ASSESSMENT OF DISAGGREGATING THE SDN CONTROL PLANE

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Adib Rastegarnia

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2019

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

I would like to dedicate this dissertation to my beloved wife, *Fatemeh Rouzbeh*, who provided the inspiration necessary for me to complete this process and also sacrificed immensely along the way, to *my parents and brothers* for their endless support in this journey, and to anyone who loves computer networks.

ACKNOWLEDGMENTS

First and foremost, I would like to express my sincere gratitude to my advisor Prof. Douglas Comer for his continuous support, mentorship, and encouragement throughout my PhD study and research work. He gave me freedom to define my PhD project that is essential for training of an independent researcher and at the same time he supervised me to find and solve the best research topics. I learned from him to think critically without getting caught up in tiny and irrelevant technical details and to present my research ideas concisely and elegantly. In addition his technical and editorial advice was essential to complete this dissertation and has taught me innumerable lessons and insights on doing academic research in general. Prof Comer has been, and will remain a role model for me in my professional life and I am forever in his debt.

My sincere thanks must also go to my PhD committee members, Dr. Kihong Park, Dr. Roopsha Samanta, and Dr. Sanjay Rao for providing constructive feedback and insightful discussions that significantly improved the quality of this dissertation.

I would like to thank cooperate partners of Computer Science Department at Purdue University who funded my research for one academic year that helped me a lot to focus on my research.

I thank graduate office of Computer Science Department for their support during my PhD and selected me for Computer Science Teaching Fellow Program. The teaching fellow program was an amazing teaching experience for me that gave me the opportunity to improve my teaching skills. I also thank my mentors Prof Douglas Comer and Prof He Wang that shared their teaching experiences with me that helped me a lot to make my lectures more productive and useful for the students.

I would like to thank my wife, Fatemeh Rouzbeh, for her unbounded love and support throughout my graduate school journey. I would like to thank my parents and brothers for their love and support. I would like to thank my friend and lab mate, Rajas H.Karandikar, has offered me invaluable advice, expertise, and encouragement. Last, but not least, special thank to Mr. Michael Motuliak, Mr. Daniel Trinkle, and the entire hardware team in the Compute Science Department at Purdue University, for providing us with equipment for our projects and helping us to setup SDN testbeds.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

ARP	Address Resolution Protocol
API	Application Programming Interface
CPU	Central Processing Unit
DDOS	Distributed Denial of Service
DOS	Denial of Service
ESB	Enterprise Service Bus
FPGA	Field-Programmable Gate Array
FRP	Functional Reactive Programming
FSM	Finite State Machine
gNMI	gRPC Network Management Interface
gNOI	gRPC Network Operation Interface
HAL	Hardware Abstraction Layer
HTTP	Hyper Text Transfer Protocol
IDL	Interface Description Language
IETF	Internet Engineering Task Force
IP	Internet Protocol
JSON	Java Script Object Notation
LLDP	Link Layer Discovery Protocol
MAC	Media Access Control
MIB	Management Information Base
NAT	Network Address Translation
NB	Northbound
NETCONF	Network Configuration Protocol
OVSDB	Open Vswitch Database Management Protocol

P4	Programming Protocol-Independent Packet Processors
PAD	Programmable Abstraction of Datapath
PDP	Programmable Data Plane
POF	Protocol Obvious Forwarding
QoS	Quality of Service
REST	Representational State Transfer
ROFL	Revised OpenFlow Library
RPC	Remote Procedure Call
SB	Southbound
SDN	Software Defined Networking
SNMP	Simple Network Management Protocol
SOA	Service Oriented Architecture
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
VLAN	Virtual Local Area Network
WLAN	Wireless Local Area Network
XML	Extensible Markup Language

ABSTRACT

Rastegarnia, Adib Ph.D., Purdue University, December 2019. Assessment of Disaggregating the SDN Control Plane. Major Professor: Douglas Comer.

Current SDN controllers have been designed based on a monolithic approach that integrates all of services and applications into one single, huge program. The monolithic design of SDN controllers restricts programmers who build management applications to specific programming interfaces and services that a given SDN controller provides, making application development dependent on the controller, and thereby restricting portability of management applications across controllers. Furthermore, the monolithic approach means an SDN controller must be recompiled whenever a change is made, and does not provide an easy way to add new functionality or scale to handle large networks. To overcome the weaknesses inherent in the monolithic approach, the next generation of SDN controllers must use a distributed, microservice architecture that disaggregates the control plane by dividing the monolithic controller into a set of cooperative microservices. Disaggregation allows a programmer to choose a programming language that is appropriate for each microservice. In this dissertation, we describe steps taken towards disaggregating the SDN control plane, consider potential ways to achieve the goal, and discuss the advantages and disadvantages of each. We propose a distributed architecture that disaggregates controller software into a small controller core and a set of cooperative microservices. In addition, we present a software defined network programming framework called Umbrella that provides a set of abstractions that programmers can use for writing of SDN management applications independent of NB APIs that SDN controllers provide. Finally, we present an intent-based network programming framework called OSDF to provide a high-level policy based API for programming of network devices using SDN.

1 INTRODUCTION

1.1 Motivation and Problem Statement

The Internet project is one of the most successful projects in the history of computer networks. The Internet has created a digital society that provides connectivity between almost every pair of digital devices anywhere in the world. In the context of computer networks, network management refers to processes, tools, applications that a network administrator uses for planning, installation, operation, and monitoring of a network. Traditional IP networks are complex and hard to manage despite their widespread adoption [1]. Consequently, network management has changed significantly since the beginning of the Internet. Early in networking history, network administrators configured each network device manually, one at a time, using low-level vendor-specific interfaces to implement the desired high-level network policies. As the Internet grew, traditional *Internet Protocol (IP)* networks become more complex and difficult to manage using conventional network element management tools. In addition, enforcing high-level network policies without using automatic reconfiguration mechanisms became very challenging across IP networks due to the dynamic environment of the Internet.

Another aspect of traditional IP networks that makes their management more challenging arises from *vertically integrated* network devices. Each network device has two major sets of functions: control plane functions that specify how to handle incoming data traffic and data plane functions that specify packet processing mechanisms and details, such as packet classification and forwarding rules that can be derived from control plane functions policies and specification. Vertically integrated traditional network devices means control plane and data plane functions are integrated into a single box, such as a switch or router. The vertical integration does not provide flexibility to network managers nor does it allow them to translate high-level network requirements into programs that update, modify, and add new control and data plane functions.

The weaknesses of conventional IP networks led researchers and engineers to work on a new paradigm, *Software Defined Networking (SDN)*, that allows software to control forwarding state in data plane from a remote location for the design and implementation of next generation of computer networks. The first and most important aspect of SDN network architectures arises from breaking the vertical integration of network devices by decoupling the control and data plane functions from each other. In the SDN paradigm, network devices become simple forwarding devices and the control plane functions are implemented in a logically centralized piece of software called an SDN controller. The SDN paradigm has advantages when compared with conventional IP networks. Some of the most important advantages are:

- SDN allows network managers to implement required network functions by programming network devices using software.
- SDN allows a move from proprietary management interfaces to open *Application Program Interfaces (APIs)*.
- SDN allows a move from network element management to network management and automate network-wide configuration.
- SDN gives the flexibility to program network devices using a cross layer approach.

Although it helps solve many network management problems, SDN introduces new problems. For example, the separation of control and data plane from each other introduces new security threats, such as exploiting centralized SDN controllers that are specific to SDN.

Since SDN emerged as a new paradigm for designing and implementing of next generation of computer networks, multiple research groups and organizations have explored various aspects of SDN. One of the most important aspects of SDN that is addressed by academia and industry lies in the design and implementation of SDN controller software. Open source projects, such as *Open Network Operating System (ONOS)* and OpenDayLight [2, 3], have focused on creating SDN controllers that industry can adopt and deploy in production environments. Researchers also use the controllers in research projects.

Current SDN controllers and associated components exhibit several weaknesses as follows:

- **Monolithic and Proprietary**: The current architecture of SDN controllers is based on a monolithic approach that aggregates all control plane subsystems into a single, huge, monolithic program. In the aggregated control plane model, each controller defines its own set of programming interfaces and services; a programmer can only use the controller's set when creating management applications. The approach makes application development dependent on a particular SDN controller and the specific programming language that has been used to implement the controller, consequently restricting portability of management applications across multiple controllers by making each application depend on a specific controller. Using a monolithic approach for the design of an SDN controller allows a vendor to create a controller and ensure the cohesion of all pieces. However, the approach does not provide an easy way for users to incorporate new services or adopt the controller quickly.

- **Lack of a Uniform Set of Northbound (NB) Application Program Interfaces (APIs)**: Through this dissertation, we will use the term *Northbound (NB)* to refer to the mechanism an application uses when communicating with a controller. Even if they use the same general form of interaction (e.g., RESTful interface), SDN controllers, such as ONOS [4] and OpenDayLight [3], each offer NB APIs that differs from the APIs offered by other controllers in terms of syntax, naming conventions, and resources. The lack of uniformity among NB

APIs makes each external management application dependent on the NB API of a specific SDN controller.

- **Lack of Reusability of Software Modules**: In the current SDN architecture, the dependency between a management application and a specific type of controller limits reuse of SDN software. In many cases, when porting an application from one controller to another, a programmer must completely recode even the basic modules that collect topology information, generate and install flow rules, monitor topology changes, and collect flow rule statistics.

- **Lack of External Reactive SDN Applications**: In the current SDN architecture, programmers can use NB APIs to program network devices *proactively* by building applications that install flow rules to handle all possible cases before data traffic arrives. To exploit the advantages of SDN fully, a network management system must also support a *reactive* approach in which external management applications can be informed of changes in the network or data traffic, and can then react to change forwarding rules. Unfortunately, current SDN controllers do not provide any notification mechanisms that can inform external applications about changes in network conditions.

- **Lack of Scalability and Reliability**: In current SDN controllers, there are strong dependencies between the subsystems. If one of the subsystems fails, the failure affects the operation of other susbsystems. In addition, a monolithic SDN controller is a gigantic and complex software that is difficult to scale because a given subsystem cannot be changed or scaled independently.

- **Low level northbound APIs**: Most of the SDN controllers provide a simplified northbound API such as *Representational State Transfer (REST)* API that programmers use to program network devices by specifying the details of flow rules using *JavaScript Object Notation (JSON)* or *Extensible Markup Language (XML)* file formats. Therefore, a programmer needs to write code that parses

JSON or XML formats to retrieve required information such as network topology, statistics, network parameters and use them for programming of network devices. Although some SDN controllers provide high-level services to make network programming easier, a programmer still needs to deal with low-level flow rule details.

To help overcome the above weaknesses, this dissertation focuses on redesigning the current SDN control plane architecture, and provide mechanisms to achieve the following design goals:

- Migrate from a monolithic control plane design to a microservice control plane architecture that allows an SDN controller to be divided into a set of many, small interconnected microservices instead of a single monolithic application.
- Add support for external reactive applications using facilities other than conventional SDN controller APIs.
- Allow programmers to choose an arbitrary programming language when developing SDN applications without requiring to use the same programming language that was used to implement the controller.
- Increase portability of SDN applications across SDN controllers, and make it easy for a programmer to evaluate a specific application on multiple SDN controllers.
- Provide a new set of abstractions for SDN applications, keeping the abstractions independent of the NB APIs used by specific SDN controllers.
- Provide high-level APIs that allows managers to express network configuration requirements using application based and domain specific network policies, and allows them to write management applications without worrying about low-level details such as flow rule details, network topology related information (e.g end-to-end paths), etc.

- Reduce programming complexity by allowing a programmer to write SDN applications without requiring the programmer to master specific low-level details for each SDN controller, and avoid locking an application to a specific controller.

The following section explains the contributions this dissertation makes toward achieving goals stated above:

1.2 Contributions

The contributions of this dissertation are summarized as follows:

- It proposes a new architecture for next generation of SDN controllers that disaggregates control plane functions into a set cooperative microservices and migrates away from a monolithic to a microservice architecture.
- It discusses the design and implementation of two software defined network programming frameworks to overcome the weaknesses in current frameworks.
- It evaluates the proposed architecture and frameworks from multiple perspectives including functionality and performance.

The contributions can each be explained briefly as follows:

1.2.1 An Architecture for Control Plane Disaggregation

In the first contribution of this dissertation, we introduce the control plane disaggregation concept and explain the advantages of migrating from a monolithic architecture to a microservice architecture in designing and implementing of SDN controllers. In addition, we propose a microservice architecture for the next generation of SDN controllers and introduce its key components. We discuss one of the key components of the proposed architecture, an event distribution system, that allows network event processing to be externalized and allows the control plane to be disaggregated into a set of cooperative microservices. We consider two event distribution systems based on the publish-subscribe and point-to-point messaging models. Furthermore, we implemented the proposed event distribution systems using extant technologies and compared them with each other using performance metrics, ease of use, and functionality.

1.2.2 Umbrella: A Unified Software Defined Development Framework

In the second contribution of this dissertation, we propose *Umbrella*, a unified software defined development framework that provides a set of high-level abstractions that we can use for writing SDN management applications independent of SDN controllers. The Umbrella framework shows how we can build controller-independent services and management applications on top of a set of minimum viable controller components, including southbound protocols and an event distribution system. The Umbrella architecture employs a driver-based approach to provide a set of high-level and controller independent programming abstractions and a set of translation modules to map them into the heterogeneous NB APIs that SDN controllers provide. We explain the architecture of Umbrella framework and the usage of Umbrella APIs by implementing of a few example SDN management applications. Furthermore, we use existing SDN controllers to evaluate Umbrella's performance and functionality.

1.2.3 OSDF: An Intent Based Software Defined Network Programming Framework

In the third contribution of this dissertation, we propose an intent-based NB API that provides abstractions that hide low level details of the network objects and services, allowing managers to express policies and intents in a descriptive manner instead of a prescriptive manner. The OSDF intent NB API allows network managers to specify network configuration requirements and constraints by reffing to network applications and domains. Furthermore, in this contribution we explore integration of a policy conflict management system with OSDF to resolve network configuration conflicts at the intent level.

This dissertation is organized as follows: Chapter 2 surveys related work and explains the required background for this dissertation. Chapter 3 explains the proposed SDN framework architectures. Chapter 4 presents the implementation details of the proposed frameworks. Chapter 5 discusses the performance evaluation and experimental results of the proposed SDN frameworks. Chapters 6 and 7 discusses open research problems and concludes the dissertation, respectively.

2 BACKGROUND AND LITERATURE SURVEY

2.1 An Introduction to SDN

SDN refers to a network architecture that allows software to control forwarding state in the data plane from a remote location. In other words, SDN decouples the control plane and data plane functionalities from each other and permits external software to program the data plane hardware directly. An SDN controller is the software that defines the control logic and provides the abstractions and facilities that programmers can use to program network devices based on a set of network policies. In the context of SDN, we define a set of forwarding rules for programming of network devices. Forwarding rules can be based on flows instead of destination addresses. That means each network device in a network performs a set of well-defined instruction set (e.g. forward, drop, send to the controller, modify, etc) based on a set of match fields in packets that arrive at a network device. To achieve the goal of separating the SDN control plane and data plane functions from each other, we need to define a set of programming interfaces between the controller and network devices and between the controller and management applications.

This chapter presents a survey of the literature related to SDN, and gives background pertinent to the dissertation. The following sections explain the key components of SDN architectures and survey the related work for each. Section 2.3 explains network event processing in the context of SDN. Section 2.4 briefly explains messaging patterns that applications can use to communicate and exchange information with each other. Finally, Section 2.5 presents three well known software architectures that programmers can use for implementing of software systems.

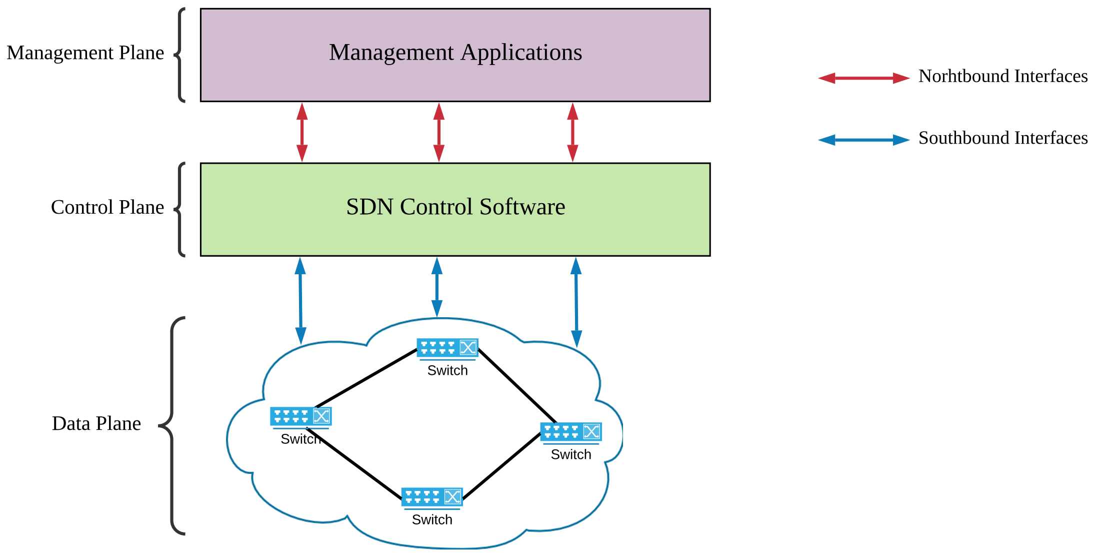

Figure 2.1.: A general SDN architecture

2.2 Literature Survey

This section explains five key building blocks of SDN architectures as Figure 2.1 illustrates, including network infrastructure and *Southbound (SB)* interfaces, SDN controllers, NB interfaces, programming languages, and management applications. In addition, we briefly survey the related work for each of the above building blocks [1].

