specific and domain-agnostic concerns. For each local controller, a *BigSwitchSerive* component exposes one abstract device to the global node: it represents an aggregation of the real network elements that compose the topology of a local site. This way, the global ONOS has fewer devices and link data structures to deal with. Path computation will involve only these aggregated items, while the actual network provisioning will be achieved by the local site controllers. The relevant topology information for the global node are the connect points representing the demarcation line between a Service Provider and its customers network, the connect points between two Service Provider networks, and relative ports characterisation relevant to the services deployed at the global level, such as bandwidth for an admission control scheme and the geographical coordinates

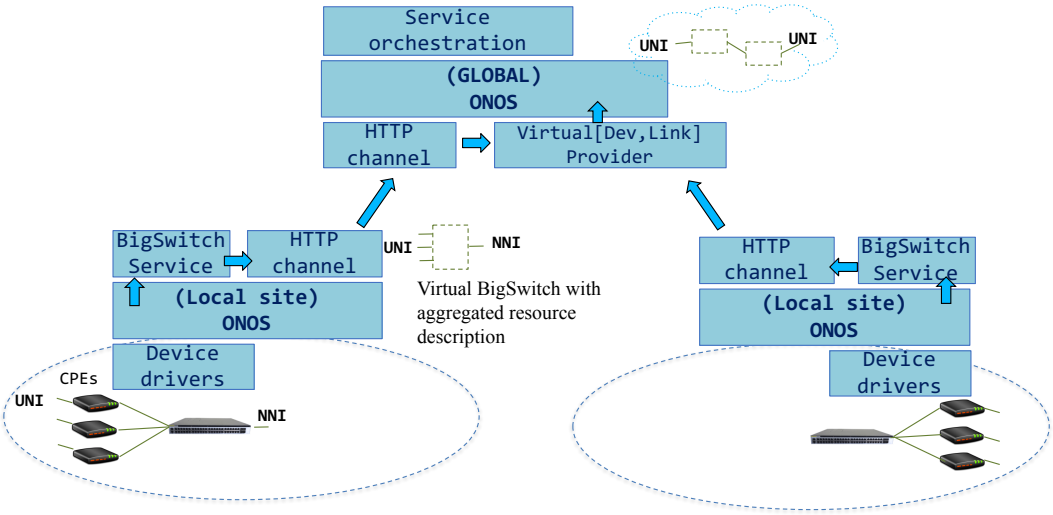


Fig. 5.1 Bottom-up topology discovery phase.

to display the connect points into a graphical map.

The bottom up procedure depicted in Fig. 5.1 shows the topology discovery phase where the local controller's *BigSwitchService* aggregates the physical devices into a single device data structure with related relevant connect points to expose to the global controller via an HTTP channel. In the picture, the connect points are marked as UNI (User-to-Network Interface) for those facing the customer side and NNI (Network-to-Network Interfaces) for those neighboring with an external network, following the terminology adopted by the MEF consortium [70]. This stems from our primal use case of the platform that was the Carrier Ethernet service orchestration. The *BigSwitchSerive* is responsible to apply the one-to-one mapping between physical and virtual connect points and to notify the global about those changes in the local topology that would affect the aggregated virtual topology; it *listens* for events of the local topology and propagates events related to the virtual topology using the well-known Whiteboard Pattern of OSGi.

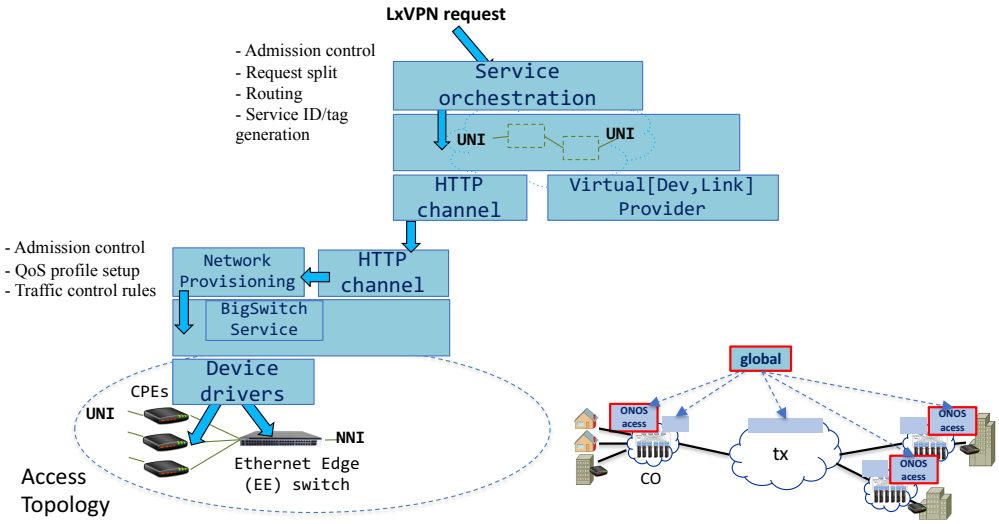


Fig. 5.2 Top down service request elaboration.

5.2.2 Service Orchestration

The service orchestration is the most generic part of the platform in the sense that different connectivity services could be implemented here, exploiting the virtual topology stored in the global node. A service implementation is accompanied by a proper extension of the communication channel to enable data exchange and a proper implementation of the domain-specific network provisioning application within the local controllers. In Fig. 5.2 it is depicted the elaboration of a generic LxVPN request: the common actions applied in the global node are the admission control routine, the split of the request into rules for the virtual devices, path computation and isolation of target traffic. We will talk a bit more of a specific service implementation in Sect. 5.3 about Carrier Ethernet service.

5.2.3 The Communication Channel

There are different options to implement a communication mechanism for data exchange between controllers. Our primal abstractions are defined through java APIs, decoupled by the actual communication protocol. The simplest and fastest way to design and implement the communication was via REST, because the Apache Karaf container is delivered with the HTTP server Jetty, on top of which a framework based

on the *javax* library to build REST applications is available in ONOS; so we took that choice: a REST server in the global controller to sense topology events from the underlying domains and a REST server in the local controllers to receive service requests.

The client side of the communication is a single-threaded executor OSGi component. In the global node, it is actuated every time a service request needs to be propagated to a remote domain. In the local nodes, the executor is actuated upon reception of topology events posted by the *BigSwitchService* mentioned above. The HTTP client component implements as many Java APIs as the number of specific services offered by the platform and registers itself as a listener object. When a service request arrives to the global node, the request is typically split into multiple instructions targeting the virtual domain devices and, for each device, the service orchestration calls all the registered listeners among which only the one that implements the communication with the domain the device belongs to processes the instructions for that device. The remote domain endpoints, IP, port and credentials, are given by configuration. Choosing a client/server REST channel gave us the opportunity to prototype quickly the platform and to use it in E-CORD for field trials in different Telco laboratories with a variety of network devices.

Even if the stream of application data is bidirectional, topology information in one direction and service data in the other, they present different characteristics. Topology data models are limited and fixed while service data can vary a lot. For stable data, stable APIs are good enough and REST remains a very good choice. For variable data, a model-driven approach is the best candidate (as discussed also in Sect. 2.2). For this reason, an interesting approach to investigate is a compound solution of REST and YANG/Netconf. ONOS implements the client side of the communication with Netconf-enabled devices. This means that for the virtual devices of the global node, we should define a proper driver modelled using yang to express service data to exchange and then use the auto-generated Java classes to insert the logic into the device drivers and send messages via the Netconf subsystem of ONOS. In the local controllers we should simply add a Java implementation of a Netconf server and use the same yang models to parse the messages coming from the global node.

5.2.4 Domain-specific Network Provisioning

This part of the local controllers is the boundary between common abstractions and domain-specific provisioning. It can be compared to the device drivers of ONOS, but instead of interacting with a single physical device, it is an application of ONOS that can use either the NB APIs or directly device behaviours (see Chapt. 2.2) to interact with the whole local topology to fully exploit device capabilities (Fig. 5.2). For each service definition, we define a Java interface implemented by both the client component of the global node and by this component so that encoding from Java to json in the global node and decoding from json to Java in the local nodes of service data is achieved with the same helper classes called codecs and packaged in the same bundle of the service interface class, named following the convention: *organisation.service_name.api*.

The network provisioning component translates the service interface methods into some actions on the network; in the access networks we control Customer Premise Equipments (CPEs), in the Telco's Central Offices we have NFV infrastructures built on top of bare-metal servers inter-connected by white-box switches that replace legacy networking appliances (see next section), on the transport network it is plenty of diverse technologies for the physical, the data-link and IP layer. Fig. 5.2 illustrates an access network with CPEs connected to an Ethernet Edge device; as a counter example, in Fig. 5.3 we show an optical transport domain under the control of our platform. A common procedure to all domain-specific components is the resolution of the actual ingress and egress points associated to whatever service request into the real physical connect points of the local topology. The application does so by querying a method exported by the *BigSwitchService* that maps a virtual port to the physical one and viceversa. Then, if the retrieved physical connect points belong to different physical devices, the network provisioning shall happen via intent-based programming if the actual *Intent subsystem* of the controller support the specific kind of provisioning; otherwise path computation should be applied to the local topology to get the complete list of devices involved. It is not always the case that a service request contains more than a single virtual port, for example, the instantiation of a QoS profile usually targets a single interface.

In the following section we present a concrete use-case of the presented platform.

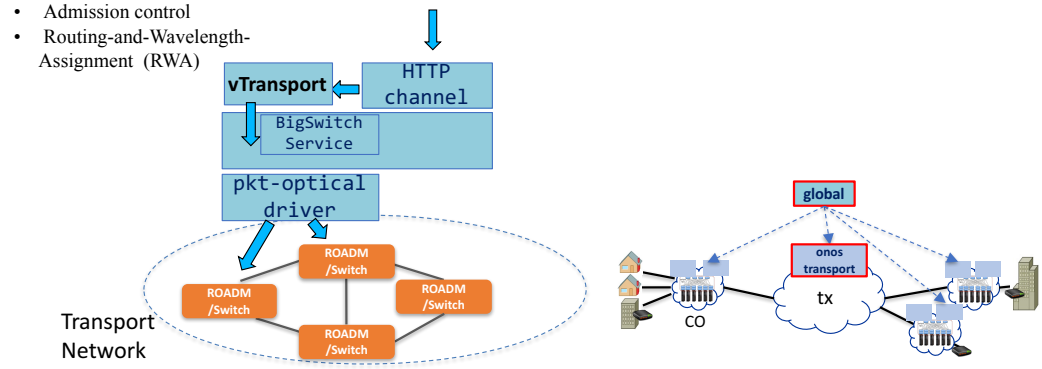


Fig. 5.3 Optical domain transport network

5.3 Enterprise CORD

5.3.1 CORD: Central-Office-Rearchitected-as-a-Datacenter

CORD is an architecture for the Telco Central Office that combines SDN, NFV, and elastic cloud services - all running on commodity hardware - to build cost effective, agile networks with significantly lower CAPEX/OPEX and to enable rapid service creation and monetisation. For a detailed description of the project, the rationale behind its architectures and software components, refers to the white paper [71]. The goal of CORD is not only to replace today's purpose built hardware devices with their more agile software conunter parts, but also to make the Central Office an integral part of every Telco's larger cloud strategy, enabling them to offer more valuable services. The illustrative example of value proposition offered by CORD in Fig. 5.4 shows commodity hardware infrastructures, connected by a leaf-spine fabric for the East-West traffic between the access networks that connect customers to the Central Office and the upstream links that connect the Central Office to the operator's backbone. The NFV and SDN control plane is composed by ONOS, Openstack and XoS as orchestrator [72]. All controller entities run on Docker containers along with the deployed VNFs. On top of the software infrastructure different use-case domains leveraged on CORD: Residential, Mobile and Enterprise.