2.2.1 An Overview of Network Infrastructure and SB Interfaces

SDN networks are similar to the traditional networks in terms of infrastructure. An SDN infrastructure consists of network devices such as routers, switches, and middlebox appliances. The major difference between an SDN infrastructure and a traditional network resides in the fact that network devices become simple forwarding devices and a centralized SDN controller implements the network intelligence. To achieve the goal of separating the control plane and data plane from each other, the SDN controllers need to communicate with the network devices via an API. In this context, SB interfaces define the communication protocols between the controller and network devices. OpenFlow [5] is one of the first SDN standards that enables the implementation of SDN concepts in hardware and software and we explain it as follows:

The OpenFlow is a flow-oriented protocol that defines a SB API that allows a controller to update flow table rules and associated actions that a switch performs for each of the flows that pass through it. The OpenFlow architecture includes three main components as follows:

- **OpenFlow switches**: An SDN controller uses OpenFlow to monitor or update flow table rules in a set of switches. Figure 2.2 illustrates the main components of an OpenFlow switch:

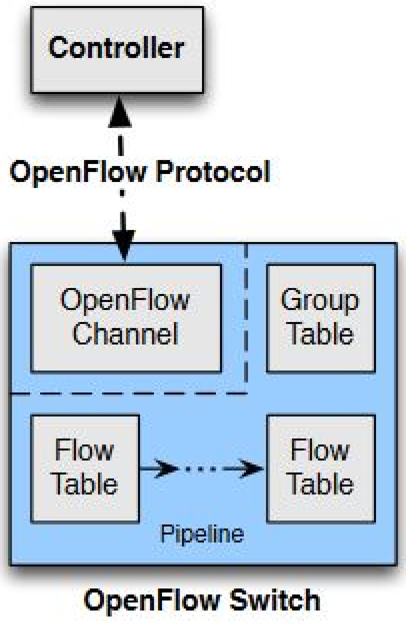

Figure 2.2.: The main components of an OpenFlow switch [6]

  – **Flow table(s) and a group table**: The tables store OpenFlow rules and actions associated with each Openflow rule.

  - **Communication channel**: The controller and switch use a communication channel to transmit commands and messages between themselves.
  - **OpenFlow protocol**: Defines the syntax and semantic of messages passed between a controller and an OpenFlow switch.

  There are two types of OpenFlow compliant switches: *OpenFlow-only*, and *OpenFlow-hybrid* switches. OpenFlow-only switches should process packets using the OpenFlow pipeline that we explain it later.

- **Controllers**: A controller is responsible for updating (i.e. adding, deleting, and revising) flow table rules and the associated actions that a switch performs for each of the flows that pass through it.
- **Flow entries**: Each flow table consists of flow entries. Each flow entry consists of a set of *match fields*, *counters*, a set of *instructions*, *priority*, *timeouts*, and *cookie*.

As Figure 2.3 illustrates, OpenFlow pipeline [6] processes packets through a series of tables, using three main steps for each table:

- Find the highest priority matching OpenFlow rule in the table.
- Apply the associated action to the packet. The actions include modifying a packet, updating match fields, updating the action set, and updating metadata.
- Finally, the switch sends match data and action to the next table in the pipeline.

OpenFlow is not the only SB protocol that researchers propose. Some of the proposed SB protocols are *Open Vswitch Database Management Protocol (OVSDB)* [7], ForCES [8], *Protocol Obvious Forwarding (POF)* [9], OpenState [10], *Revised OpenFlow Library (ROFL)* [11], *Hardware Abstraction Layer (HAL)* [12], and *Programmable Abstraction of Datapath (PAD)* [13].

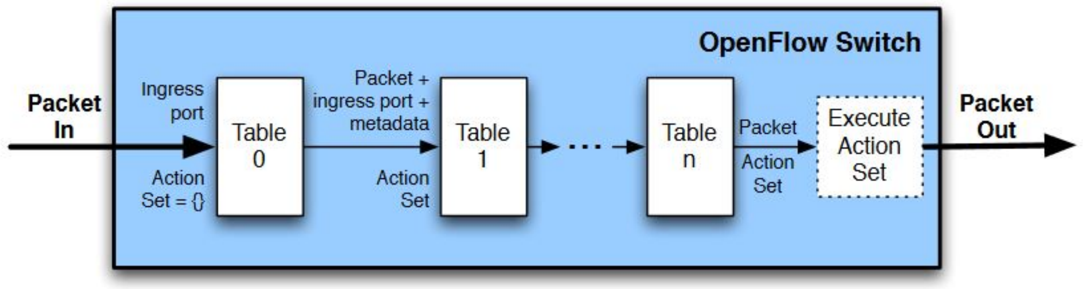

Figure 2.3.: The OpenFlow pipeline [6]

In OpenFlow switches, a manufacturer defines the data-plane functionality. That means programmers cannot program data-plane directly to add or remove new functionalities or upgrade OpenFlow protocol in an OpenFlow switch.

Next generation of SDN will be based on *Programmable Data Planes (PDPs).* In the PDP architecture, the data plane functionality is not fixed in advance but a programmer can define it using software. *Programming Protocol-Independent Packet Processors (P4)* [14] is a high-level programming language for programming data plane network devices. P4 provides flexibility to programmers that allows them to specify how a switch handles packets. Unlike a conventional switch in which the vendor chooses data plane functions that cannot be modified, P4 allows a programmer to specify the data plane functionality. In addition, P4 is protocol independent that does not tie a switch to a set of specific packet formats. P4 is also target independent which means that we can compile and execute P4 programs on different platforms, such as *Field-Programmable Gate Array (FPGA)* [15], general purpose *Central Processing Units (CPUs)*, software switches, etc. In PDP paradigm, the control plane communicates with the data plane using the same channels as in a traditional SDN switches, but the set of tables and other objects in the data plane are no longer fixed and we can define them by a P4 program. The P4 compiler generates P4 Runtime that is a silicon-independent and protocol-independent API from a P4 program. Programmers can use P4 Runtime API for local or remote control plane applications.

Figure 2.4 illustrates the workflow when developers program a programmable switch using a P4 program.

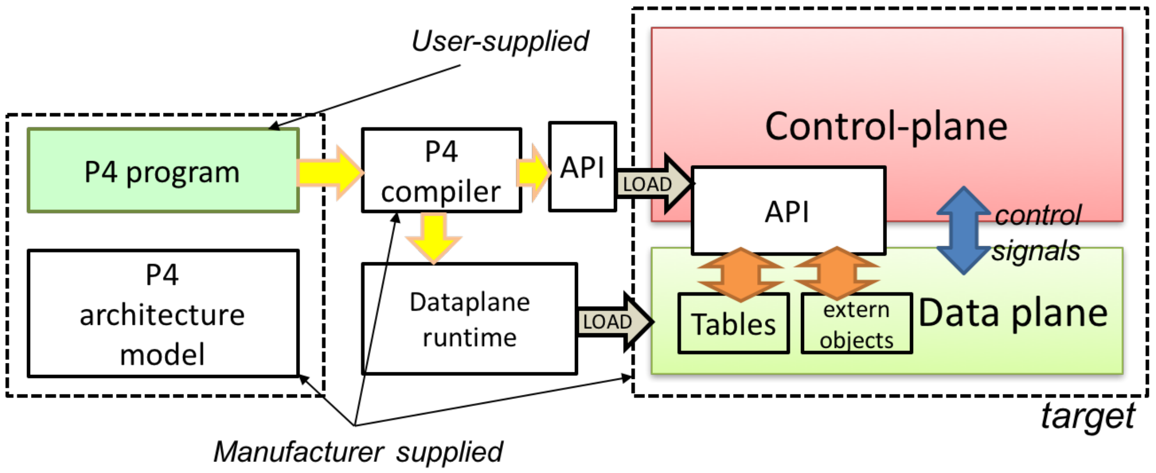

Figure 2.4.: Programming a target using a P4 program [16]

SB interfaces have not been limited to the APIs that programmers can use for defining of control plane functionalities. Researchers and engineers proposed APIs that programmers can use to install, modify, delete, and retrieve configuration from a network device. For example, *Internet Engineering Task Force (IETF)*, an open international community of network designers, operators, and vendors proposed *Network Configuration Protocol (NETCONF)* [17] that is a network configuration protocol that provides mechanisms to install, modify, retrieve, and delete the configuration of network devices. NETCONF uses an XML-based data encoding for the configuration data as well as the protocol messages. The NETCONF protocol operations can be implemented on top of a simple *Remote Procedure Call (RPC)* layer. RPC is a technique that allows a program on a local computer to execute a program on a remote computer. *gRPC Network Management Interface (gNMI)* [18, 19] is a new network management interface protocol that is proposed by *OpenConfig* which is an informal working group of network operators. gNMI is a gRPC based protocol for the modification and retrieval of configuration from a network device. Developers and network administrators can also use gNMI to control generation of telemetry streams from a network device to a data collection system. OpenConfig working group also defines *gRPC Network Operations Interface (gNOI)* [18] that is a protocol for executing operational commands such as reboot, ping, traceroute, etc on network devices.

2.2.2 An Overview of SDN Controllers

An SDN controller is one of the key components of SDN architectures. An SDN controller provides programming abstractions and services such as topology discovery, flow rule service, and event notification service that programmers and network administrators can use to program SDN network devices based on a set of high-level network policies [1]. We can categorize the SDN controllers into two major categories: *centralized controllers* and *distributed controllers* [20]. A centralized SDN controller is a single entity that manages all of the network devices in a network. A centralized approach in designing of SDN controllers is good enough for small size networks but that is not a salable solution for a network with a large number of network devices. Furthermore, a centralized SDN controller potentially is a single point of failure and can become a bottleneck while dealing with an increasing number of requests. Beacon [21], Ryu [22], NOX-MT [23], and Floodlight [24] are some of the centralized SDN controllers. To address the problems of the centralized approach in designing of SDN controllers, researchers and engineers proposed distributed control plane architectures. A distributed control plane can be a centralized cluster of nodes or can be distributed physically across set of elements. We can categorize distributed control plane architectures into two major categories:

- *Distributed flat control plan*: In the flat control plane architecture, we partition a network into multiple regions and each instance of the controller manages the switches in one region. Onix [25], ONOS [4], HyperFlow [26], DISCO [27] are examples of flat control plane architecture.
- *Distributed hierarchical control plane* partitions the control plane into multiple levels depending on the required services. Kandoo [28] is an example of dis-

tributed hierarchical control plane that divides the control plane into two parts to distinguish local control applications that process events locally from non-local applications that require access to the network-wide state. Furthermore, B4 [29], and Espresso [30] are two other examples of distributed hierarchical control plane architecture.

2.2.3 An Overview of NB Interfaces and Programming Languages

The SDN controllers provide NB interfaces that define the communication between SDN management applications and the controller. SDN management applications use NB APIs to communicate with the controller, express network behaviors, define configuration requirements, and program forwarding devices. We can categorize NB APIs into two major categories [20] as follows:

- *Low-level Internal APIs*: These APIs refer to a set of low-level APIs that each of the SDN controllers provide and programmers can use them for implementing of SDN management applications within the controller. Using these APIs to develop management applications tightly couple the applications with a specific SDN controller and the programming language that the controller uses to implement the control plane services.
- *High-Level External APIs*: These APIs refer to the high-level APIs that SDN controllers provide via web based interfaces such as REST API or high-level programming languages that programmers can use to implement external management applications (i.e. outside of the SDN controller) using an arbitrary programming language.

In [31], authors define seven major categories used to classify the paradigms that SDN programming languages follow: *imperative*, *declarative*, *logic*, *event-driven*, *functional*, *Functional Reactive Programming (FRP)*, and *reactive*. The following provides a brief explanation of each:

- An imperative programming model refers to traditional computer programming model using well known programming languages such as C++, JAVA, Python.
- In declarative approach, a programmer specifies what a program must do without specifying the details of how it should be done.
- Logic programming requires a programmer to specify a set of logic rules that the programming system transforms into complex symbolic structures.
- An event-driven programming approach allows programmers to write pieces of code that each to react to an event.
- The functional programming model performs computation by evaluating a set of mathematical functions. It avoids state and mutable data.
- FRP is a model that allows a program to manipulate streams of data values without requiring a programmer to write event-driven programs.
- A reactive programming approach is a programming paradigm that programmers can use to write a program as set of reactions to external events and conditions without worrying about the order of events.

In the last few years, researchers proposed various high-level programming languages for writing SDN management applications. The following reviews major SDN programming systems:

- *Frenetic*: In [32], the authors present a high-level programming language; *Frenetic*; that programmers can use to program OpenFlow networks. It consists of two sub-languages:
  - *Network query language*: this language provides a collection of orthogonal, high-level query operators that read the state of the network but do not modify it.

  - *Network policy management library*: uses a combinator library for FRP to govern the forwarding of packets through the network. In other words, a program "sees every packet" instead of processing traffic indirectly by manipulating OpenFlow rules.

  The Frenetic language is a set of Python libraries with a design and implementation based on FRP. One of the major components of the *Frenetic* is a runtime system that sits between the high-level program and the SDN controller. The runtime system is responsible for managing, installing, and uninstalling OpenFlow rules on the network switches. In addition, the runtime system generates the required communication patterns between the network switches and the controller.

- *Pyretic* [33] is an open-source Python based platform that provides a set of high-level abstractions that allow programmers to write SDN applications and specify network policies using a high-level language instead of using low-level OpenFlow mechanisms. One of the main features of Pyretic is that a programmer can specify a policy for the entire network and does not need to worry about the configuration of each switch separately. Furthermore, Pyretic uses composition operators, such as parallel and sequential composition, to help programmers to develop modular applications by combining multiple policy. In addition, Pyretic can create dynamic policies whose behavior will change over time. Furthermore, Pyretic provides a topology abstraction facility that helps programmers to take an abstract view of the underlying network when writing an application, and to compose new Pyretic programs from existing abstractions.

- *Merlin*: In [34], the authors present a policy based network programming language, *Merlin*, a declarative language based on regular expressions. We can use Merlin to express network requirements using high-level network policies. Although Merlin simplifies network programming, a programmer still needs to use a predicate language to specify a set of header fields used to classify network traffic.

- *Procera*: In [35], the authors present *Procera*, a controller architecture and a high-level network language to support reactive network control policies. The design of Procera is around FRP to express network policies that characterize reactive and temporal network behaviors. Procera uses a set of signal functions and abstract data types to express network policies using windowed event histories.

- *Kinetic*: Kinetic [36] is a domain-specific language that programmers can use to express network policies in terms of *Finite State Machines (FSMs)* that is a mathematical model of computation to simulate sequential logic. In addition, it provides the capability to write network applications that capture responses to dynamic network conditions. Each of the states in a FSM denotes a distinct forwarding behavior and network events trigger the transition between the states.

- *Maple*: Maple [37], is an SDN programming framework that provides a standard programming language that programmers can use to express network behaviors using a centralized algorithm. In the framework, a programmer defines a function f to route each packet, and the framework derives the corresponding flow rules automatically. In addition, the paper introduces two new components, including an SDN optimizer that is responsible for deriving reusable forwarding decisions, and a runtime scheduler that acts as an optimizer by providing a significant horizontal scalability across multiple cores.

- *Nettle*: Nettle [38] is a domain specific language based on FRP that allows programmers to configure a network using a declarative style. One of the key contributions of the paper is a discrete event-based abstraction that allows the system to develop controllers that treat each message stream as a whole rather than as individual messages. Furthermore, the paper describes an abstraction that captures continuous, time-varying quantities such as traffic volume on individual links. The abstraction gives programmers flexibility to write traffic engineering applications.

- *PonderFlow* [39] defines a policy specification language for OpenFlow networks. PonderFlow is an object-oriented declarative language that extends the *Ponder* policy specification language [40]. One of the main goals of the project involves providing a policy language to decouple it from conventional programming languages. In other words, an administrator can configure a network without any knowledge of conventional programming languages such as Python and Java.

- *ONOS Intent Framework [2]*: The ONOS SDN controller provides a high-level user interface that allows programmers to specify network configuration requirements in the form of a policy rather than a mechanism. An ONOS intent is an immutable model object that specifies the following key items:

  – *Network Resource*: A set of object models, such as links, that define part of the networks that an intent can affect it.

  – *Constraints*: Defines a set of weights that users can apply to network resources, such as bandwidth, optical frequency, and link type.

  – *Criteria*: Packet header fields or patterns that describe a type of traffic.

  – *Instructions* : Actions to be applied to a type of traffic, such as packet header field modifications or forwarding actions (e.g., forward to a specific outgoing port.

2.2.4 An Overview of SDN Management Applications

By emergence of SDN, programmers can write management applications to implement required network control logic and strategies such as routing, network monitoring, firewall, *Network Address Translation (NAT)*, load balancing, etc. In [1], the authors categorize SDN management applications into five major categories including traffic engineering, wireless and mobility, measurement and monitoring, security and dependability, and data center networking applications. We briefly explain each of the above categories as follows:

- *Traffic Engineering Applications*: Traffic engineering is one of the well known type of SDN applications that allows us to monitor network traffic, monitor changes, and decide how to route network traffic to improve utilization of network resources and performance [41]. ElasticTree [42], Hedra [43], MicroTe [44], PolicyCop [45], QNOX [46], ALTO [47], FlowQoS [48] are examples of the traffic engineering applications.

- *Wireless and Mobility Applications*: SDN provides the opportunity to implement required control plane features in different types of wireless networks (e.g. *Wireless Local Area Networks (WLANs)*, cellular networks) such as managing the limited spectrum and interference, allocating radio resources, implementing handover mechanisms, and performing load-balancing between cells in an easier and more efficient manner. For example, the authors in [49], proposed Odin, an SDN-based solution that provides programming abstractions to support features that enterprise and provider networks need to make Wi-Fi networks programmable. As another example, authors in [50] presents OpenRadio a programmable wireless data plane that provides modular and declarative programming interfaces that programmers can use to program the physical and *Media Access Control (MAC)* layers.

- *Measurement and Monitoring Applications*: The authors in [1], categorize measurement and monitoring applications into two major categories: 1) The first category of measurement and monitoring applications provides services to other networking services. For example, a programmer can implement a topology monitoring application that informs other services and applications about the topology changes. 2) The second category of applications uses sampling and estimation techniques to reduce the burden of the control plane due to the collection of data plane statistics. OpenTM [51], PayLess [52], OpenSample [53] are examples of measurement and monitoring applications in the context of SDN.