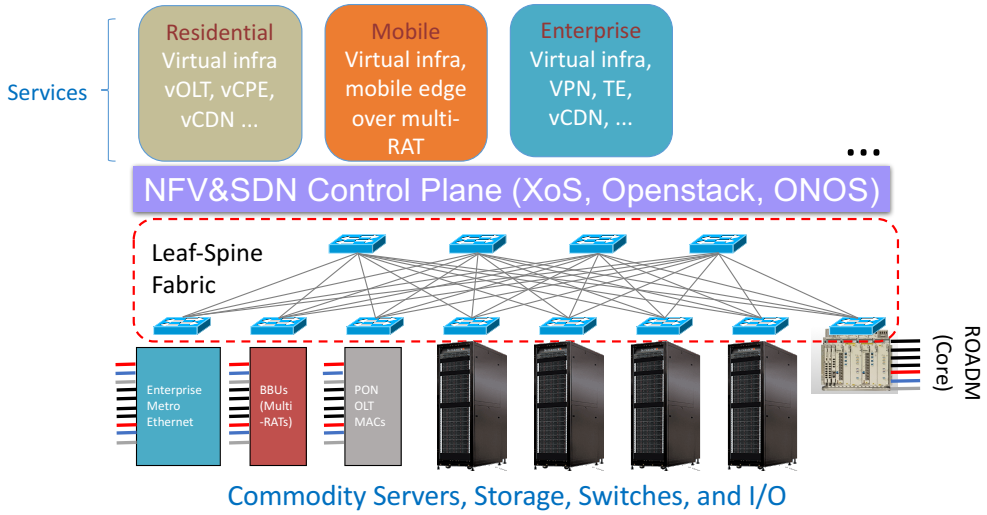


Fig. 5.4 CORD Infrastructure

5.3.2 CORD for Enterprise

Enterprise CORD (E-CORD) is a CORD use-case that offers enterprise connectivity services over metro and wide area networks, using open source software and commodity hardware. E-CORD builds on the CORD infrastructure to support enterprise customers, and allows Service Providers to offer enterprise connectivity services (L2 and L3VPN). It can go far beyond these simple connectivity services, as it includes Virtual Network Functions (VNFs) and service composition capabilities to support cloud-based enterprise services. In turn, enterprise customers can use E-CORD to rapidly create on-demand networks between any number of endpoints or company branches. These networks are dynamically configurable, implying connection attributes and SLAs can be specified and provisioned on the fly. Furthermore, enterprise customers may choose to run network functions such as firewalls, WAN accelerators, traffic analytic tools, virtual routers, etc. as on-demand services that are provisioned and maintained inside the service provider network.

The following is a list of the basic terms used in E-CORD

- Central-Office/Local POD: identified also as E-CORD site, it is a standard CORD POD equipped with specific access equipment, such as an Ethernet

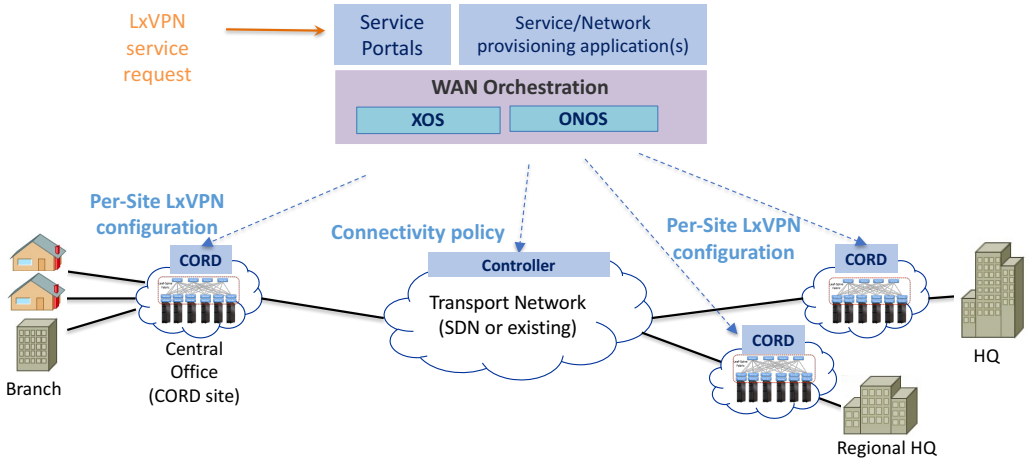
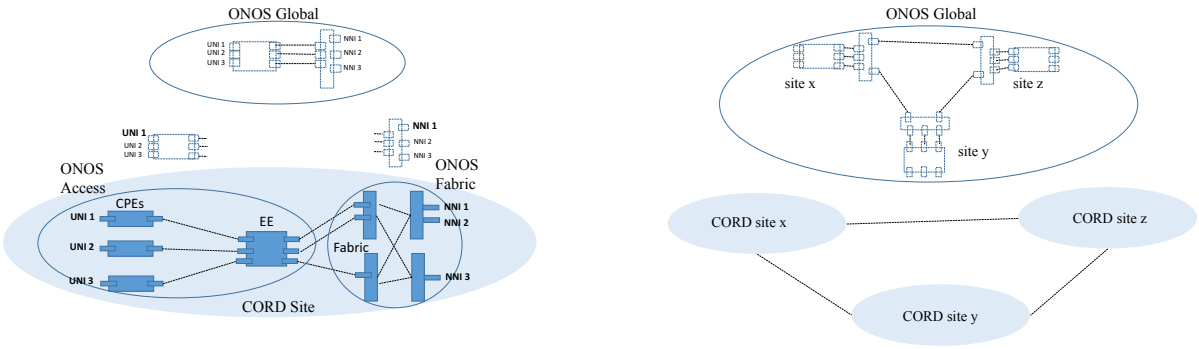


Fig. 5.5 ECORD scenario

edge switch. It is usually located in the Service Providers' Central Offices and is mainly used to: 1) connect the enterprise user to the service provider network; b) run value added user services at the edge of the network, such as firewalls, traffic analytic tools or WAN accelerators. Upstream, the POD connects to the service provider metro/transport network.

- Global node: it is a single machine running either in the cloud, or in any other part of the Service Provider's network, used as general orchestrator that coordinates between all the local PODs of the E-CORD deployment. It is composed by an instance of XoS and one ONOS cluster.

A typical E-CORD deployment is made of one orchestrator global node and multiple (min. 2) CORD sites (PODs), connected to the same transport network (Fig. 5.5). Each site comprises a CORD POD with of one or more compute nodes, and one or more fabric switches. The transport network provides connectivity between the CORD sites. It can be almost anything, from an optical network requiring a converged view of the logical layers to a single packet switch. The transport network can be composed of white-boxes, legacy equipment, or a mix of both. The minimum requirement in order to deploy E-CORD is to provide Layer 2 connectivity between the PODs, specifically between the leaf fabric switches, facing the upstream/metro network of the COs. Usually, for lab trials, the leaf switches of the two sites (PODs)



(a) Single site topology abstraction

(b) Multiple CORD sites

Fig. 5.6 ECORD topology abstraction

get connected directly through a cable, or through a legacy L2 switch, but we successfully integrated ROADM switches, one per site, connected to the leaf switch of the fabric to simulate the optical transport network.

The hierarchical platform described earlier is used as the SDN control plane to connect multiple CORD sites together via Carrier Ethernet circuits established on-demand. Each local site uses two ONOS controllers that are part of the reference architecture of CORD: *ONOS Access* and *ONOS Fabric*. The Carrier Ethernet application of E-CORD uses both controllers to provision the physical network:

- *ONOS Access* runs the application that controls the edge network, including the CPE devices and the Ethernet Edge (EE) devices. In this part of the network it is performed isolation and policing of the customer traffic.
- *ONOS Fabric* runs the application configuring the cross connections within the fabric of CORD to bridge the CPEs to the transport network and eventually to the remote sites. Alternatively, it bridges customer's traffic to a chain of VNFs before being routed to the Internet gateway.

So for each site, the E-CORD application exposes two abstract devices to the global node: one is exposed by *ONOS Access* and the other by *ONOS Fabric* (Fig. 5.6).

The *Service Orchestrator* of the global node is the Carrier Ethernet application that exports APIs to setup, tear down and update Ethernet Virtual Circuits (EVCs) spanning multiple sites. An EVC is identified by a service tag (outer vlan tag of the

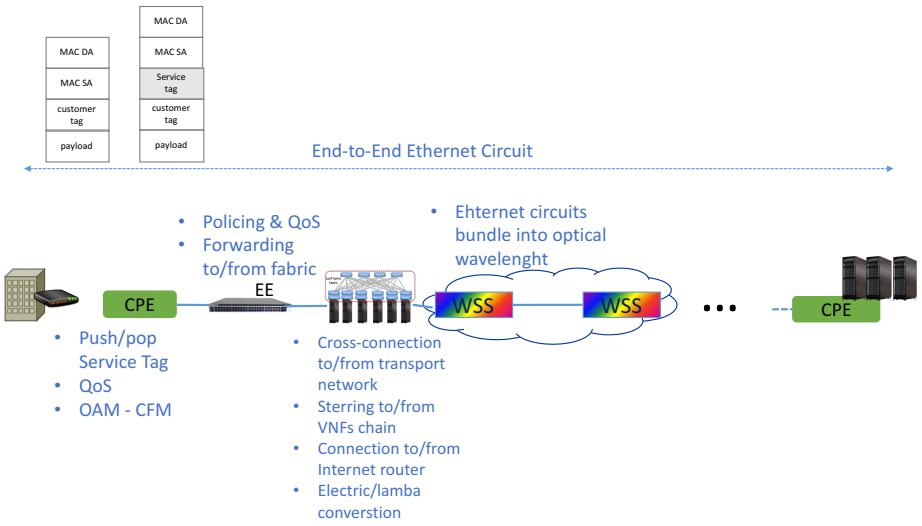


Fig. 5.7 EVC on data path

802.1ad protocol), one or more customer vlan tags (802.1q) mapped to the service tag, a bandwidth profile and a set of UNI ports among which we want to create the layer-2 VPN based on Ethernet. The EVC request is split by the *Service Orchestrator* into as many forwarding constructs as the number of virtual devices along the path between the UNIs. The forwarding constructs are sent to the local controllers which are responsible to allocate the appropriate network resources. The network functions provided in data path are illustrated in Fig. 5.7; the southbound protocols used to control the devices are Netconf for the CPE, Openflow 1.3 for the EE and the fabric switches and OpenFlow 1.3 + Optical Transport Protocol Extensions (ONF TS-022) for the ROADMs. The CPE in use is a custom SFP of Microsemi, the ea1000 featured with an embedded Linux operating system and a FPGA board programmable via Yang/Netconf. This device implements the connection-fault-management (cfm - IEEE 802.1ag) for standard OAM operations. The developers of the device has made available the cfm APIs in ONOS to test end-to-end link properties such as packet loss, delay and delay jitter. The EE is a whitebox switch, the Centec v350. The fabric whitebox switches are EdgeCore 5712 and the ROADM are custom disaggregated appliances provided by TIM.

Using our implementation of hierarchical controllers we are able to establish end-to-end connections on-demand on a composite data path of network devices. The specific Carrier Ethernet service is used to offer dedicated line to enterprise

customers with dynamic SLA that resolves into QoS profiles applied at the edge of the network and in OAM operations for end-to-end link monitoring. You can update a circuit any time via REST-API, for example to augment the amount of bandwidth needed or to add another branch office to the circuit. This makes such scenario extremely attractive for companies that leverages on their own cloud infrastructure to offer their services because it opens the path for dynamic SLA based on real-time traffic load, accelerates revenue and service deployment with operational simplicity (micro-service architecture) and improves service and application performance by extending automation from the data-center to the network.

Chapter 6

Related Work

In this dissertation we introduced system and platform design contributions to the control plane of network infrastructures. We did not cover the analytical problem of the controller placement in large-scale WAN networks, which is treated in [15, 17, 16], although the authors in [16] only consider networks with less than 100 nodes. The platforms described in this dissertation provide virtual networks as a service since the physical network infrastructure is abstracted to the northbound applications as logical nodes and links with associated resources. Virtual network embedding problems are deeply investigated in the literature, in [73–75] the authors report the most recent advances in this field.

We showed how to achieve scalability using the network controller ONOS with ICONA in Chapter 3; then we presented a framework for end-to-end QoS provisioning in Chapter 4 which is based on it. An holistic discussion on scalability in SDN is reported in [76] where it is highlighted that scalability challenges are not restricted or inherently to SDN, rather they are faced in traditional network design too; they remark that SDN by itself is neither likely to eliminate the control plane design complexity or make it more or less scalable, but it allows to rethink the constraints traditionally imposed on control protocol designs, encourages to apply common software and distributed systems development practices and frees the control plane from basic but challenging issues like topology discovery and state distribution. Our contributions leverage on ONOS clustering for scalability and high-availability within a local domain, while between ONOS clusters there is no distribution algorithm but rather an event-driven micro-service communication

interface to notify topology and service events. In [77] the authors propose a novel consensus algorithm to provide a resilient service chain in distributed environment that outperform the Raft consensus algorithm used by ONOS.

To some extent, the most complementary work to the content presented in this dissertation is *Orion* [11], a hybrid hierarchical control plane for flow-based large-scale SDNs. Complementary because it demonstrates analytically how a hybrid hierarchical structure can effectively reduce the computational complexity of the control plane; it explains analytically how to represent an abstract view of the network topology within the root controllers and reports a theoretical evaluation on the computational complexity and how to overcome the *path stretch* problem that appears in hierarchical control plane. On the other side, datapath interoperability is not addressed, their architecture is based on pure Openflow controllers, thus much of the features rely on Openflow messages making it unusable in heterogeneous networks. Beside that, our framework proposal for end-to-end QoS provisioning could apply to *Orion* itself, althtough the prototype implementation they describe is based on Floodlight rather than ONOS, precluding the adoption of the driver subsystem essential for all brownfield deployments and those greenfield deployment not based on Openflow.

In [78–80] it is discussed the QoS provisioning in SDN. In [78] a general framework for dynamic QoS control is presented, the authors mention an inter-domain interface to exchange control and application information between SDN domains for end-to-end control, although it does not provide any insight on the architectural details. In [79] the authors present a QoS-guaranteed approach for bandwidth allocation that satisfies the QoS requirements for all priority cloud users by using Open vSwitch, while the authors in [80] demonstrate how an SDN/Openflow control environment can overcome the limitations of the best effort shortest path routing and IntServ architecture of the Internet. In this regards, their contribution is a motivation to foster design and architectural contribution for end-to-end QoS provisioning in SDN as we did in Chapter 4.