- *Security and Dependability Applications*: Researchers address the security research problems in SDN from two different perspectives: 1) using SDN to improve security of computer networks and 2) improve the security of SDN itself. Programmers use SDN to implement applications to improve security of current networks such as firewalls, traffic anomaly detection, *Denial of Service (DoS)* attack (a cyber attack which makes computers and network services inaccessible to end users) detection, etc. For example, the authors in [54] show how programmers can use SDN and OpenFlow networks to detect anomalies inside home and home office networks. Furthermore, the authors in [55] presents a lightweight method to detect *Distributed Denial of Service (DDoS)* attacks based on traffic flow features using OpenFlow networks.

- *Data Center Networking Applications*: We can benefit from SDN to solve some of the data center networking problems such as live network migration [56], troubleshooting, network management, optimizing of data center resources, etc. For example, *FlowDiff* [57] is a framework for detecting operational problems in data centers such as host and network failures, and unauthorized access.

There are two major paradigms for implementing of SDN applications: *Reactive* and *Proactive*. In proactive paradigm, a management application instructs a controller to install flow rules in each network device before network traffic arrives. In other words, in the proactive approach, we program network devices for all possible traffic in advance and flow table does not change while the network is running. On the other hand, the reactive paradigm allows a management application to program network devices dynamically. In the reactive paradigm, a controller installs default flow rules on network devices to send their traffic to the controller whenever there is no forwarding rule to handle the incoming traffic. When the controller receives a packet, it installs an appropriate forwarding rule in a network device, and then returns the packet to the device to forward it to the next hop. The reactive paradigm gives more flexibility to the programmers to implement management applications that can react to the network changes dynamically. However, it introduces an additional delay for processing of the first packet before installing forwarding rules on the network devices and returning back the packet to the pipeline. Furthermore, in the reactive paradigm, the controller can become a bottleneck specially if we send a huge amount of traffic between two end-points. On the other hand, proactive paradigm does not give programmers so much flexibility in designing and implementing of dynamic management applications. Furthermore, proactive paradigm is a better approach for programming of network devices if network administrators know about all of the incoming traffic to a network in advance. In addition, proactive approach does not introduce an additional setup time even for the first incoming packet.

2.3 An Overview of Event Processing in SDN

In this context, we define an event as a change in a network. We briefly explain some of the most important events in a network as follows:

- *Link event*: This event refers to a change in one of the links in the network topology. For example, when a new link appears (i.e. a LINK_ADDED event), a link disappears (i.e. a LINK_REMOVED event), or there is a change in the link characteristics (i.e. a LINK_UPDATE event) in the network topology, then a link event occurs.
- *Device event*: Device event refers to a change in one of the network devices in the network topology. For example, when we add or remove a device from the network topology (i.e. a DEVICE_ADDED or a DEVICE_REMOVED event), or when there is change in one of the ports of a network device (i.e.a

PORT_ADDED, PORT_REMOVED, or a PORT_UPDATED) then a device event occurs.

- *Packet event*: Packet event occurs when a switch sends an incoming packet to the controller if there is no other forwarding rule for the packet (i.e. a PACKET_IN event). In addition, a packet event occurs when the controller should return an incoming packet to the pipeline (i.e. a PACKET_OUT event).
- *Host Event*: Host event refers to a change in one of the hosts in the network topology. When we remove or add a host (i.e. a HOST_ADDED event, a a HOST_REMOVED event), or when a host moves (i.e. a HOST_MOVED event),
- *Flow Rule Event*: Flow rule event occurs when there is a change in a flow rule in the system. For example when we remove, add, or update a flow rule (i.e. a FLOW_RULE_ADDED event, a FLOW_RULE_REMOVED event, or a FLOW_RULE_UPDATED event) a flow rule event occurs.

Providing and processing of events are the key tasks that SDN controllers perform to implement required core control plane services. As an example, we explain how ONOS SDN controller provides network events to the core subsystems and application in the following.

ONOS interacts with the underlying network devices with help of its providers as Figure 2.5 illustrates to achieve the following design goals:

- Execute the requests that a provider receives from one of the core subsystems such as add, remove, or update flow rules, collect network statistics (e.g. number of flow rules in the system), set or get *Management Information Base (MIB)* parameters in *Simple Network Management Protocol (SNMP)*, an Internet protocol that can be used to collect and organize information about managed network devices, set or get port status in a NETCONF [17] device, etc.

- Providers process and notify core subsystems about the events that they receive from the underlying network devices such as packet, link, device, flow rule related events.

As the Figure 2.5 illustrates, the core subsystems send their requests to the providers and they communicate with SB protocols to receive and then provide the requested information to the core subsystems. Furthermore, the core subsystems provide APIs to the applications that they use to receive events such as packet, link, device, and flow rule events. In addition, applications use NB core APIs to install, remove, and update flow rules by sending their requests to the core subsystems.

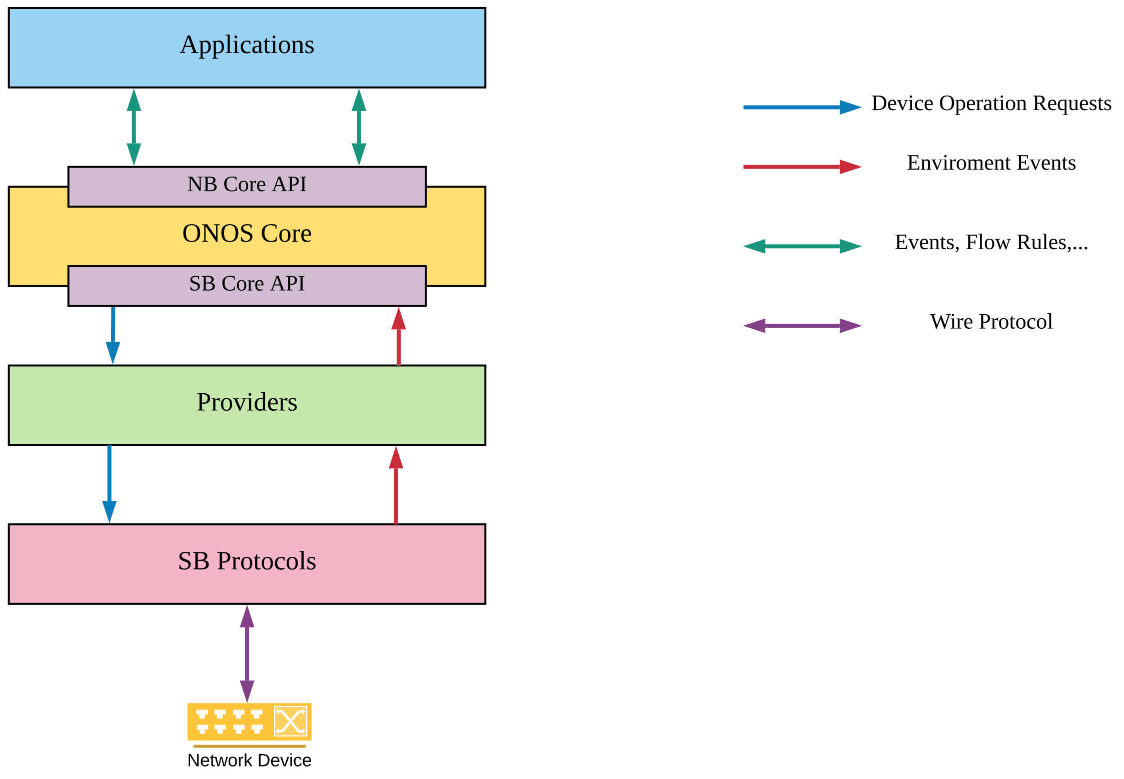

Figure 2.5.: ONOS architecture

2.4 An Overview of Messaging Patterns

A *messaging pattern* is a network-oriented architecture that specifies how two end-points (e.g. two applications or two subsystems of one application) of a messaging system connect and communicate with each other [58]. As Figure 2.6 illustrates, an application communicates with another application using a message channel where one application (i.e. the sender application) writes information to the channel and the other application (i.e. the receiver application) reads that information from the channel [59].

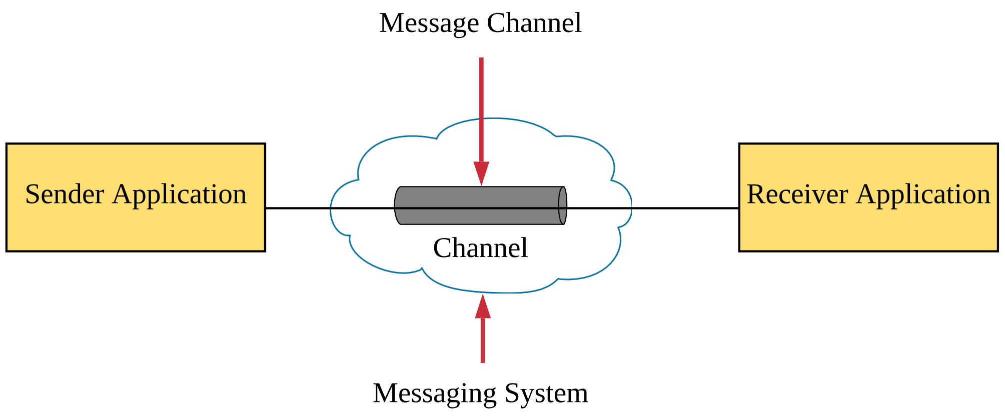

Figure 2.6.: The architecture of a general messaging system

We can categorize message channels into nine categories including *point-to-point*, *publish-subscribe*, *data type*, *invalid message*, *dead letter*, *guaranteed delivery*, *channel adapter*, *messaging bridge*, and *message bus* channels [59]. For the use of this dissertation, we explain point-to-point and publish-subscribe channels as follows:

- *Point-to-point channel*: As Figure 2.7 illustrates, if a channel has multiple receivers, a point-to-point channel guarantees that only one receiver consumes a particular message. An application can use a point-to-point channel to make an RPC or transfer messages.

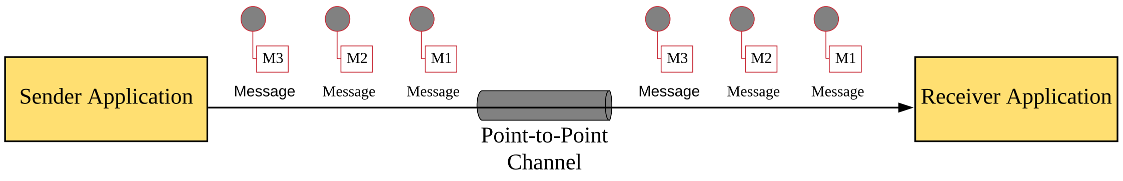

Figure 2.7.: Point-to-point message channel

- *Publish-subscribe channel*: In this model as Figure 2.8 illustrates, a publisher publishes its messages (e.g. system events) on a publish-subscribe channel and a set of receivers subscribe to receive a specific type of message on separate output channels (i.e. the messaging system assigns a separate channel for each subscriber).

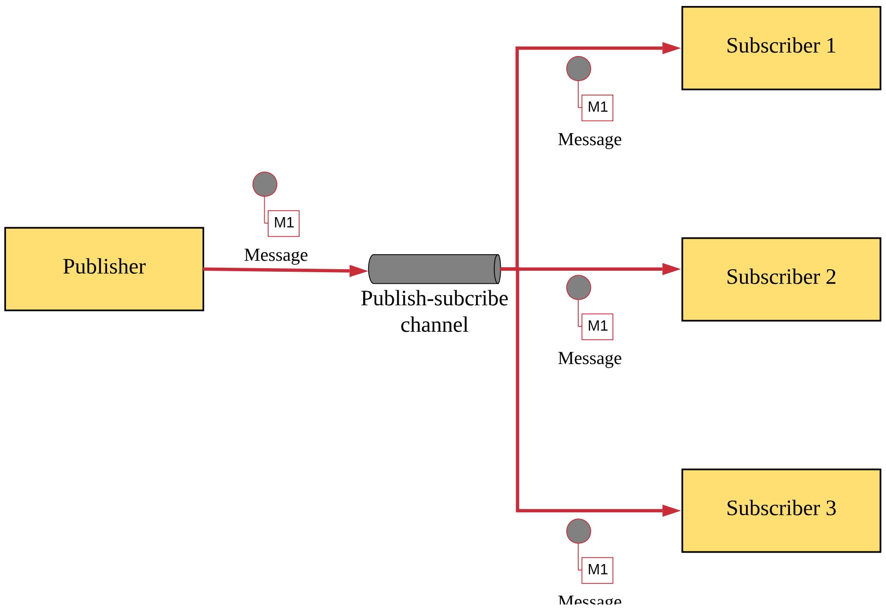

Figure 2.8.: Publish-subscribe message channel

2.5 An Overview of Software Architectures

We explain three software architectures that programmers can use to implement software systems including monolithic, microservice, and *Service Oriented Architecture (SOA)* architectures in the following subsections.

2.5.1 Monolithic Architecture

Monolithic architecture is a traditional architectural style for building of software systems that all of the software components combined and tightly coupled into one single program (i.e. composed all in one piece). Easy to develop, test, and deploy are the main advantages of monolithic architecture. Furthermore, developers can scale a monolithic software horizontally by running multiple copies behind a load balancer. However, it has some disadvantages as well. For example, whenever there is a change in one of the components of the software, programmers need to build the whole program again that makes updating of the software fast and correctly challenging. In addition, a bug in of the components can potentially affect other components and bring down the entire software. The last but not the least, If developers want to scale certain components of software, they must scale the entire software.

2.5.2 Microservice Architecture

The evolution of technologies such as containers (e.g. Docker [60]), cloud services, and container orchestrators such as Kubernetes (k8s) [61] enable the ability to develop distributed, more scalable, and reliable software systems. Microservice architecture is an application architecture that allows programmers to build an application as a suite of cooperative services (i.e. loosely coupled components). Each microservice supports a specific business goal and uses a simple, well-defined interface to communicate with other microservices. In the microservice architecture, developers can run each microservice in a container and use container orchestrators to orchestrate microservices. Some of the main advantages of the microservice architecture are as follows:

- It allows multiple working groups work in parallel and independent of each other on development of microservices.
- It allows programmers to build and deploy each microservice without requiring to build all of the components.
- Gives the ability to scale a given microservice horizontally, independent of other microservices.
- In the microservice architecture, If one of the microservices fails, it will not affect other microservices. Moreover, a microservice can be repaired and restarted without recompiling the entire application and without restarting other microservices.
- In the monolithic architecture, a single, large code base includes all components. Consequently, changes to even a small seldom-used component requires changing the code base. In the microservice architecture, each component can be in a separate codebase, isolating changes.

2.5.3 Service Oriented Architecture (SOA)

A SOA is a software architecture design style that application components provide services to other components via a communications protocol over a network. SOA and microservice architectures at some aspects are very similar that we explain them briefly as follows:

- Both architectures are a collection of services that each focuses on a specific set of business goals and are much smaller in scope than an entire monolithic software system.

- In both architectures, programmers are free to choose their programming language and technologies to implement a service that brings technology diversity into a development team.

- Both architectures are proposed to solve the issues associated with the monolithic architecture.

However, SOA and microservice architectures are similar at some sense but they are different in many aspects that we explain some of them briefly as follows:

- In microservice architecture each service focuses on implementing of a function when compared with SOA that each service is composed of multiple functions with many interdependencies. In other words, services in the microservice architecture are much smaller in terms of size and scope when compared with services in SOA.

- Microservice architecture and SOA are very different in terms of communication mechanism between services. In the microservice architecture, microservices communicate via language agnostic protocols over network. On the other hand, in SOA services communicate via *Enterprise Service Bus (ESB)*. ESB is an architecture that defines a set of rules and principles to integrate a set of services and applications together over a bus like infrastructure. Each of the mentioned communication mechanisms has advantages and disadvantages. For example, in SOA, ESB can be a single point of failure that potentially can impact the entire system. Microservices are much better in providing fault resiliency, however using language agnostic protocols over network increases the number of remote function calls and communication overhead.

- Another major difference between the microservice architecture and SOA is the way they use data storage. In SOA the services share data storage, while in the microservice architecture, each microservice can have independent data storage.

Figure 2.9 illustrates an example of the monolithic architecture vs. the microservice architecture.

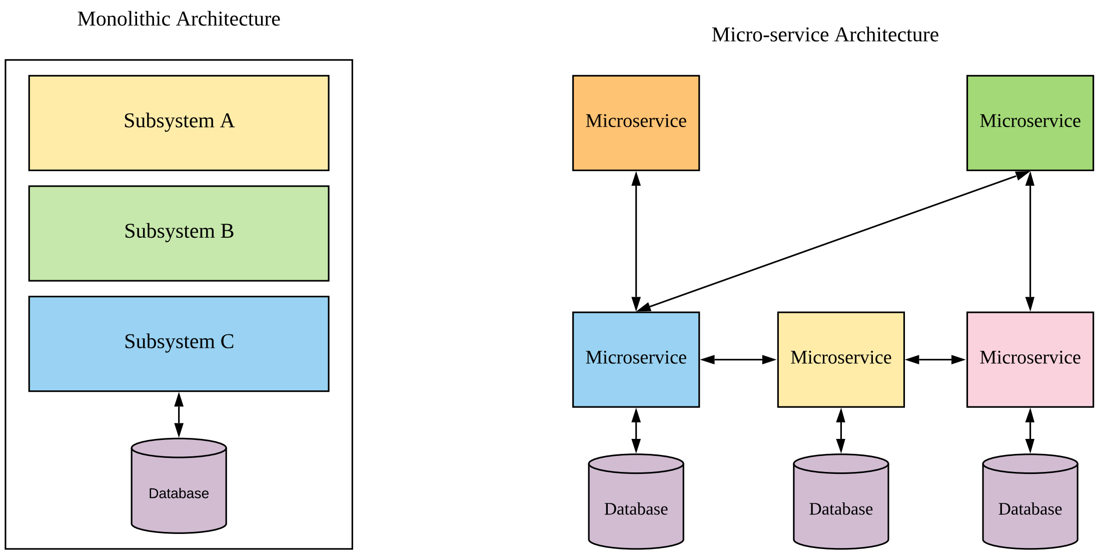

Figure 2.9.: Monolithic architecture vs. microservice architecture

3 AN OVERVIEW OF THE SDN FRAMEWORK ARCHITECTURES PROPOSED IN THIS DISSERTATION

3.1 SDN Control Plane Disaggeragtion

This section explains the concept of SDN control plane disaggregation and introduce a distributed architecture for next generation of SDN controllers. In addition, it explains the steps that we need to take towards disaggregating the SDN control plane and potential ways that developers can use to achieve the goal.