The rest of this chapter reports related work on distributed SDN platform which can be broadly divided into those using a mesh structure and those using a hierarchy of controllers. Another distinction factor is whether they focus on functionalities or on scalability, state distribution and fault tolerance. Except for [81–83], the following

papers focus on the latter aspects rather than functionalities which basically are reduced to forwarding control via Openflow.

ONIX [33] provides an environment on top of which a distributed NOS can be implemented with a logically centralized view. The distributed Network Information Base (NIB) stores the state of network in the form of a graph; the platform is responsible for managing the replication and distribution of the NIB, the applications have to detect and resolve conflicts of network state. Scalability is provided through network partitioning and aggregation. Regarding the fault tolerance, the platform provides the basic functions, while the control logic implemented on top of ONIX needs to handle the failures. ONIX has been used in the B4 network, the private WAN [6] that inter-connects Google's data-centers around the world. Its high level design is similar to ICONA. ICONA however is not tailored to a specific use case, providing a reusable framework on top of which it is possible to build specific applications.

The Kandoo [13] architecture addresses the scalability issue by creating an architecture with multiple controllers: the so-called root controller is logically centralized and maintains the global network state; the bottom layer is composed of local controllers in charge of managing a restricted number of switches. The Kandoo architecture does not focus on the distribution/replication of the root controller and on fault tolerance neither in the data plane nor in the control plane.

HyperFlow [9] focuses on both scalability and fault tolerance. Each HyperFlow instance manages a group of devices without losing the centralized network view. A control plane failure is managed by redirecting the switches to another HyperFlow instance. The applicability of such approach to WAN scenarios with large delays between the different HyperFlow instances is not considered.

DISCO [10] architecture considers a multi-domain environment. This approach is specifically designed to control a WAN environment, composed of different geographical controllers, that exchange summary information about the local network topology and events. This solution overcomes the HyperFlow limitations, however it does not provide local redundancy: in the case of a controller failure, a remote instance takes control of the switches, increasing the latency between the devices

and their primary controller.

ElastiCon [12] and Pratyaastha [14] aim to provide an elastic and efficient distributed SDN control plane to address the load imbalances due to static mapping between switches and controllers and spatial/temporal variations in the traffic patterns.

SMaRtLight [84] considers a distributed SDN controller aiming at a fault-tolerant control plane. It only focuses on control plane failures, assuming that data plane failures are dealt with by SDN applications on top of the control platform. Service Provider-SDN (SP-SDN) [85] envisages for an extended SDN architecture with the introduction of a service orchestration layer which spans several administrative domains supporting both network and cloud services. The multi-domain applications run on top of the service layer. Similarly, Lifecycle Service Orchestration (LSO) [86] proposes an orchestration layer on top of the SDN control layer. Compared to SP-SDN, LSO envisages for a hierarchical orchestration layer. The lower layer has narrow scope, in fact the LSO components run on top of the single SDN controllers in the different domains. On top of this layer there is a global LSO that orchestrates the intra-domain LSOs.

In OpenDaylight [3], an initial work on clustering has been provided in the Helium release using the Akka framework [87] and the RAFT consensus algorithm. Finally, hierarchical SDN Orchestration solutions for WANs are presented in [81] and [88]; in [88] the authors discuss general design considerations and alternatives when considering multiple controllers in a parent-child relationship, while in [81] they focus on the orchestration of heterogeneous technologies, referred as domains, of the underlying network.

In [82, 83] the authors highlight the opportunities of having programmable control and management functions at a number of layers, allowing applications to control network resources and information across different technology domains, in particular they focus on data-centers Ethernet and optical transport domains. The control plane presented in [82] is intended for intra and inter data-centers networks using Openflow within the data-centers and GMPLS for the carrier networks. The hierarchy in the control plane is based on PCE [65] and the data path nodes is

composed by Openflow-controlled ROADMs within the data-center and GMPLS-controlled SSON (Spectrum Switched Optical Network) for the transport network. The challenge they highlight stands in using SDN to simplify and better integrate operational and business systems, by means of open and standard interfaces and the use of existing functional entities. GMPLS already provided carrier-grade and multi-domain support with flexibility and automation but it has no user-friendly, high-level abstraction to be used for business and operation activities. A more detailed paper on SDN for optical domain networks is [83]. A unified control plane architecture based on OpenFlow for optical SDN tailored to cloud services is introduced. First, they highlight the potential of having programmable networks in a multi-technology and multi-domain environment, then they present a general architecture based on Openflow capabilities for intra and inter data-centers network infrastructures; to this regards, the goal of the paper is very close to the main goal of this dissertation, however their focus is on the packet-switched and optical-circuits integration rather than on the software platform design and implementation to address the interoperability in the multi-technology, multi-domain environment they mention. The ECORD platform described in Chapter 5 includes the integration of the optical domain via the Openflow extensions for the transport domain, but we do also include full control of the telco Central-Office data-center fabric and the customer equipment premises via in-band Netconf control sessions.

Chapter 7

Conclusion

Contributions.
In this dissertation we have presented two frameworks based on ONOS for the control plane of network infrastructures. The design and implementation aspects of each of them leverage on the knowledge and the lessons learnt from the previous work.

In Chapter 4 we presented a resource reservation scheme for end-to-end QoS provisioning. We analysed all the essentials aspects of the framework application running on top of a centralised network controller: the admission control, the inter-domain communication required to achieve the end-to-end guarantee, the interaction with the core controller components and the employment of software drivers to decouple the functional intents from the device-specific traffic control rules. A lot remains to explore though: the integration between the flow-based and class-based QoS within the controller, the automation of a policy-driven mechanism to enable dynamic SLA and an investigation focusing exclusively on the communication design between domains.

In Chapter 5 we described the architecture of a hierarchical SDN platform suitable for end-to-end network services provisioning. The platform incorporated in E-CORD is a step ahead towards the adoption of open-source software for Service Provider systems. It is used to connect multiple CORD sites together; in each site, a small data-center made up of commodity hardware and white box switches replaces legacy premises of the Telco's Central Offices, thus building a highly distributed cloud infrastructure spanning wide area networks. The platform is in continuous

development, improvements and integrated testing under the ONF sponsorship. Thanks to the PhD activity we contributed to the project and we were able to reuse knowledge and code from the ICONA project (Chapter 3) to prototype the SDN platform of ECORD.

Our hope is that concepts and ideas herein presented could be a useful contribution for the concretisation of what is deeply investigated in the literature. We expect to see a convergence among the current efforts coming from multiple IT organisations, academies and industries, to lay the foundations of a widely acknowledged reference architecture for the Service Providers' networks control plane, as much as the Linux operating system has become for personal computers, general purpose devices and embedded systems.

Considerations and open issues.
Some considerations are to be given about *complexity*, *generality* and *reliability* in SDN for WANs.

ONOS, Opendaylight and the prototype work for distributed control plane mentioned in the related work chapter are designed to host network applications on top of them; but what we have realised in these three years of activities is that the only reasonable way to achieve progress in building SDN applications on top of such platforms is to reason by use-cases and isolate them during the implementation, thus avoiding the *generality* the platform is designed for. The coexistence of different use-cases within the single platform increases the overall system *complexity* so much so that sometimes the feasibility itself is compromised. In CORD, for example, there are two ONOS involved as SDN controllers: one dedicated solely to the data-center fabric control and the other to the virtual tenant networks for service chaining. All the use-cases provided in ONOS (SDN-IP, packet-optical, routing protocols etc.) could hardly coexist. Stated in another way, it is fairly easy to abstract device capabilities and behaviours via common set of APIs, it is much harder to make network services coexisting within the same SDN platform via slicing and virtual views; this is especially true if the platform runs as a single process entity with complete resource sharing as in the case of ONOS and Opendaylight running in the JVM as OSGi platforms, no matter if distributed. OSGi provides modularity and dynamic service activation, but certainly it does not provide resource isolation and data protection. The core functions of these controllers, such as topology and flow management and related stores, live together in the same process and scope,

making the platform monolithic rather than micro modular. The model-driven approach adopted by Opendaylight expects applications to be model-aware making the generality of the platform even less conceivable. A novel design of such SDN platform is certainly a topic of investigation.

By using distributed controllers, either in a hierarchy or in a mesh, one can reduce the algorithmic time complexity w.r.t. legacy distributed protocols for, e.g., routing and best path selection, but the system and software complexity remains high and it can further increases w.r.t. a logically centralised controller. This is because you are increasing the chance of having inconsistent states between controllers. For the same reason *reliability* is still a general concern in SDN, because if it is true that automation reduces the risk of manual faults, it also introduces new risks on the virtual layer, exposure to bugs, malware and potential incompatibility with drivers, protocols, firmware and other pieces of the data stack. SDN does not provide reliability as a core asset, it does provide automation and flexibility, so it has to be designed and programmed very carefully.

References

[1] Opencord website - https://opencord.org.

[2] Onos website - http://onosproject.org/.

[3] Opendaylight - http://www.opendaylight.org/.

[4] Opnfv website - https://www.opnfv.org/.

[5] Openroadm website - http://www.openroadm.org/home.html.

[6] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, 43(4):3–14, August 2013.

[7] Pankaj Berde, Matteo Gerola, Jonathan Hart, Yuta Higuchi, Masayoshi Kobayashi, Toshio Koide, Bob Lantz, Brian O'Connor, Pavlin Radoslavov, William Snow, and Guru Parulkar. Onos: Towards an open, distributed sdn os. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM.

[8] J. Medved, R. Varga, A. Tkacik, and K. Gray. Opendaylight: Towards a model-driven sdn controller architecture. In *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014*, pages 1–6, June 2014.

[9] A. Tootoonchian and Y. Ganjali. Hyperflow: a distributed control plane for openflow. In *2010 internet network management conference on Research on enterprise networking*, 2010.

[10] K. Phemius, M. Bouet, and J. Leguay. Disco: Distributed multi-domain sdn controllers. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–4, May 2014.

[11] Y. Fu, J. Bi, Z. Chen, K. Gao, B. Zhang, G. Chen, and J. Wu. A hybrid hierarchical control plane for flow-based large-scale software-defined networks. *IEEE Transactions on Network and Service Management*, 12(2):117–131, June 2015.

[12] A.A. Dixit, F. Hao, S. Mukherjee, T.V. Lakshman, and R. Kompella. Elasticon: an elastic distributed sdn controller. In *Tenth ACM/IEEE symposium on Architectures for networking and communications systems*, 2014.

[13] H. Y. Soheil and Y.Ganjali. Kandoo: a framework for efficient and scalable offloading of control applications. In *First workshop on Hot topics in in software defined networking*, 2012.

[14] A Krishnamurthy, S. P. Chandrabose, and A. Gember-Jacobson. Pratyaastha: an efficient elastic distributed sdn control plane. In *Third workshop on Hot topics in in software defined networking*, 2014.

[15] S. Lange, S. Gebert, J. Spoerhase, P. Rygielski, T. Zinner, S. Kounev, and P. Tran-Gia. Specialized heuristics for the controller placement problem in large scale sdn networks. In *2015 27th International Teletraffic Congress*, pages 210–218, Sept 2015.

[16] B. Heller, R. Sherwood, and N. McKeown. The controller placement problem. In *First workshop on Hot topics in in software defined networking*, 2012.

[17] Stefan Schmid and Jukka Suomela. Exploiting locality in distributed sdn control. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 121–126, New York, NY, USA, 2013. ACM.

[18] M. Canini, P. Kutnetsov, D. Levin, and S. Schmid. A distributed and robust sdn control plane for transactional network updates. In *The 34th Annual IEEE International Conference on Computer Communications*, 2015.

[19] Bob Braden, Lixia Zhang, Steve Berson, Shai Herzog, and Sugih Jamin. Resource reservation protocol (rsvp) – version 1 functional specification. RFC 2205, RFC Editor, September 1997. http://www.rfc-editor.org/rfc/rfc2205.txt.

[20] Ecord codebase - https://gerrit.opencord.org/admin/projects/carrierethernet.

[21] Onf website - https://www.opennetworking.org.

[22] Raft consensus algorithm - https://raftconsensus.github.io/.

[23] Apache karaf - http://karaf.apache.org/.

[24] Openflow switch specification version 1.5.0 - https://www.opennetworking.org/images/stories/ downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf.

[25] David Erickson. The Beacon OpenFlow Controller. In *HotSDN*. ACM, 2013.

[26] Floodlight - http://www.projectfloodlight.org/floodlight/.

[27] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. Nox: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[28] Pox - https://github.com/noxrepo/pox.

[29] Ryu - http://osrg.github.io/ryu/.

[30] Stefan Wallin and Claes Wikström. Automating network and service configuration using netconf and yang. In *Proceedings of the 25th International Conference on Large Installation System Administration*, LISA'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[31] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, Keith Amidon, and Martin Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[32] M. Gerola, F. Lucrezia, M. Santuari, E. Salvadori, P. L. Ventre, S. Salsano, and M. Campanella. Icona: A peer-to-peer approach for software defined wide area networks using onos. In *2016 Fifth European Workshop on Software-Defined Networks (EWSDN)*, pages 37–42, Oct 2016.

[33] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A distributed control platform for large-scale production networks. In *9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[34] M. Caesar and J. Rexford. Bgp routing policies in isp networks. *IEEE Network*, 19(6):5–11, Nov 2005.