3.1.1 An Introduction to the Control Plane Disaggeragtion

To migrate from a monolithic approach to a microservice architecture in designing of the SDN controllers, we need to disaggregate the control plane services into a set of cooperative microservices that can communicate with each other via standard APIs. In the disaggregated model, a controller core provides the minimum required functionality, and microservices that exist outside of the controller core provide all other services as Figure 3.1 illustrates. In the disaggregated model, developers can run each microservice in a container and orchestrate all of the microservices via container orchestrator technologies. We explain some of the main advantages of control plane disaggregtation as follows:

- *Flexibility to scale*: One of the main advantages of the control plane disaggregation arises from its ability to scale a given core service horizontally, independent of other subsystems and services.
- *Freedom in choosing the programming language*: The control plane disaggregtation allows programmers to choose an arbitrary programming language, programming technology, and third-party libraries when building an SDN man-

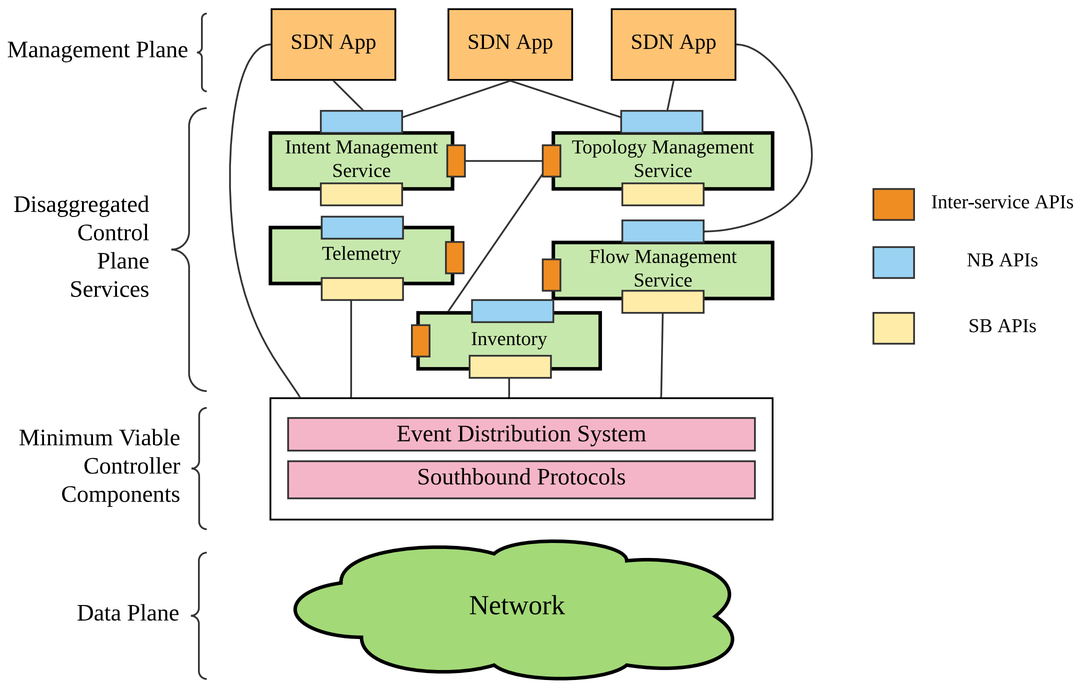
Management Plane
SDN App
SDN App
SDN App
Disaggregated Control Plane Services
Intent Management Service
Topology Management Service
Telemetry
Flow Management Service
Inventory
Inter-service APIs
NB APIs
SB APIs
Minimum Viable Controller Components
Event Distribution System
Southbound Protocols
Data Plane
Network

Figure 3.1.: A disaggregated SDN control plane architecture

agement application. The approach makes the application development process more flexible, and allows programmers to choose a programming language that is appropriate to a given application.

- *Fault isolation*: In current SDN controllers, if one of the subsystems fails it can affect the entire controller. A disaggregated control plane means that the failure of a given microservice will not affect other microservices. Moreover, developers can repair or restart a microservice without requiring to recompile the controller or restarting other microservices.

- *A controller core with minimal components*: In the monolithic approach, every instance of a controller includes all services and applications, even if the instance only needs a small subset. In the disaggregated model, the controller core contains the minimum viable set of components and functions, and only the required applications and services need to be deployed outside of the controller core.

- *A Disaggregated Code Base*: In the monolithic architecture, a single, large code base includes all services and applications. Consequently, changes to even a small seldom-used service requires changing the controller code base. In the disaggregated model, each service and each application can be in a separate code base that allows programmers to isolate changes.

The key components of a disaggregated control plane architecture consist of:

- **Minimum Viable Controller Components**: The first step towards SDN control plane disaggregation involves identifying a minimal set of viable controller core components. As we point out earlier in the previous chapter, control plane core services communicate with the underlying network devices with help of providers that communicate with SB protocols to execute requests that they receive from core subsystems and provide them the requested information. When it receives a request from a management app or from a control plane service, a control plane core service uses a SB protocol to communicate with the underlying network devices, and then uses the information to respond to the request. Consequently, a controller with minimum functionality needs the following two major subsystems:

  – **An Event Distribution System**: This subsystem notifies microservices (i.e. control plane services and management applications) about events that occur in the underlying network devices, such as flow rule, network link, device, or packet exception events. The event distribution system provides a standard API used to stream events to the external processes (i.e. microservices); to permit multilingual apps, the API must be independent of the programming language used to implement each microservice.

  – **SB Protocols**: SB protocols define the communication between underlying network devices and the core of the controller. The event distribution system communicates uses SB protocols to receive events from underlying devices, and then uses the event distribution mechanism to propagate event notifications to external microservices.

- **Disaggregated Control Plane Services**: In the disaggregated model, each of the core control plane services (e.g. topology service, flow service, etc) is a microservice that can run in a container. The microservices (i.e. the containers in which the service runs) can be orchestrated using a conventional container orchestrator technology, such as *Kubernetes*. Each control plane service provides three sets of APIs: NB APIs that define the communication between control plane services and applications, inter-service APIs that define the communication between each microservice, and SB APIs that define the communication between each microservice and controller core components.
- **Management Plane**: Management plane comprises a set of management applications that use disaggregated control plane services to implement network control functions, policies, and strategies such as routing, network monitoring, firewall rules, NAT, and load balancing. To access microservices, management applications use the NB APIs that the microservices offer. In fact, each management application forms a microservice that runs in a container exactly like other microservices.

3.1.2 Externalization of Event Processing in SDN

As the previous section explains, disaggregating control plane requires external event processing. This section provides some examples of using of externalization of event processing for implementing of control plane services and management applications as microservices outside of the SDN controller as follows:

- *Topology Discovery Service*: a typical SDN controller provides several built-in services, such as a topology discovery. Developers can implement some of the core services of an SDN controller as microservices outside of the controller, if we can externalize packet processing. *Link Layer Discovery Protocol (LLDP)* and *Address Resolution Protocol (ARP)* are two well known Internet protocol that can be used for discovering of links and hosts in a network topology. For example, to implement a topology discovery microservice, programmers need an event distribution mechanism to capture LLDP and ARP packets and provide them to an external microservice to derive a mapping of the links between switches and end-hosts. We can also notify topology service about the topology changes by capturing of topology related events such as link and device events. Consequently, topology service can receive the changes via the event distribution system to update the topology accordingly.

- *External Reactive Management Applications*: Externalization of event processing and specifically packet processing, allows us to process incoming packets in an external packet processor that underlying network devices send to the controller whenever there is no forwarding rule to handle the incoming packets. For example, suppose an external reactive forwarding application that receives an incoming packet, extract required match fields, generate flow rules, install the rules on appropriate network devices, and then return the incoming packet to the network.

3.2 An Overview of the Proposed Event Distribution Mechanisms

This section reviews two approaches that can be used to implement an event distribution system that will externalize event processing in an SDN controller core [62].

3.2.1 An Event Distribution Mechanism Using the Publish-Subscribe Model

The first event distribution approach employs a publish-subscribe model. As Figure 3.2 illustrates, an event distribution system that runs inside the controller listens for all events occurring in the controller core, and pushes the events into one of the topics that an event broker offers. The broker forwards the event, and external processes and management applications consume events by subscribing to a topic and pulling all events published to the topic. In other words, the event distribution system acts as a producer by pushing data to brokers, and external applications and services act as consumers by pulling events from the broker. In this model, the broker provides as set of *topics* used to separate events according to their type (i.e. each topic corresponds to a specific type of event). For example, if an external application or a service needs to receive *packet* events, it subscribes to the *packet event* topic by sending a subscription request to the event distribution system. Once the subscription has been registered, the event distribution system starts pushing the packet events into the packet topic.

3.2.2 An Event Distribution Mechanism Using the Point-to-Point Model

The second event distribution system approach employs a point-to-point model. As Figure 3.3 illustrates, the event distribution system uses a point-to-point channel to send event notifications to each of the applications and services. In this model, each application or service subscribes to a specific type of event and the event distribution system streams events to each of the subscribers.

In the following sections, we propose two SDN programming framework to address some of the weaknesses that Chapter 1 lists and explain how the event distribution system can be used to implement each of them.

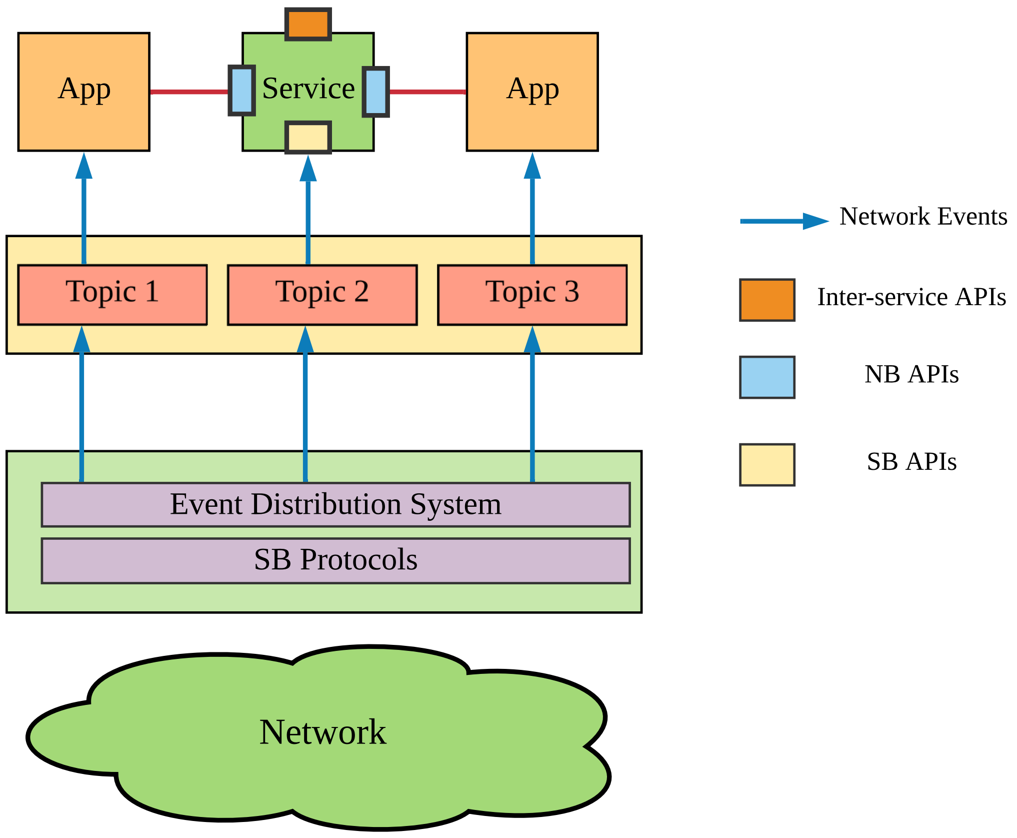

Figure 3.2.: An illustration of an event distribution mechanism using the publish-subscribe messaging model

3.3 Umbrella: A Unified Software Defined Network Programming Framework

As we mentioned in Chapter 1, the NB APIs that current SDN controllers offer for programming external applications differ in terms of syntax, naming conventions, data resources, and usage. Therefore, even porting basic external SDN management applications from one controller to another requires recoding modules, such as those that collect topology information, generate and install flow rules, and compiler flow rule statistics.

We take a new approach to provide a standardized programming interface by first creating new management abstractions and then providing a way to map the abstractions onto heterogeneous NB APIs. We call our unified development framework

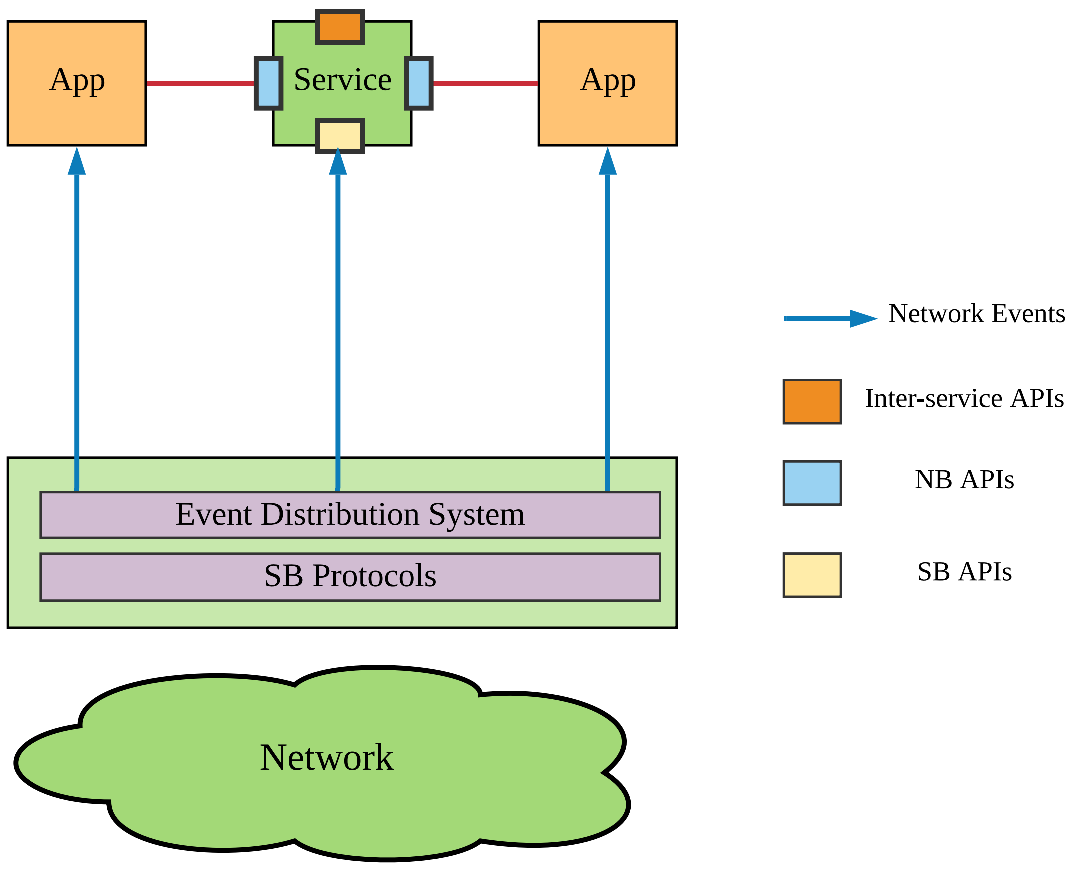

Figure 3.3.: An illustration of an event distribution mechanism using the point-to-point messaging model

Umbrella [63]. We use an architecture based on the approach that operating systems follow when they define high-level I/O abstractions and install a set of device drivers to map the abstractions into hardware commands suitable for a specific commercial hardware device. Our architecture takes an analogous approach by dividing the development framework into two conceptual parts: a module that provides a high-level, controller-independent NB API abstractions, and a set of controller-specific translation modules that map the abstractions into NB API requests and commands that are suitable for various SDN controllers. The main design goals are:

- Design and implement a development framework that provides a new set of abstractions for external SDN management applications that remain independent of the NB APIs that specific SDN controllers provide.
- Design and implement a set of modules that use the proposed abstractions to provide information needed by SDN applications, such as topology, network statistics, and real time topology changes.
- Increase the portability of SDN applications across SDN controllers, and make it easy for a programmer to evaluate a specific application on multiple SDN controllers from various perspectives, including performance.
- Provide a SDN programming framework that reduces programming complexity, allows a programmer to write SDN applications without requiring a programmer to master low-level details for each SDN controller, and avoids locking an application to a specific controller.
- Provide a framework using a hybrid approach that allows a programmer to choose between reactive and proactive paradigms, thereby offering better scalability than a completely reactive or proactive network management system.

Figure 3.4 illustrates the architecture of the Umbrella framework, which includes the following key components:

- *Controller Independent Services*: Umbrella provides a set of controller independent services that programmers use to write external SDN management applications. The services provide a set of high-level and generic APIs that allow applications to install and remove flow rules, retrieve topology information, monitor topology changes, retrieve network statistics and a list of installed flow rules, compute end-to-end paths between network endpoints, and employ custom path finding algorithms.
- *Controller Specific Drivers*: A set of drivers translate between Umbrella's high-level APIs and the NB API of specific controllers; when an application expresses

a request or command, a driver translates into the controller-specific equivalent, and sends the result to the controller to be executed.

- *Applications*: Programmers use Umbrella services and its APIs to write portable and controller-independent SDN applications. If new controllers appear or if the NB API used by a controller changes, a programmer can write a new driver module for the controller or modify an existing driver.

Umbrella employs the event distribution system explained in Section 3.2 to allow a programmer to use the framework for implementing both reactive and proactive SDN applications. Programmers can also use the event distribution system to implement new controller-independent services, such as a general event monitoring service.

3.4 OSDF: An Intent Based Software Defined Network Programming Framework

Although, there is no standard NB APIs for SDN controllers [64], adopting an *intent-based* approach to the design of SDN interfaces seems promising. An intent NB API is independent of a specific network technology, and allows managers to express constraints in a vocabulary and information related to applications. An intent NB API provides abstractions that hide low level details of the network objects and services, and programmers can use them to express their intents in a descriptive manner instead of a prescriptive manner. This section attempts to answer some of the questions surrounding the intent-based approach, including:

- How can we integrate a policy conflict management module with an intent-based SDN programming framework to resolve network configuration conflicts at the intent level?