[35] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *9th ACM Workshop on Hot Topics in Networks*, 2010.

[36] Netem - http://www.linuxfoundation.org/collaborate/ workgroups/networking/netem.

[37] Geant - the core national research and education networks (nrens) european backbone - http://www.geant.net/pages/default.aspx.

[38] F. Lucrezia, G. Marchetto, M. Gerola F. Risso, and M. Santuari. A proposal for end-to-end qos provisioning in software-defined networks. *International Journal of Electrical and Computer Engineering*, 7(4), 2017.

[39] G. P. Fettweis. The tactile internet: Applications and challenges. *IEEE Vehicular Technology Magazine*, 9(1):64–70, March 2014.

[40] Xipeng Xiao and L. M. Ni. Internet qos: A big picture. *Netwrk. Mag. of Global Internetwkg.*, 13(2):8–18, March 1999.

[41] D. H. Lorenz and A. Orda. Qos routing in networks with uncertain parameters. *IEEE/ACM Transactions on Networking*, 6(6):768–778, Dec 1998.

[42] R. Guerin and A. Orda. Qos based routing in networks with inaccurate information: theory and algorithms. In *INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution., Proceedings IEEE*, volume 1, pages 75–83 vol.1, Apr 1997.

[43] Pawan Goyal, Simon S. Lam, and Harrick M. Vin. Determining end-to-end delay bounds in heterogeneous networks. In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video*, NOSSDAV '95, pages 273–284, London, UK, UK, 1995. Springer-Verlag.

[44] D. C. Verma, H. Zhang, and D. Ferrari. Delay jitter control for real-time communication in a packet switching network. In *Proceedings of TRICOMM '91: IEEE Conference on Communications Software: Communications for Distributed Applications and Systems*, pages 35–43, Apr 1991.

[45] Hu Jia and Zhou Jinhe. The design of finegrained network qos controller and performance research with network calculus. *TELKOMNIKA Indonesian Journal of Electrical Engineering*, 12(6):4468–4474, 2014.

[46] A. R. Bashandy, E. K. P. Chong, and A. Ghafoor. Generalized quality-of-service routing with resource allocation. *IEEE Journal on Selected Areas in Communications*, 23(2):450–463, Feb 2005.

[47] Xin Yuan and Xingming Liu. Heuristic algorithms for multi-constrained quality of service routing. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 2, pages 844–853 vol.2, 2001.

[48] Rosario G. Garroppo, Stefano Giordano, and Luca Tavanti. A survey on multi-constrained optimal path computation: Exact and approximate algorithms. *Comput. Netw.*, 54(17):3081–3107, December 2010.

[49] F. Kuipers, P. Van Mieghem, T. Korkmaz, and M. Krunz. An overview of constraint-based path selection algorithms for qos routing. *IEEE Communications Magazine*, 40(12):50–55, Dec 2002.

[50] A. Juttner, B. Szviatovski, I. Mecs, and Z. Rajko. Lagrange relaxation based method for the qos routing problem. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 2, pages 859–868 vol.2, 2001.

[51] Hui Zang, Jason P Jue, Biswanath Mukherjee, et al. A review of routing and wavelength assignment approaches for wavelength-routed optical wdm networks. *Optical Networks Magazine*, 1(1):47–60, 2000.

[52] Liu Hui. A novel qos routing algorithm in wireless mesh networks. *TELKOMNIKA Indonesian Journal of Electrical Engineering*, 11(3):1652–1664, 2013.

[53] John Wroclawski. The use of rsvp with ietf integrated services. RFC 2210, RFC Editor, September 1997. http://www.rfc-editor.org/rfc/rfc2210.txt.

[54] Steven Blake, David L. Black, Mark A. Carlson, Elwyn Davies, Zheng Wang, and Walter Weiss. An architecture for differentiated services. RFC 2475, RFC Editor, December 1998. http://www.rfc-editor.org/rfc/rfc2475.txt.

[55] Technical report.

[56] E. Rosen and Y. Rekhter. Bgp/mpls ip virtual private networks (vpns). RFC 4364, RFC Editor, February 2006.

[57] L. Berger, D. Gan, G. Swallow, P. Pan, F. Tommasi, and S. Molendini. Rsvp refresh overhead reduction extensions. RFC 2961, RFC Editor, April 2001.

[58] Scott Shenker and Lee Breslau. Two issues in reservation establishment. *SIGCOMM Comput. Commun. Rev.*, 25(4):14–26, October 1995.

[59] Scott Shenker, Craig Partridge, and Roch Guerin. Specification of guaranteed quality of service. RFC 2212, RFC Editor, September 1997. http://www.rfc-editor.org/rfc/rfc2212.txt.

[60] Debasis Mitra. Stochastic theory of a fluid model of producers and consumers coupled by a buffer. *Advances in Applied Probability*, 20(3):646–676, 1988.

[61] Soohan Ahn and V. Ramaswami. Fluid flow models and queues—a connection by stochastic coupling. *Stochastic Models*, 19(3):325–348, 2003.

[62] M. Alasti, B. Neekzad, J. Hui, and R. Vannithamby. Quality of service in wimax and lte networks [topics in wireless communications]. *IEEE Communications Magazine*, 48(5):104–111, May 2010.

[63] J. Costa-Requena. Sdn integration in lte mobile backhaul networks. In *The International Conference on Information Networking 2014 (ICOIN2014)*, pages 264–269, Feb 2014.

[64] A. Farrel, J.-P. Vasseur, and J. Ash. A path computation element (pce)-based architecture. RFC 4655, RFC Editor, August 2006. http://www.rfc-editor.org/rfc/rfc4655.txt.

[65] JP. Vasseur and JL. Le Roux. Path computation element (pce) communication protocol (pcep). RFC 5440, RFC Editor, March 2009. http://www.rfc-editor.org/rfc/rfc5440.txt.

[66] Kiyohito Yoshihara, Manabu Isomura, and Hiroki Horiuchi. Distributed policy-based management enabling policy adaptation on monitoring using active network technology. 2001.

[67] Leonidas Lymberopoulos, Emil Lupu, and Morris Sloman. An adaptive policy-based framework for network services management. *J. Netw. Syst. Manage.*, 11(3):277–303, September 2003.

[68] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. Boutaba. Policycop: An autonomic qos policy enforcement framework for software defined networks. In *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*, pages 1–7, Nov 2013.

[69] Ecord guide - https://guide.opencord.org/profiles/ecord/.

[70] Mef - https://www.mef.net/resources/technical-specifications.

[71] Cord white paper - http://opencord.org/wp-content/uploads/2016/03/cord-whitepaper.pdf.

[72] Larry Peterson, Scott Baker, Marc De Leenheer, Andy Bavier, Sapan Bhatia, Mike Wawrzoniak, Jude Nelson, and John Hartman. Xos: An extensible cloud operating system. In *Proceedings of the 2Nd International Workshop on Software-Defined Ecosystems*, BigSystem '15, pages 23–30, New York, NY, USA, 2015. ACM.

[73] F. Esposito, I. Matta, and Y. Wang. Vinea: An architecture for virtual network embedding policy programmability. *IEEE Transactions on Parallel and Distributed Systems*, 27(11):3381–3396, Nov 2016.

[74] Fady Samuel, Mosharaf Chowdhury, and Raouf Boutaba. Polyvine: policy-based virtual network embedding across multiple domains. *Journal of Internet Services and Applications*, 4(1):6, Mar 2013.

[75] F. Esposito, D. Di Paola, and I. Matta. On distributed virtual network embedding with guarantees. *IEEE/ACM Transactions on Networking*, 24(1):569–582, Feb 2016.

[76] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali. On scalability of software-defined networking. In *IEEE Communications Magazine 51 (2), pp. 136-141*, 2013.

[77] Flavio Esposito. Catena: A distributed architecture for robust service function chain instantiation with guarantees. *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–9, 2017.

[78] I. Bueno, J. I. Aznar, E. Escalona, J. Ferrer, and J. A. Garcia-Espin. An opennaas based sdn framework for dynamic qos control. In *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*, pages 1–7, Nov 2013.

[79] A. V. Akella and K. Xiong. Quality of service (qos)-guaranteed network resource allocation via software defined networking (sdn). In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 7–13, Aug 2014.

[80] S. Tomovic, N. Prasad, and I. Radusinovic. Sdn control framework for qos provisioning. In *2014 22nd Telecommunications Forum Telfor (TELFOR)*, pages 111–114, Nov 2014.

[81] R. Vilalta, A. Mayoral, R. Munoz, R. Casellas, and R. Martinez. Hierarchical sdn orchestration for multi-technology multi-domain networks with hierarchical abno. In *Optical Communication (ECOC), 2015 European Conference on*, pages 1–3, Sept 2015.

[82] R. Casellas, R. Munoz, R. Martinez, R. Vilalta, L. Liu, T. Tsuritani, I. Morita, V. Lopez, O. Gonzalez de Dios, and J. P. Fernandez-Palacios. Sdn based provisioning orchestration of openflow/gmpls flexi-grid networks with a stateful hierarchical pce. In *OFC 2014*, pages 1–3, March 2014.

[83] Mayur Channegowda, Reza Nejabati, and Dimitra Simeonidou. Software-defined optical networks technology and infrastructure: Enabling software-defined optical network operations

invited

. *J. Opt. Commun. Netw.*, 5(10):A274–A282, Oct 2013.

[84] F Botelho, A Bessani, F Ramos, and P Ferreira. *SMaRtLight: A Practical Fault-Tolerant SDN Controller.* ArXiv e-prints, 2014.

[85] James Kempf, Martin Korling, Stephan Baucke, Samy Touati, Victa McClelland, Ignacio Mas, and Olof Backman. Fostering rapid, cross-domain service innovation in operator networks through service provider SDN. In *IEEE International Conference on Communications, ICC 2014, Sydney, Australia, June 10-14, 2014*, pages 3064–3069, 2014.

[86] A. Mayer and S. Mansfield. The third network: Lifecycle service orchestration vision. Technical report, MEF, 02 2015.

[87] Akka framework - http://akka.io/.

[88] R. Ahmed and R. Boutaba. Design considerations for managing wide area software defined networks. *IEEE Communications Magazine*, 52(7):116–123, July 2014.

Appendix A

ONOS Driver based on YANG Data Model compiled with BUCK

This appendix explains the steps required to develop a driver in ONOS based on YANG models via auto-generation of Java classes from such models. This procedure enables auto-generated code to be used in the driver after the yang files have been compiled via the *YANG Compiler* module. The procedure is based on ONOS version 1.12.0. The build tool is BUCK.

The required steps are the following:

1. Create a new directory that has the name of the subject driver under *onos/-models*, e.g. create *onos/models/mydriver*. Under this directory create the following path: */src/main/yang*. In the leaf directory, *yang*, put your model files plus any other model file that is imported by the original model files.

2. In *onos/models/mydriver* insert a BUCK file with the following content:

```
yang_model(
app_name = 'org.onosproject.models.mydriver',
title = 'mydriver YANG Model',
)
```

3. Create a new directory that has the name of the subject driver under *onos/-drivers*, e.g. *onos/drivers/mydriver*. Within this directory, put your driver

implementation code under *src/main/java/path/to/package*' together with a *package-info.java* file.

4. Create a BUCK file in *onos/drivers/mydriver* with your dependencies in compilation, test and run time. A base BUCK file has the following content:

```
COMPILE_DEPS = [
'//lib:CORE_DEPS',
'//lib:ONOS_YANG',
'//drivers/utilities:onos-drivers-utilities',
'//models/nomedriver:onos-models-nomedriver',
] + YANG_TOOLS

TEST_DEPS = [
'//lib:TEST_ADAPTERS',
'//utils/osgi:onlab-osgi-tests'
]

APPS = [
'org.onosproject.yang',
#       'org.onosproject.yang-gui',
'org.onosproject.models.nomedriver'
]

osgi_jar_with_tests (
deps = COMPILE_DEPS,
test_deps = TEST_DEPS
)

onos_app (
app_name = 'org.onosproject.drivers.nomedriver',
title = 'nomedriver sample device driver',
category = 'Drivers',
url = 'http://onosproject.org',
description = 'ONOS nomedriver device driver.',
```

```
    required_apps = APPS,
    )
```

Libraries dependencies shall be added in “COMPILE_DEPS” and “TEST_DEPS" if required. Dependencies on ONOS applications stay under “APPS".

5. Adds the entries of the driver and of the model to *onos/modules.def* file. To write the entry it is necessary to replace the backslash "/" with a dash "-" and to add "-oar" as suffix, e.g. *onos/drivers/mydriver* becomes *onos-drivers-mydriver-oar*. In our example we are required to add in *onos/modules.def* the following:

```
ONOS_DRIVERS = [
...

  '//drivers/nomedriver:onos-drivers-mydriver-oar'

...
]

MODELS = [
...

'//models/nomedriver:onos-models-nomedriver-oar'

...
]
```

6. Run BUCK to execute the build of ONOS from the root directory of the ONOS codebase:

```
$ tools/build/onos-buck build onos --show-output
```

If the build is successful, the generated Java files are located in *onos/buck-out/gen/models/mydriver/onos-models-mydriver-yang#srcs__yang-gen* so that you can import them into your driver.