- How can we incorporate the flexibility of a reactive approach into an intent-based SDN programming framework?

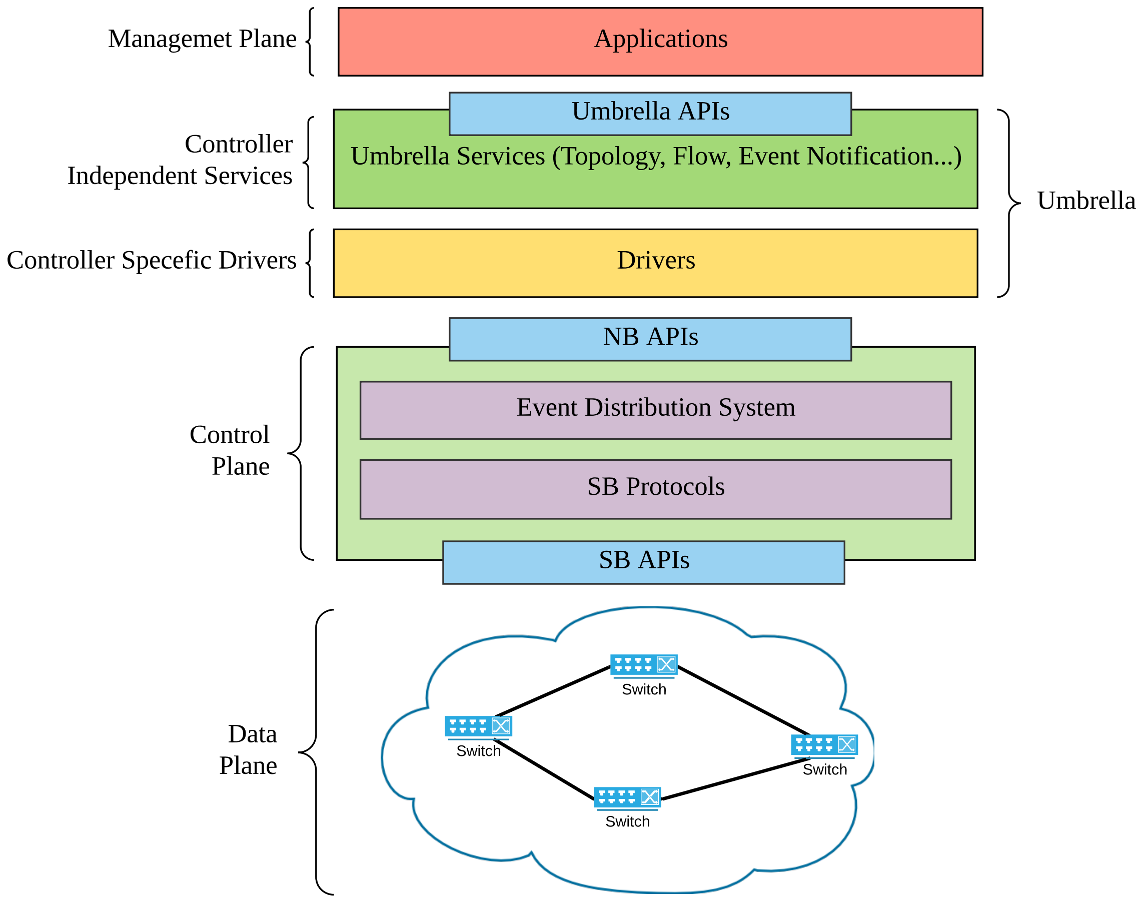
Managemet Plane
Applications
Umbrella APIs
Controller Independent Services
Umbrella Services (Topology, Flow, Event Notification...)
Umbrella
Controller Specefic Drivers
Drivers
NB APIs
Control Plane
Event Distribution System
SB Protocols
SB APIs
Data Plane
Switch
Switch
Switch
Switch

Figure 3.4.: The Umbrella architecture

- How can we design and implement an intent-based specification that allows managers to to express network requirements for application and policies for multiple domains?

To answer the above questions and address some of the weaknesses that Chapter 1 discusses, we propose a policy-based network programming framework called *Open Software Defined Framework (OSDF)* [65, 66]. The OSDF framework achieves the following design goals:

- Provide a high-level API that allows managers to express network configuration requirements using application based and domain-specific network policies, and allows them to write management applications without worrying about low-level details such as flow rule fields, network topology related information (e.g end to end paths), etc.

- Define a set of high-level network services that programmers can invoke in their management applications to configure network switches and provide QoS without knowing the details of the SB API (e.g., OpenFlow or an alternative).

- Design and develop a framework that can run management applications similar to the conventional operating systems which run processes (i.e., allow a network management application to use services provided by the framework similar to the way conventional applications use services provided by an operating system).

- Devise a hybrid approach that allows programmers to specify network configuration requirements both proactively, by deriving configuration flow rules from high-level network policies and reactively, by modifying flow rules as flows and conditions change.

- Design and develop a policy conflict management system to detect well-known types of policy conflicts and recommend potential conflict resolution solutions to the managers.

Figure 3.5 illustrates OSDF architecture. The following highlights its key components.

- *Network operation services*: OSDF provides a set of high-level network operation services that programmers use to configure and monitor a network based on high-level network policies that a network administrator provides. Each service takes the following steps to configure network switches based on current active policies in the system:

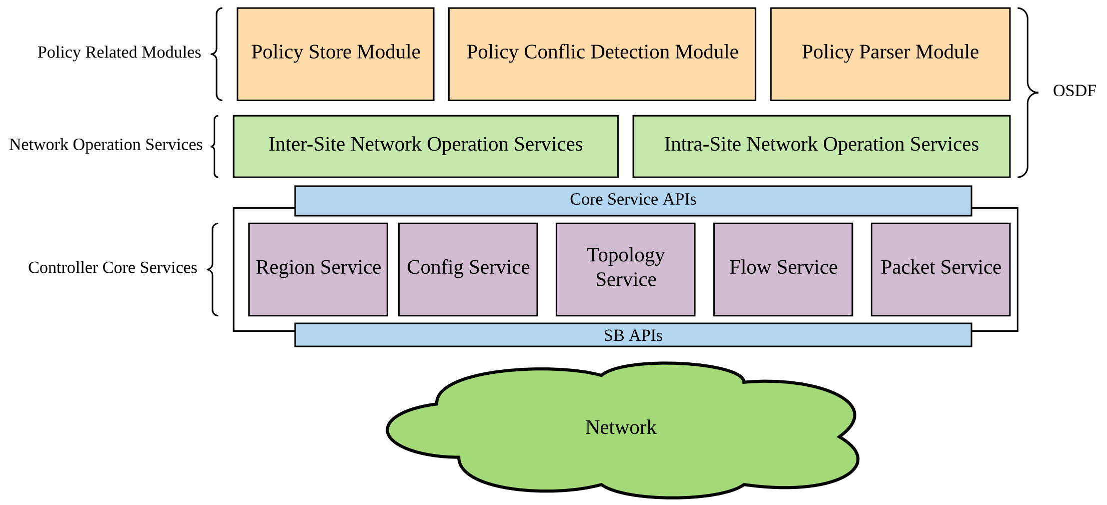

Figure 3.5.: OSDF architecture

1. First, each service reads the high-level network policies from a database of currently active policies and filters them based on the type of service.
2. Second, each service uses the policy parser module to parse filtered policies and generate network-wide forwarding rules for incoming flows.
3. Finally, each service uses flow rule, topology, region, and configuration services to install generated rules in appropriate network switches.

Each service uses a hybrid approach to create basic rules proactively from the high-level requirements that programmers specify in the network policies and to extract low-level information reactively from incoming packets and use the results to generate, install, and update flow rules. Each network operation service uses the following controller core services:

– *Packet Service*: This component provides packet processors that programmers can use to accept and parse incoming packets to the controller reactively to extract low-level match fields (e.g., IP addresses, MAC addresses, ports, and protocol number) to use for configuration of network devices. A network administrator can defer configuration and hide low-level details that is required to configure and monitor a network by processing of incoming packets to the controller reactively. Packet processors parse the current active policies, which are stored in the policy database, and use traffic selector builder service to generate a subset of match fields for a flow based on the application type and high-level requirements for the type that an administrator specifies in the network policies. A packet processor function chooses an appropriate action for each flow rule, based on the high-level operation. For example, in intra-site routing, the flow rules associated with a given flow specify forwarding packets to appropriate outgoing port in each switch. For inter-site routing, rules may specify rewriting the MAC address, *Virtual Local Area Network (VLAN)* tagging, encapsulation, or other actions.

– *Flow Service*: The flow service is responsible for generating and installing flow rules in appropriate network devices. It uses the set of match-action fields that traffic selector builder service generates.

– *Topology Service*: OSDF uses a topology service to find and determine an appropriate path for incoming flows based on high-level network policies and the interconnections among network devices. The *path selection service* uses the topology information that topology service provides to determine an end to end path according to requirements that an administrator specifies in a network policy.

– *Region Service*: A *region* refers to a group of devices located in a common physical (i.e., geographical) or logical region. The region service provides information about devices inside a region. The network regions description parser uses the information that region service provides to distinguish intra-region and inter-region domain traffic flows.

– *Configuration Service*: The configuration service provides an interface that programmers can use to access the items that they define in a configuration file, including both details of individual devices, their IDs and locations, the IP prefixes used, the mapping of IP prefixes to regions, and predefined items, such as default gateways.

- *Policy Store Module*: The policy store module stores and retrieves application-based network policies that an administrator enters to the system. The module provides a policy store management service that high-level network operation services use to read current active policies in the system. In addition, an administrator can list, update, and delete current network policies dynamically at runtime.

- *Policy Parser Module*: The policy parser module is responsible for analyzing application-based policies, incoming flows, and deriving a set of match fields that are then used to generate a set of flow table rules. The module includes of the following three subcomponents that operation services invoke:

  - *Path Selection Service*: This service is responsible for providing a set of pre-defined algorithms for choosing among a set of existing paths between two end points (e.g., shortest path). A network programmer can extend this module by specifying additional path finding algorithms.

  - *Traffic Selector Service*: This service generates a set of match fields based on application based policies and incoming packets. A programmer can extend this module by defining new types of applications and specifying combinations of packet match fields for each application.

  - *Network Region Parser*: The network region parser module uses information that the region service provides, and parses incoming flows and categorizes them based on the regions they span.

- *Policy Conflicts Management Module*: This module is responsible for detecting well known conflicts between the current active policies and providing potential high-level solutions to the network administrator. The module includes the following subcomponents:

- *Policy Conflicts Detection Service*: This service implements a conflict detection algorithm, as explained in subsection 3.4.3 .
- *Policy Conflicts Recommender Service*: This service implements a policy conflict resolution algorithm that suggests potential ways to resolve conflicts among existing policies. Subsection 3.4.4 explains the operation in detail.

Figure 3.6 illustrates packet processing and flow rule installation procedure based on given high-level policies.

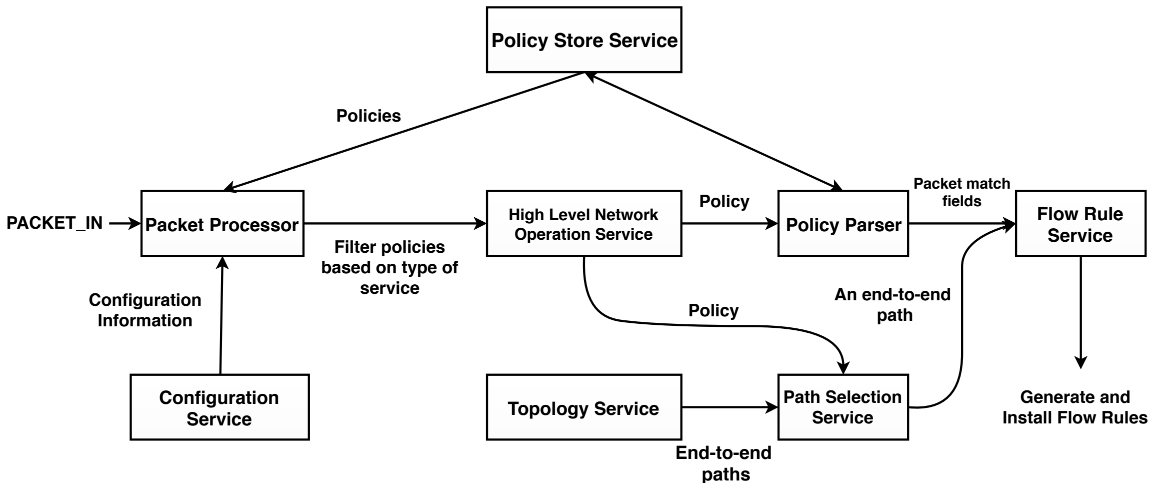

Figure 3.6.: OSDF packet processing and flow rule installation procedure

3.4.1 Application-Based Network Policies

An application-based network policy specifies high-level network requirements for a given application (type of packet). Network administrators and programmers use policies to configure and monitor network devices. We divide all policies into two major categories: *intra-domain policies* (inside a region or a site) and *inter-domain* *policies* (among multiple regions or sites). An application policy includes the following key items:

- *Traffic Profile*: A traffic profile that specifies high-level characteristics and requirements for an application, such as an application name (e.g. WEB), the transport protocol used (e.g. *Transmission Control Protocol (TCP)* or *User Datagram Protocol (UDP)*, and a traffic type (e.g. real time vs best effort). The system provides a set of pre-defined traffic profiles that support typical uses, such as web, video, and voice traffic. A network administrator can extend traffic profiles by introducing new traffic types and applications.

- *High-Level Network Function*: Each policy is associated with a high-level network function that defines configuration and monitoring of network devices, such as intra-site-routing and inter-site-routing. Associating each policy with a network operation allows a packet processor to accommodate policies that are related to a specific function and ignore non-relevant functions.

- *Partial Hosts And Devices Information (address space conditions)*: An administrator has the flexibility to provide high-level information about devices and hosts (e.g specify a name for a host or a network device). Path selection service uses these high-level information to determine an end-to-end path between the source and destination for a specific traffic. For example, an administrator can use device names when specifying that a given type of traffic should pass through a specific set of network devices; the path selection service will choose the best possible path that meets the given requirements. The system uses partial host information when reporting network policy conflicts.

- *Traffic Conditions*: Traffic conditions specify high-level *Quality of Service (QoS)* requirements such as the traffic rate limit for a traffic which specified in the policy.

- *Priority*: An administrator can assign a priority to a policy or use a default. Priorities become important when the systems tries to resolve conflicts among policies.

- *Source And Destination Regions*: A network administrator can define a policy for the interior of a region or can specify traffic routing among multiple regions. To achieve the goal, the system allows a manager to specify both source and destination regions for each policy.

Figure 3.7 defines the syntax of OSDF policy specification.

$$\textit{Policy} ::= \textit{OP}, \textit{APP}, \textit{P}|\phi, \textit{SR}, \textit{DR}, \textit{ASC}|\phi, \textit{TC}|\phi$$
$$\textit{Operation (OP)} \in \textit{High level network operation services}$$
$$\textit{Application (APP)} \in \textit{List of Applications}$$
$$\textit{Priority (P)} \in \mathbb{N}$$
$$\textit{Source Region (SR)} \in \textit{List of Regions}$$
$$\textit{Destination Region (DR)} \in \textit{List of Regions}$$
$$\textit{Address Space Condition (ASC)} \in \textit{List of hosts and network devices}$$
$$\textit{Traffic Conditions (TC)} \in \textit{List of traffic conditions}$$

Figure 3.7.: The syntax of OSDF policy specification

3.4.2 High-Level Network Operation Services

We define a minimum set of high-level network operations to support typical network configurations as follows:

- *Intra-Site-Route:* Network administrators can use this network operation service to specify traffic routing within a specific region according to a network policy for the region.

- *Inter-Site-Route:* Network administrators can use this network operation service to specify traffic routing among multiple regions according to the global policies.

- *Intra-Site-Alert:* Network administrators can use this network operation service to set an alert on specific data traffic and users. If a user attempts to send the traffic inside a region, the system logs user unauthorized attempts and avoid a specific traffic to be routed inside a region.
- *Inter-Site-Alert:* This abstract operation is the same as Intra-Site-Alert except it applies the same operation between multiple regions.
- *Intra-Site-Route-QoS-Provisioning*: Network administrators can use this network operation service for simultaneous traffic routing and rate limiting within a specific region according to a network policy for the region.
- *Inter-Site-Route-QoS-Provisioning*: Network administrators can use this network operation service for simultaneous traffic routing and rate limiting among multiple regions according to the global policies.

3.4.3 Policy Conflict Detection Service

In [67], the authors classify flow rule conflicts based on flow rule details such as layer 2-4 addresses, action, and priority. We use their work as a reference to define types of network policy conflicts based on high-level application based network policies instead of low level flow rule details. As we illustrate in Table 3.1 , we categorize some of the well known network policy conflicts based on application based network policies and explain them as follows:
Suppose two policies, P_i and P_j, specify the same traffic profile, source and destination regions:

- *Redundancy*: P_i is redundant to P_j if both specify the same network operation ($P_{i,OP} = P_{j,OP}$), address space condition of P_i is a subset of address space condition of P_j ($P_{i,SC} \subseteq P_{j,SC}$), and priority of P_i is less than or equal to priority of P_j ($P_{i,p} \leq P_{j,p}$). For example, suppose the following two policies:

1. Route web traffic inside region A using priority 100 between hosts $H1$ and $H2$.
2. Route web traffic inside region A using priority 100 between all hosts.

In the example, the first policy is redundant because second policy is broader (assuming that H1 and H2 are in region A).

- *Shadowing*: P_i is shadowed P_j if each policy specifies a different network operation ($P_{i,OP} \neq P_{j,OP}$), the address space condition of P_i is a subset of the address space condition of P_j ($P_{i,SC} \subseteq P_{j,SC}$), and the priority of P_i is less than to priority of P_j ($P_{i,p} < P_{j,p}$). Shadowing is a critical error because it shows a conflict in a security policy implementation [68]. For example, suppose the following two policies:

  1. Route VIDEO traffic between all hosts in region A and all hosts in region B using priority 100.
  2. Set alert on VIDEO traffic between any hosts in region A and region B using priority 200.

  The first policy is shadowed by the second policy and will never be invoked because its priority is less than the priority of second policy.

- *Generalization*: P_i is a generalization of P_j if each policy specifies a different network operation ($P_{i,OP} \neq P_{j,OP}$), address space condition of P_i is a superset of address space condition of P_j ($P_{i,SC} \supseteq P_{j,SC}$), and priority of P_i is less than to priority of P_j ($P_{i,p} < P_{j,p}$). For example, suppose the following two policies:

  1. Route VOICE traffic inside region A using priority 200 and all host pairs.
  2. Set alert on VOICE traffic inside region A using priority 300 and host pair (H3,H4).

The first policy is a generalization of the second policy because its address space is a superset of address space of the second policy (assuming that H3 and H4 are in region A).

- *Correlation*: P_i is a correlation of P_j if each policy specifies a different network operation ($P_{i,OP} \neq P_{j,OP}$), address space condition of P_i is not a subset or superset of address space condition of P_j but they have some common items ($P_{i,SC} \cap P_{j,SC} \neq \phi$), and priority of P_i is less than to priority of P_j ($P_{i,p} < P_{j,p}$). For example, suppose the following two policies:

  1. Route WEB traffic inside region A using priority 100 and host pairs (H1,H2), (H1,H3).
  2. Set alert on WEB traffic inside region A using priority of 200 and host pairs (H1,H2), (H2,H4).

  The address space of first policy is not a subset or superset of address space of the second policy but their intersection is not empty. Consequently, there is a correlation conflict between above policies (assuming that H1,H2,H3, and H4 are in region A).

- *Overlap*: P_i is an overlap of P_j if each policy specifies the same network operation ($P_{i,OP} = P_{j,OP}$), address space condition of P_i is not a subset or superset of address space condition of P_j but they have some common items ($P_{i,SC} \cap P_{j,SC} \neq \phi$). Note that priority is not important in this case. For example, suppose the following two policies:

  1. Route WEB traffic between regions A and region B using priority 100 and host pairs (H1,H4), (H2,H3).
  2. Route WEB traffic between region A and region B using priority 200 and host pairs (H1,H4), (H2,H5).

  Both of the policies specify the same network operation and the intersection of the first policy address space and second policy address space is not empty.

Consequently, there is an overlap conflict between above policies (assuming that H1,H2 are in region A and H3,H4, and H5 are in region B).

Algorithm 1 illustrates the algorithm that programmers can use to detect policy conflicts based on two given policies.

Table 3.1.: Types of Policy Conflicts

	Traffic Profile (TP)	Src Region (SR)	Dst Region (DR)	Operation (OP)	Address Space Conditions (SC)	Priority (P)
Redundancy	$P_{i,TP} = P_{j,TP}$	$P_{i,SR} = P_{j,SR}$	$P_{i,DR} = P_{j,DR}$	$P_{i,OP} = P_{j,OP}$	$P_{i,SC} \subseteq P_{j,SC}$	$P_{i,p} \leq P_{j,p}$
Shadowing	$P_{i,TP} = P_{j,TP}$	$P_{i,SR} = P_{j,SR}$	$P_{i,DR} = P_{j,DR}$	$P_{i,OP} \neq P_{j,OP}$	$P_{i,SC} \subseteq P_{j,SC}$	$P_{i,p} < P_{j,p}$
Generalization	$P_{i,TP} = P_{j,TP}$	$P_{i,SR} = P_{j,SR}$	$P_{i,DR} = P_{j,DR}$	$P_{i,OP} \neq P_{j,OP}$	$P_{i,SC} \supseteq P_{j,SC}$	$P_{i,p} < P_{j,p}$
Correlation	$P_{i,TP} = P_{j,TP}$	$P_{i,SR} = P_{j,SR}$	$P_{i,DR} = P_{j,DR}$	$P_{i,OP} \neq P_{j,OP}$	$P_{i,SC} \cap P_{j,SC} \neq \phi$	$P_{i,p} \leq P_{j,p}$
Overlap	$P_{i,TP} = P_{j,TP}$	$P_{i,SR} = P_{j,SR}$	$P_{i,DR} = P_{j,DR}$	$P_{i,OP} = P_{j,OP}$	$P_{i,SC} \cap P_{j,SC} \neq \phi$	Not Necessary

Algorithm 1: Conflict detection algorithm based on high-level policies

```
Conflict Detection ($P_i, P_j$);
Input  : Two policies $P_i$ and $P_j$
Output: Return the type of the conflict
if $P_{i,TP} \neq P_{j,TP}$ OR $P_{i,SR} \neq P_{j,SR}$ OR $P_{i,DR} \neq P_{j,DR}$ then
    return $NO\_CONFLICT$
end
if $P_{i,SC} \subseteq P_{j,SC}$ AND $P_{i,p} \leq P_{j,p}$ AND $P_{i,OP} = P_{j,OP}$ then
    return $P_{i,Redundancy}$;
end
if $P_{i,SC} \subseteq P_{j,SC}$ AND $P_{i,p} \leq P_{j,p}$ AND $P_{i,OP} \neq P_{j,OP}$ then
    return $P_{i,Shadowing}$;
end
if $P_{i,SC} \supseteq P_{j,SC}$ AND $P_{i,p} \leq P_{j,p}$ AND $P_{i,OP} \neq P_{j,OP}$ then
    return $P_{i,Generalization}$;
end
if $P_{1,SC} \cap P_{2,SC} \neq \phi$ AND $P_{i,p} \leq P_{j,p}$ AND $P_{i,OP} \neq P_{j,OP}$ then
    return $P_{i,Correlation}$;
end
if $P_{1,SC} \cap P_{2,SC} \neq \phi$ AND $P_{i,OP} = P_{j,OP}$ then
    return $P_{i,Overlap}$;
end
```

3.4.4 Policy Conflict Recommendation Service

This service implements a policy conflict recommendation algorithm as Algorithm 2 illustrates to provide a high level advice about how an administrator can resolve the conflicts between current policies in the system. Each type of conflict is resolved as follows:

- *Redundancy*: To resolve this type of conflict, a network administrator can remove a policy that has a lower or equal priority and its address space is a subset of another policy.
- *Shadowing*: To resolve this type of conflict between two policies, a network administrator can remove the policy that is shadowed (i.e. the policy which has lower priority and its address space is a subset of another policy).
- *Generalization*: To resolve this type of conflict between two policies, a network administrator can remove the policy that has a more generalized address space, and then update its address space conditions by removing those conditions that are specified in another policy address space. Finally, the network administrator inserts the updated policy into the system.
- *Correlation*: To resolve this type of conflict between two policies, a network administrator can update the address space of one of the policies (e.g the policy by lower priority) by removing the common items and then inserting the updated policy into the system.
- *Overlap*: To resolve this type of conflict between two policies, a network administrator should first remove both of the policies from the system and then insert a new policy with an address space equal to the union of both of address spaces.

The asymptotic time complexity of the policy conflict recommendation algorithm is $O(n^2)$ where n is the number of policies being considered. Most practice, n is small because a single policy covers a broad range of network configurations.

Algorithm 2: Policy conflict recommender algorithm

```
Policy Conflict Recommender Algorithm ($P_i, P_j$);
Input  : Two polices $P_i$ and $P_j$
Output: A set of recommendations for resolving the conflict
$Conflict\_Type = Conflict\_Detection(P_i, P_j)$;
if $Conflict\_type == P_{i,Redundancy}$ then
    Remove policy $P_i$;
end
if $Conflict\_type == P_{i,Shadowing}$ then
    Remove policy $P_i$;
end
if $Conflict\_type == P_{i,Generalization}$ then
    Remove policy $i$ temporary;
    Update address space condition of policy $i$ as follows:
    $(P_{i,SC} = P_{i,SC} - P_{j,SC})$;
    Insert policy i;
end
if $Conflict\_type == P_{i,Correlation}$ then
    Remove policy $P_i$ temporary;
    Update address space condition of policy $i$ as follows:
    $(P_{i,SC} = P_{i,SC} - P_{j,SC})$;
    Insert policy $P_i$;
end
if $Conflict\_type == P_{i,Overlap}$ then
    Remove policy $i$ and $j$ temporary;
    Update address space condition of policy $i$ as follows:
    $(P_{i,SC} = P_{i,SC} + P_{j,SC})$;
    Insert policy $i$;
end
```

3.4.5 Comparison with SDN Programming Frameworks and Languages

In this subsection, we compare OSDF with the SDN programming frameworks and programming languages introduced in Chapter 2 including *Merlin*, *Frenetic*, *Procera*, *Kinetic*, *Nettle*, *PonderFlow*, *Pyretic*, and the *ONOS intent framework*. We begin by summarizing key characteristics of the SDN programming systems and languages to capture the essence of each. We then use the characteristics as a base to classify and analyze each of the systems, as in [31]:

- *Programming Paradigm*: A programming paradigm defines how key components and building blocks of a software system interact. As Chapter 2 explains, SDN programming languages and frameworks can be classified into one of seven major categories: *imperative*, *declarative*, *logic*, *event-driven*, *functional*, *FRP*, and *reactive*.

- *Flow Installation Paradigm*: The abstractions that are defined part of the SDN programming languages and frameworks must be translated into a set of flow rules that implement the required network configuration and management functions. There are two approaches for generating and installing flow rules: proactive and reactive. As Chapter 2 explains, the proactive paradigm requires management applications to install flow rules in each network device before network traffic arrives. By contrast, the reactive paradigm allows a management application to change the flow rules in network devices dynamically.

- *Policy Definition*: A policy can be categorized into two major categories: *static* and *dynamic* [31]. A static policy defines a set of predefined outcomes based on a set of predefined parameters. A dynamic policy specifies actions to be triggered when conditions arise or events occur, such as packet loss, link or node failures, congestion occurs, or packets arrive on new flows.

- *Functionality*: Programmers use the abstractions that SDN programming systems offer to implement management applications and network functions. We use the term *functionality* to characterize the capabilities and richness of the programming environment along with the expressive power. A system in which the functionality suffices for all network functions and management needs is desirable.

- *Implementation Details*: Although the four characteristics listed above are fundamental, we include implementation details as a fifth characteristic because such details may limit the usefulness and generality of the system. For example, a system that only permits applications to be written in one particular programming language is inherently less general than a multi-lingual system.

Table 3.2 summarizes how OSDF compares with the SDN programming languages and frameworks discussed earlier. The table lists multiple perspectives, including the programming paradigm, flow installation paradigm, and policy definition. As we discussed earlier, network event processing is one of the key aspects of current SDN controllers because management applications use events to implement control plane functions and services. OSDF offers an event-driven framework that allows a management application to react to network changes, such as topology events (e.g. link or device failures), packet events, etc. PonderFlow and Kinetic also use an event-driven programming approach, but both use low-level abstractions that require a programmer to deal with flow rules and packet fields. OSDF raises the level of abstraction thereby hiding the details from users. Most of the SDN programming languages such as Merlin, Frenetic, Procera, Nettle, PonderFlow, and OSDF use a reactive approach for generating and installing flow rules, which gives programmers more flexibility in the design and implementation of management applications. In addition, most of the SDN programming languages and frameworks, including OSDF, support both static and dynamic policies. Combination of an event-driven approach and reactive flow installation paradigm that is used in OSDF gives programmers and network administrators flexibility to define policies that can be triggered whenever there is a change in network conditions.

Table 3.2.: SDN Programming Languages and Frameworks Characteristics

	OSDF	Merlin	Frenetic	Procera	Kinetic	Nettle	PonderFlow	Pyretic	ONOS Intent
Programming Paradigm	DE/ED	F	FRP	FRP	ED	FRP	ED	I	DE
Flow Installation Paradigm	RFI	RFI	RFI	RFI	RFI/PFI	RFI	RFI	RFI/PFI	PFI
Policy Definition	S/D	S/D	S/D	S/D	S/D	S/D	S/D	S/D	S

I=Imperative,DE=Declarative, F=Functional, FRP, ED=Event-Driven, L=Logic, RP=Reactive Programming
RFI=Reactive Flow Installation, PFI=Proactive Flow Installation
S=Static, D=Dynamic

Table 3.3 illustrates how OSDF compares to other SDN programming systems with respect to functionality. For the purpose of this comparison, we divide functionality into five broad categories:

- *Basic Network Functions*: This category refers to minimum required network functions, such as packet classification and forwarding that a SDN programming framework should support for implementing of basic SDN management applications.

- *Traffic Engineering*: This category refers to SDN management applications that allows us to monitor network traffic, monitor network changes, and decide how to route network traffic to improve utilization of network resources and performance (i.e. QoS, path selection, link failure detection, and dynamic rerouting).

- *Network Virtualization*: This category refers to the techniques such as slicing and the use of virtual switches that allow one to define and run multiple virtual networks overlaid on a shared network infrastructure. Some of the SDN programming frameworks use software switches to create multiple virtual switches in a physical switch. In addition, a network slice refers to a subset of switches, ports, and, links that logically defines a network. [31].

- *Monitoring and Telemetry*: This category refers to functions that programmers can use to gather network statistics, such as number of packets that matches to a specific flow rule. In addition, to implement dynamic management applications, programmers must be able to monitor network changes such as topology changes. Furthermore, SDN programming frameworks usually provide functions that allow network administrators to monitor users' activities and detect anomalous behavior, such as a potential security attack.

- *Security*: This category cuts across some of the others, and most SDN programming languages and frameworks support basic security functions that are needed in a network, such as access control. To implement access control functionality using SDN, a set of abstractions are needed that allow programmers to define access of a user to services or resources in a network. For example, network administrators must be able to isolate tenants' access to the servers and services in a multi-tenant data center network.

Table 3.3.: Functionalities of SDN Programming Languages and Frameworks

	Categories	OSDF	Merlin	Frenetic	Procera	Kinetic	Nettle	PonderFlow	Pyretic	ONOS Intent
Basic Network Functions	Packet Classification	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Forwarding	✓	✓	✓	✓	✓	✓	✓	✓	✓
Traffic Engineering	QoS	✓	✓			✓				
	Path Selection	✓	✓							
	Link Failure Detection	✓								
Network Virtualization	Slicing	✓	✓						✓	
	Virtual Switches								✓	
Monitoring and Telemetry	Query Language			✓						
	Alerting Mechanism	✓								
	Windowed History		✓	✓	✓	✓			✓	
Security	Access Control	✓	✓					✓		

As Table 3.3 illustrates, all of the SDN programming languages and frameworks provide a minimum set of abstractions to support basic network functions. In addition, OSDF, Merlin, Pyretic, and Kinetic are examples of SDN programming systems that go beyond supporting basic network functions and provide abstractions that can be used to support traffic engineering, network virtualization, monitoring and telemetry, and security management applications. OSDF provides capabilities such as a path selection service that allows a system to reroute a specific network traffic along an alternative path whenever a link or device fails. In addition, Frenetic provides a query language that can be used to retrieve network statistics and implement dynamic SDN management applications. Although it can be extended to support additional monitoring capabilities, OSDF uses native SDN controller capabilities to retrieve the required network statistics and does not introduce redundant functions for monitoring and telemetry. OSDF and Merlin both provide almost the same functionality, and both try to simplify network programming. However, Merlin requires the use of a predicate language to specify a set of match-action fields and to generate and install flow rules. OSDF hides such details from programmers and network administrators.

Finally, we compare the proposed SDN programming languages and frameworks in terms of the programming language that is used to implement each of the systems as Table 3.4 illustrates. Java and Python are the two popular programming languages that are used for implementing SDN management applications and program support systems. In addition, most of the systems discussed in this dissertation follow an open-source approach, making their code available to the researchers and developers.

Table 3.4.: Implementation Details of SDN Programming Languages and Frameworks

	OSDF	Merlin	Frenetic	Procera	Kinetic	Nettle	PonderFlow	Pyretic	ONOS Intent
Programming Language	Java	OCaml	Python	Haskell	Python	Haskell	Java	Python	Java
Open Source	✓	✓	✓		✓	✓		✓	✓

3.4.6 Potential Use of an Event Distribution System

Packet processing based on a set of high-level policies is one of the key design aspects of OSDF. Programmers have two potential options for processing of packets to implement each network operation service:

- *Internal Packet Processing*: In this approach, programmers can use the built-in packet processors that are available in most of the current SDN controllers for processing of incoming packets to generate appropriate flow rules based on high-level current active policies in the system.

- *External Packet Processing*: In this approach, programmers can use an event distribution system to externalize packet processing to implement each of the network operation services as a microservice outside of the controller. In other words, programmers can implement OSDF and its modules as an external and controller independent application as far as we equip an SDN controller core to an event distribution mechanism that allows us to externalize packet processing.

For example, programmers can use Umbrella services and its APIs to implement an intent-based SDN programming framework like OSDF.

4 IMPLEMENTATION DETAILS OF THE PROPOSED ARCHITECTURES

This section explains the implementation details of each of the frameworks proposed in Chapter 3, including the details about the technologies, platforms, and programming languages that we used to implement each framework.

4.1 An Overview of Apache Kafka

Apache Kafka is a streaming platform that provides the following capabilities [69]:

- Offers a publish and subscribe messaging system that programmers can use to stream records, similar to a message queue or enterprise messaging system.
- Stores streams of records in a fault-tolerant manner.
- Processes streams of records as they occur in the system.

Kafka uses a *topic* that corresponds to a category (or *feed*) to publish the records. Topics in Kafka are always multisubscriber, meaning that zero, one, or many consumers can subscribe to the topic and receive its data. Figure 4.1 illustrates a producer-consumer model that uses Kafka and includes the following key components:

- Producers: Producers are processes that publish data to the topics of their choice.
- Consumers: Consumers of topics pull messages off a Kafka topic.
- Kafka Cluster: A Kafka cluster consists of one or more servers (i.e. Kafka brokers) that run Kafka software.

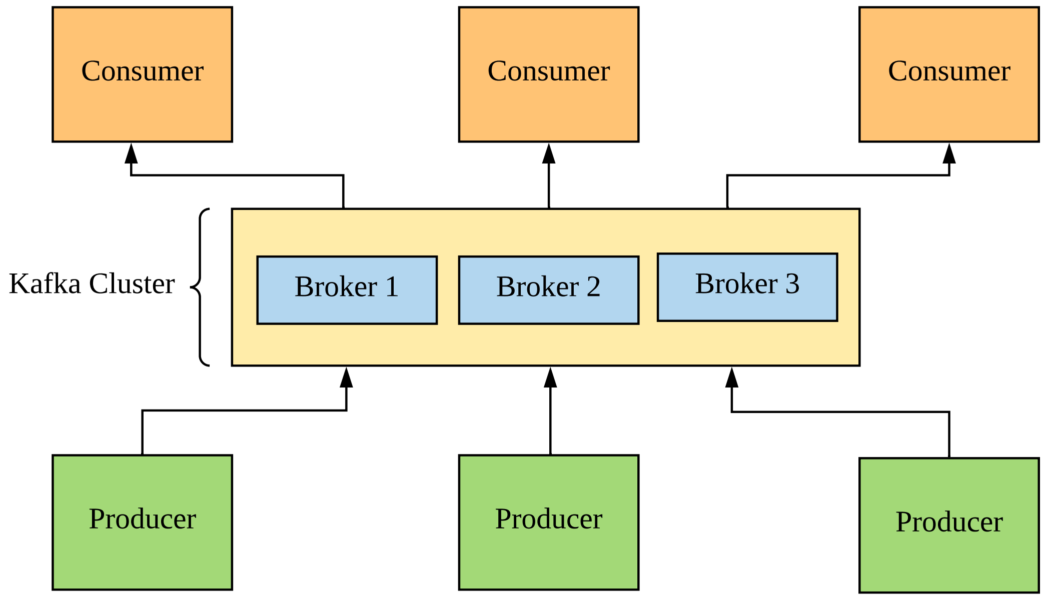

Figure 4.1.: A producer-consumer model using Apache Kafka

Kafka uses the TCP protocol to enable communication among Kafka clients and servers. Furthermore, Kafka has four core APIs:

- *Producer API*: The producer API allows an application to publish a stream of records to one or more Kafka topics.
- *Consumer API*: The consumer API allows an application to subscribe to one or more topics and consume the stream of records.
- *Streams API*: The streams API allows an application to act as a stream processor to consume an input stream from one or more topics and produce an output stream to one or more output topics.
- *Connector API*: The connector API provides the feature of building and running reusable producers or consumers that connect Kafka topics to existing applications or data systems.

4.2 An Overview of gRPC

gRPC [70] is an open source project initially designed by Google to provide an RPC system. gRPC uses *Hyper Text Transfer Protocol (HTTP)* version 2 [71] as a transport mechanism to make communication possible between two end-points of a messaging system and uses protocol buffers [72] as the *Interface Description Language (IDL)* for serializing structured data (i.e. it specifies underlying message interchange format). As Figure 4.2 illustrates, the idea of gRPC is based on defining a service and specifying the methods of a service that a client can call remotely with their parameters and return types. In the client side, a gRPC stub can directly call methods on a server application and a different machine by sending a protobuf request. On the other hand, the server implements a service and runs a gRPC server to handle client calls. A programmer can implement gRPC servers and clients in variety of programming environments. For example, a programmer can implement a gRPC server in Java and a set of clients in Go [73], Python [74], or C++. gRPC provides four types of RPCs that are explained briefly as follows:

- *Unary RPC*: This is the simplest form of an RPC that gRPC provides, where a client sends a single request and gets back a single response.
- *Server Side Streaming RPC*: In the server-side streaming RPC, a client sends a request to the server and receives a stream that can be used to read a sequence of messages until there are no more messages.
- *Client Side Streaming RPC*: In the client side streaming RPC, a client writes a stream of messages on a gRPC channel to send them to the server. Once the client has finished writing the messages on the channel, it waits for the server to read the messages and return a response.
- *Bidirectional Streaming RPC*: In bidirectional streaming RPC, both the client and server send a stream of messages using a read-write stream. The two stream

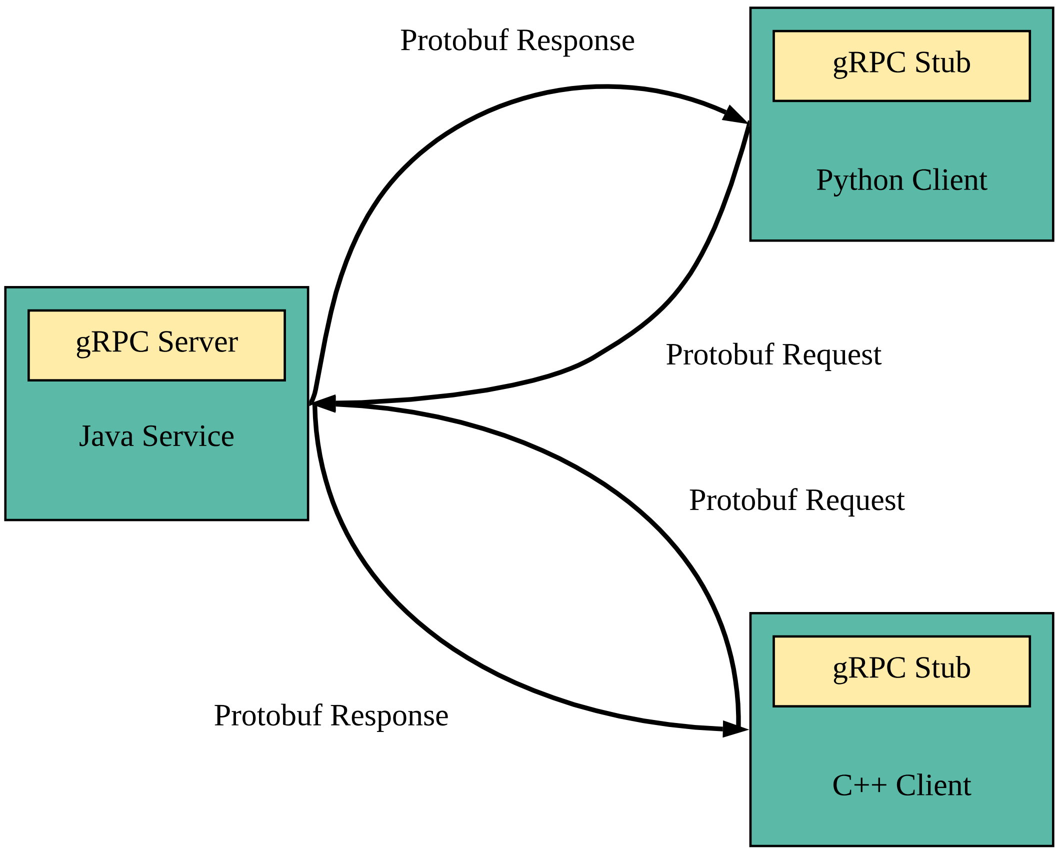

Figure 4.2.: The gRPC client-server architecture

channels operate independently, which means the clients and servers can read and write their messages in whatever order they choose.

4.2.1 An Overview of Protocol Buffers

Protocol buffers are like XML [75] that can be used to serialize structured data in an efficient, simpler, faster, and more flexible manner. A protobuf file is an ordinary text file with a .proto extension. In a protobuf file, programmers can define the following key components:

- *message*: Protocol buffer uses a message to define protobuf data, where each message is a small logical record of information containing a series of name-value pairs called fields.
- *service*: A service defines the RPC methods that clients can call from a remote server that implements the service.

After defining the protobuf messages and services for a specific purpose (e.g. an event distribution system), a programmer uses a protobuf compiler to auto generate data classes for multiple programming languages.

4.3 Implementation Details of an Event Distribution System

We use two technologies including Apache Kafka [69] and gRPC [70] to implement an event distribution system using publish-subscribe and point-to-point messaging paradigms. The following subsections explain each of the proposed event distribution systems:

4.3.1 The Architecture of an Event Distribution System Using Kafka

Figure 4.3 illustrates an architecture that uses Apache Kafka to externalize event processing which has the following key components:

- **Kafka Event Distribution Application**: The architecture follows the producer-consumer model for the event distribution based on Apache Kafka. A Kafka event distribution application listens for events that occur in the controller core, publishes the events on a Kafka cluster, and external processes and applications consume the events. In other words, the event distribution application acts as a producer by pushing data to brokers on the Kafka cluster, and external applications and services act as consumers that receive the data.
- **Applications and Services**: As Figure 4.4 illustrates, whenever an application needs to receive incoming events from the controller, the application sends

an HTTP request to the event distribution application to subscribe to a specific type of event (e.g a packet event, topology event, etc.). The Kafka event distribution application checks the request, subscribes the requesting app to the specified type of event, and then replies to the requesting app. In our implementation, the Kafka event distribution application encodes each packet in a protobuf message and publishes the message as an array of bytes to the Kafka cluster. Whenever an application receives an event by consuming it from Kafka cluster, the application must decode and parse the event.

- **Kafka Cluster**: Each Kafka broker runs a Kafka server form a cluster of servers. The event distribution system in the controller core acts as a producer that publishes events into Kafka *topics* within the broker. A consumer of a given topic consumes all messages on that topic.

4.3.2 The Architecture of an Event Distribution System Using gRPC

The second type of event distribution mechanism uses gRPC *server side streaming.* As Figure 4.5 illustrates, we equip an SDN controller with a gRPC event distribution mechanism and use server side streaming to propagate events. The application implements a push-notification system that provides incoming events to external processes. As Figure 4.6 shows, an external application or service sends a registration request to the gRPC server to register as a receiver. Then the external application subscribes to a topic (in this case a specific type of event) which causes the server to start streaming occurrences of the specified type of event.

Appendix A defines the protobuf messages and the services used to implement the above behavior. The event notification service consists of two methods: 1) a *register* method that takes care of the incoming registration requests from the clients to register them in the server and 2) *onEvent* method that streams notification messages for each type of event. Our implementation extends the *onEvent* method to support multiple types of events by adding listener interfaces for additional event types.

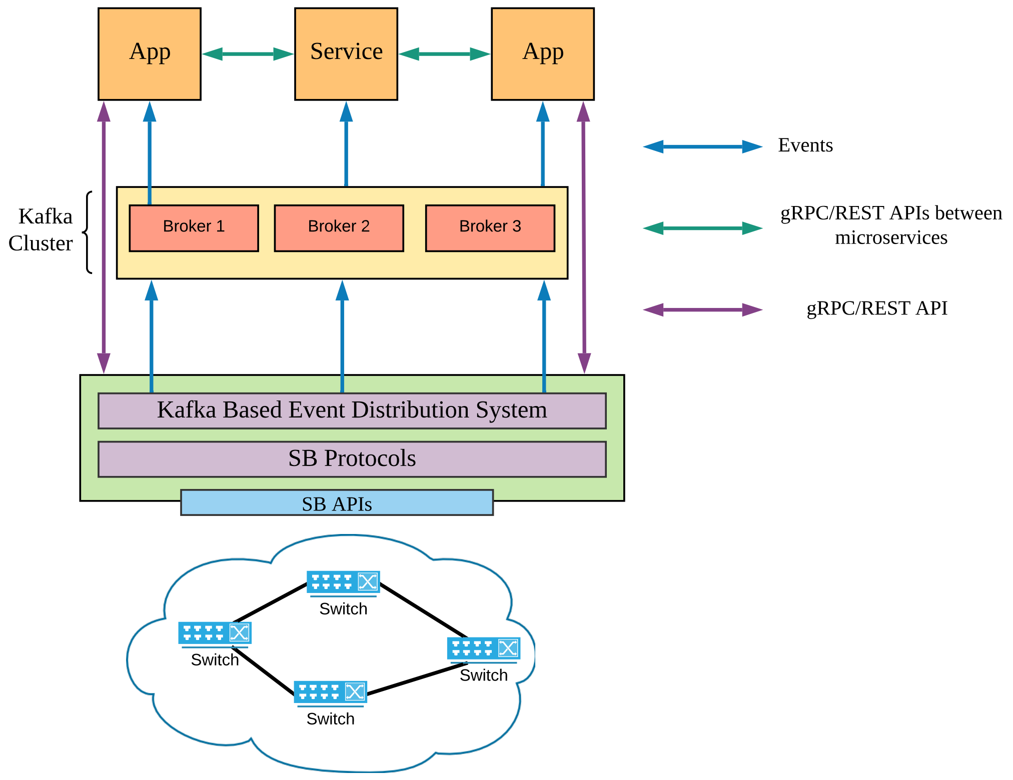

Figure 4.3.: Illustration of using Kafka for event distribution with an SDN controller

4.3.3 Using the Event Distribution Mechanisms with Current SDN Controllers

The point of injecting an event distribution mechanism into SDN controllers arises from our goal of moving toward disaggregated control plane services (i.e., restructure the controller as a set of cooperative microservices). As a first step, we equipped current SDN controllers with an event distribution system to measure and assess externalization of event processing by implementing of external reactive and proactive based management applications. To make the result useful, we defined a set of complimentary gRPC services, and then used existing NB APIs as the interface to external applications. For example, an external application or service that processes incoming packet events may need to return the packets that caused the event to the pipeline. To permit such processing, we defined a gRPC service that an application or service can invoke to return a packet to the pipeline. In addition, if an external ap-

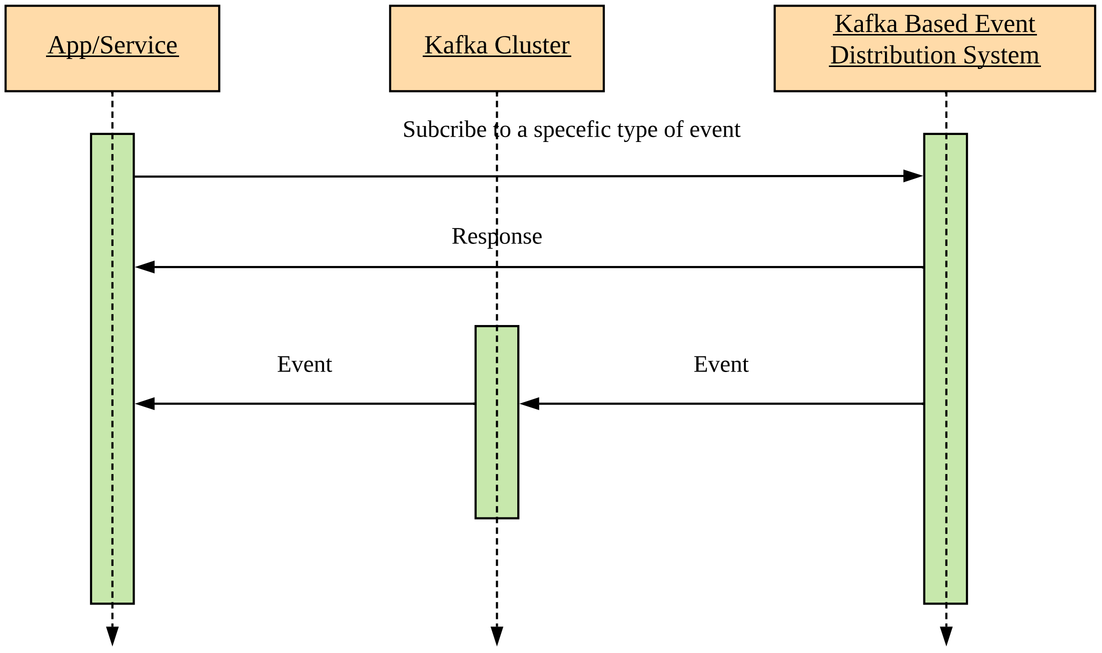
App/Service
Kafka Cluster
Kafka Based Event Distribution System
Subcribe to a specefic type of event
Response
Event
Event

Figure 4.4.: The message passing protocol for the Kafka based event distribution system

plication or service needs to install, remove, or update flow rules on network devices, and retrieve topology information, the application has a choice of using a traditional REST API or gRPC. Figures 4.7 and 4.8 show the use of Kafka and gRPC event distribution mechanisms with current SDN controllers.

4.4 Implementation Details of Umbrella Framework

We used Java programming language to implement a prototype of the Umbrella framework [76]. We wrote driver modules for the ONOS and OpenDayLight SDN controllers. Umbrella provides a set of high-level and controller independent APIs and services that programmers can use for implementing of SDN management applications. As we already pointed out, SDN controllers provide NB APIs such as REST API that are different in terms of naming convention, syntax, data resources, and

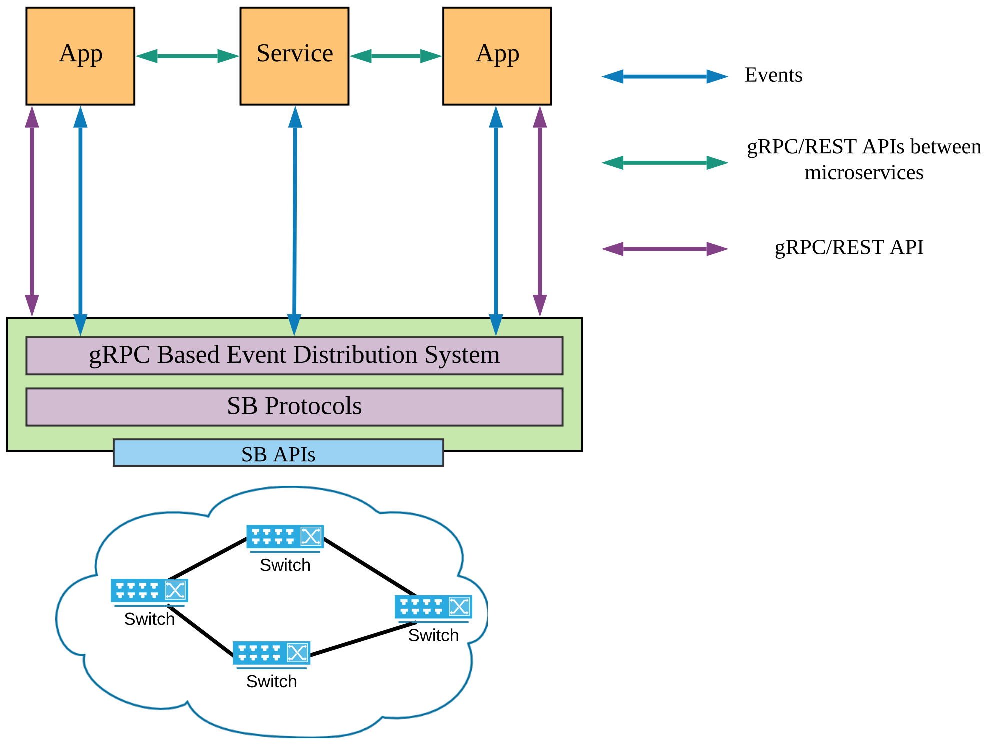

Figure 4.5.: An illustration of using gRPC event distribution with an SDN controller

usage. However, since most of the SDN controllers follow the same paradigms for defining control plane services, it is possible to define a set of high-level abstractions to be independent of the SDN controllers. In the first version of the framework, we define a set of minimum required unified and controller independent services including a flow service, a topology service, an event notification service that we explain the implementation details of each in the following subsections.

4.4.1 Flow Service

The flow service defines a set of high-level abstractions based on the match-action paradigm that most of the current SDN controllers such as ONOS and OpenDayLight

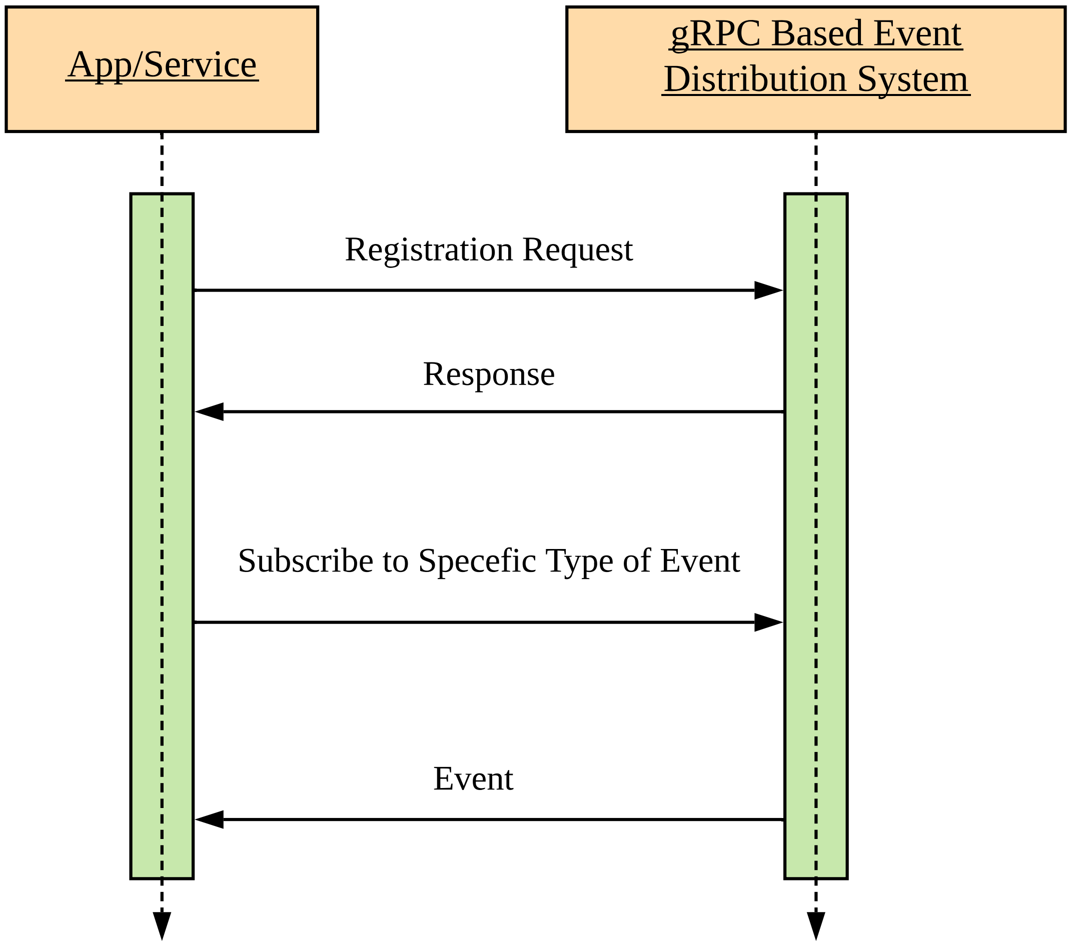
App/Service
gRPC Based Event Distribution System
Registration Request
Response
Subscribe to Specefic Type of Event
Event

Figure 4.6.: The message passing protocol for the gRPC based event distribution system

use for installing, removing, updating flow rules, and retrieving their statistics. Flow service provides a flow abstraction which has the following key items:

- *FlowMatch*: This class defines a set of match fields such as layer 2 (e.g. MAC address), layer 3 (e.g. IP addresses, protocol number), layer 4 fields (e.g. TCP or UDP ports). Programmers can expand this class to support new protocols.

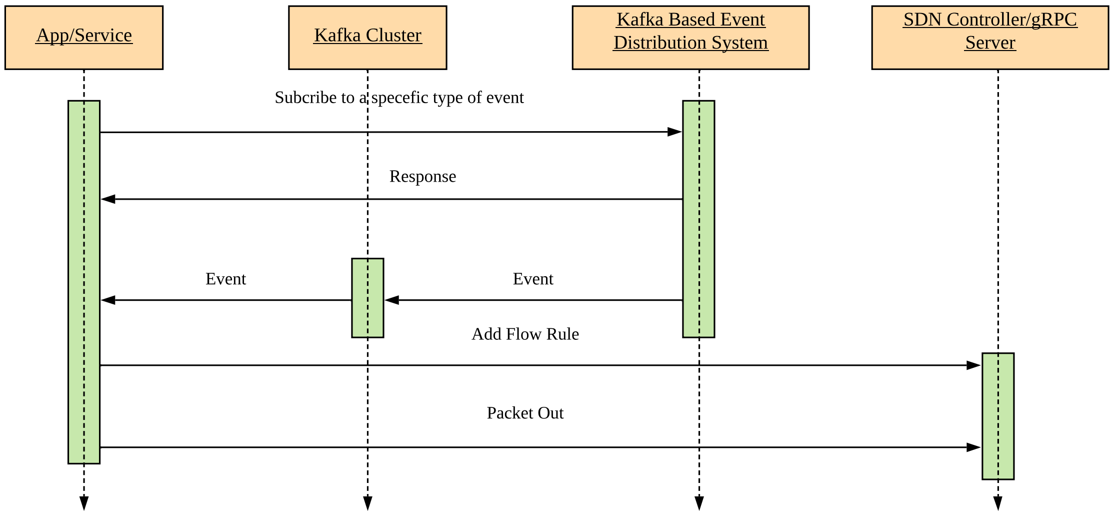
App/Service
Kafka Cluster
Kafka Based Event Distribution System
SDN Controller/gRPC Server
Subcribe to a specefic type of event
Response
Event
Event
Add Flow Rule
Packet Out

Figure 4.7.: Usage of Kafka based event distribution system with current SDN controllers

- *FlowActions*: This class defines a set of actions that each network device performs on incoming packets. Some of the typical actions that Umbrella framework supports are *DROP* that drops a packet, *OUTPUT* that forwards a packet to an output port, *CONTROLLER* that sends packet to the controller for further processing, and *GOTO_TABLE* that sends a packet to a specific table for further processing. Programmers can expand this class to support new actions.
- *Priority*: This integer field specifies the priority of a flow rule.
- *TableId*: This integer field specifies the id of the table that programmers use for installing, removing, and updating a set of flow rules on a network device.
- *DeviceId*: Each SDN controller assigns a unique id to each of the network devices in the network topology and Umbrella uses this id for finding devices in a network topology and for installing, removing, and updating a set of flow rules on a network device.

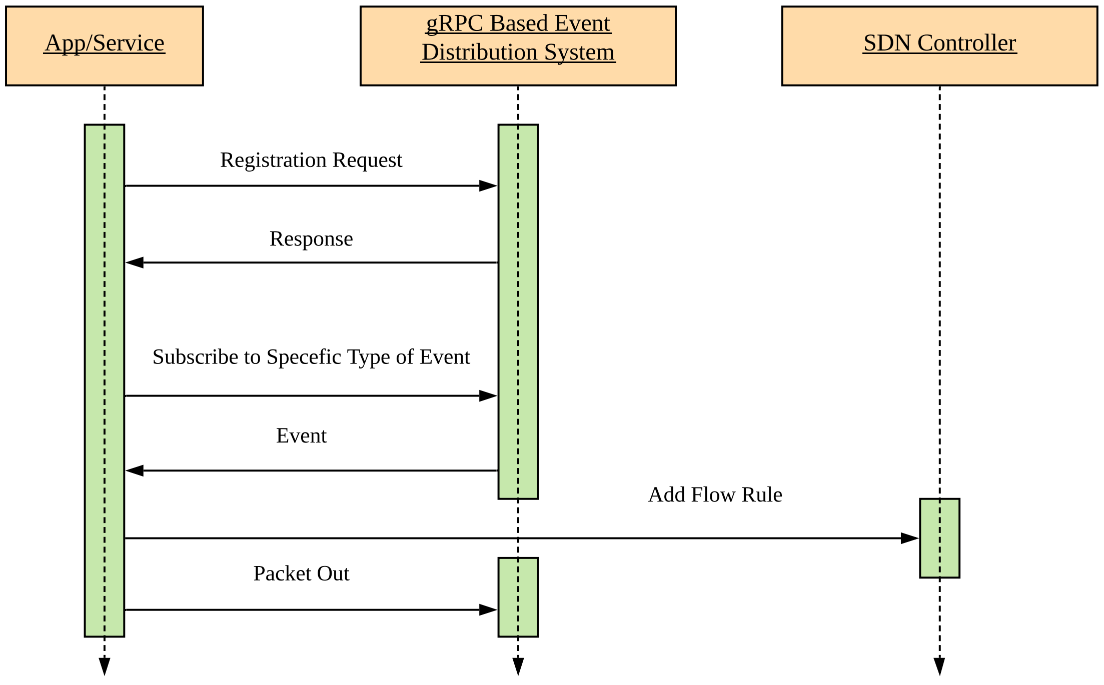

Figure 4.8.: Usage of gRPC based event distribution system with current SDN controllers

- *isPermanent*: This boolean field specifies if programmers should install a flow rule permanently or they need to specify a time out.
- *AppId*: AppId associates each flow rule with the id of an application that installs that flow rule.
- *Cookie*: This field specifies the flow rule cookie.
- *TimeOut*: Programmers can assign a time-out to each of the flow rules.
- *FlowID*: SDN controllers assign a unique ID to each of the flow rules that programmers can use for retrieving flow rule statistics, updating, and deleting flow rules.

Appendix B defines the Flow abstraction using the Java programming language.

4.4.2 Topology Service

Most of the SDN controllers use a graph as an abstraction to represent the network topology. Umbrella uses the same paradigm to define a set of unified abstractions on top of the NB APIs that are used to retrieve topology information. The Topology service defines the following abstractions for a network topology store:

- *TopoVertex*: This class represents a node in the network topology. Programmers can extend the class to represent all types of network devices in a network topology such as a *TopoHost* and a *TopoSwitch*.

- *TopoEdge*: This class represents a link in the network topology. Each *TopoEdge* has the following fields to specify the two end points of each link in the network topology:

  - *Src*: This field specifies the source end point (i.e. source device) in a link in the network topology.
  - *Dst*: This field specifies the destination end point (i.e. destination device) in a link in the network topology.
  - *SrcPort*: This field specifies the source port (i.e. source port of a device) in a link in the network topology.
  - *DstPort*: This field specifies the destination port (i.e. destination port of a device) in a link in the network topology.
  - *State*: This field specifies the state of a link which can be active or inactive.
  - *Type*: This field specifies the type of each edge in the network topology. We categories the links into three major categories: *SWITCH_TO_SWITCH*, *SWITCH_TO_HOST*, and *HOST_TO_SWITCH*.
  - *Weight*: Programmers can use this field to assign a weight to each link in the network topology.
  - *Label*: Progrmmers can use this field to assign a label to each edge in the network topology.

Appendices C and D contain definitions of the TopoVertex and TopoEdge abstractions using the Java programming language.

4.4.3 Event Notification Service

Umbrella provides an event notification service that management applications can use to receive network events such as topology events, packet events, flow rule events, etc. The Umberella event notification server has two major components:

- *eventMonitor API*: Umbrella defines an event monitor API class that can be extended to monitor different types of events. An instance event monitor class can add an event listener to listen to a specific type of event.

- *eventListener API*: Umbrella defines an event listener API class which has an *onEvent* callback function that will be called whenever an event occurs.

4.5 Implementation Details of OSDF

We used the Java programming language to implement a prototype [77] of OSDF as a reactive based application for ONOS [2] SDN controller. OSDF uses the *PacketService* class in ONOS to implement packet processors for parsing of incoming packets and filter them based on current active policies in the system for further processing. OSDF also uses the ONOS *TopologyService* class to retrieve topology information and compute end-to-end paths between two endpoints, *FlowService* to install, update, and delete flow table rules based on current active high-level policies in the system, and *RegionService* class to define a region of network devices to distinguish intra-site and inter-site high-level network policies.

5 PERFORMANCE EVALUATION AND EXPERIMENTAL RESULTS

This chapter compares the advantages and disadvantages of the proposed event distribution systems. In addition, it presents an evaluation of the Kafka and gRPC based event distribution mechanisms using various performance metrics such as response time and throughput. Furthermore, it explains the Umbrella programming APIs and their usage. Finally, the chapter presents an evaluation of the performance and functionality of the OSDF framework.

5.1 Emulation and Experimental Setup

For experimental results of this dissertation, we deployed an SDN testbed consists of 10 Ethernet switches that logically define 5 sites (departments of an organization) that are connected together as Figure 5.1 illustrates. To make our testbed emulate an enterprise network, we divide each physical switch logically into 10 independent smaller switches using virtualized mode. Each site is a Fat-tree [78] network topology as Figure 5.2 illustrates. The connections between virtual switches are 1GB links and we aggregate 1GB links to connect sites, thereby emulating high capacity links. Overall, our network consists of 50 virtual switches and more than 130 links. In addition to an SDN testbed, we used the *Mininet* [79] network emulator to evaluate the proposed architectures at a larger scale.

5.2 A Comparison of Kafka and gRPC Based Event Distribution Systems

The advantages and disadvantages of the two event distribution mechanisms can be summarized:

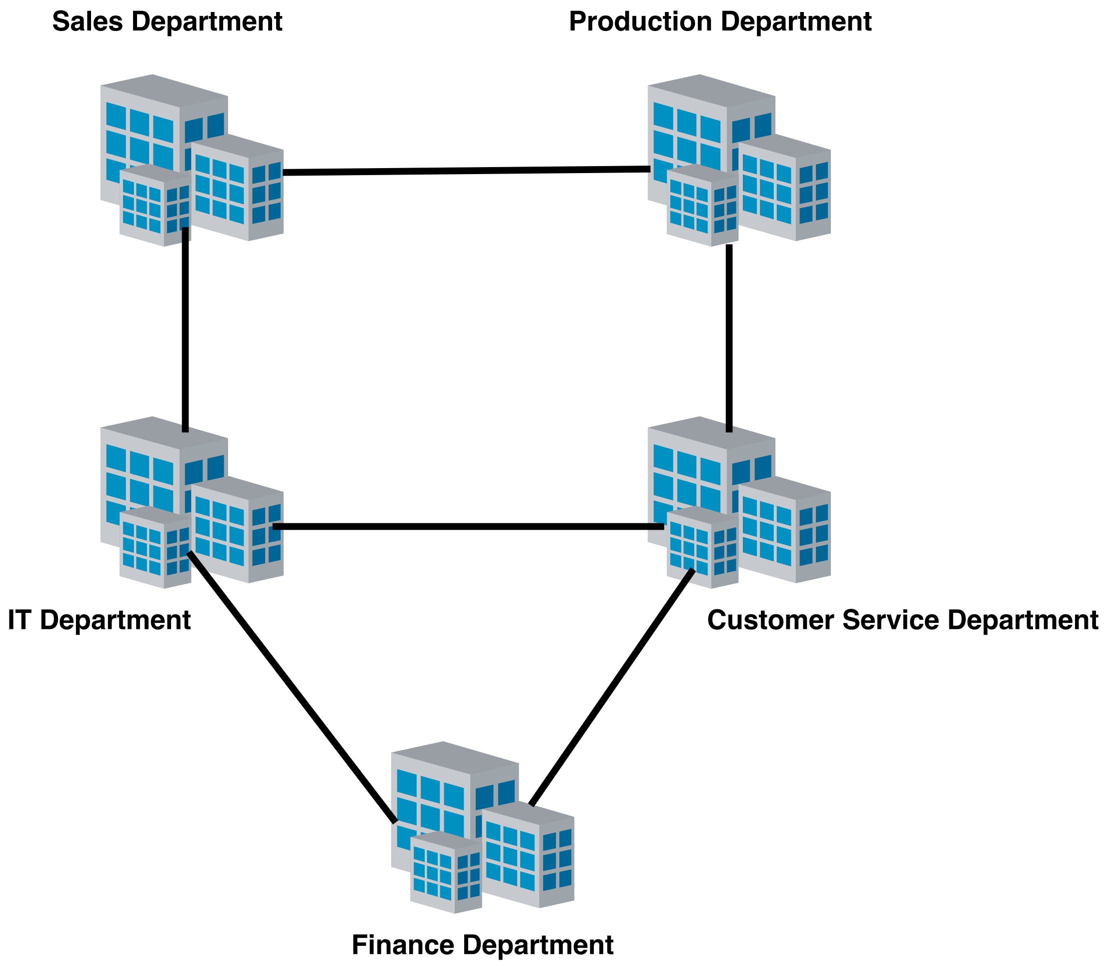

Figure 5.1.: Inter-sites network topology

- *Programming Language Agnosticism*: Each of the two event distribution mechanisms provides a facility that can be used to implement services and applications in arbitrary programming languages. From the client side perspective, however, gRPC makes it easier than Kafka to expand a distribution system to include apps written in new programming languages. To use gRPC with a new language, a programmer only needs to compile the set of protobuf messages used for communication between the event distribution system and each microservice. The gRPC technology automatically generates the gRPC stubs that external SDN applications and services need. In contrast, to use Kafka with a

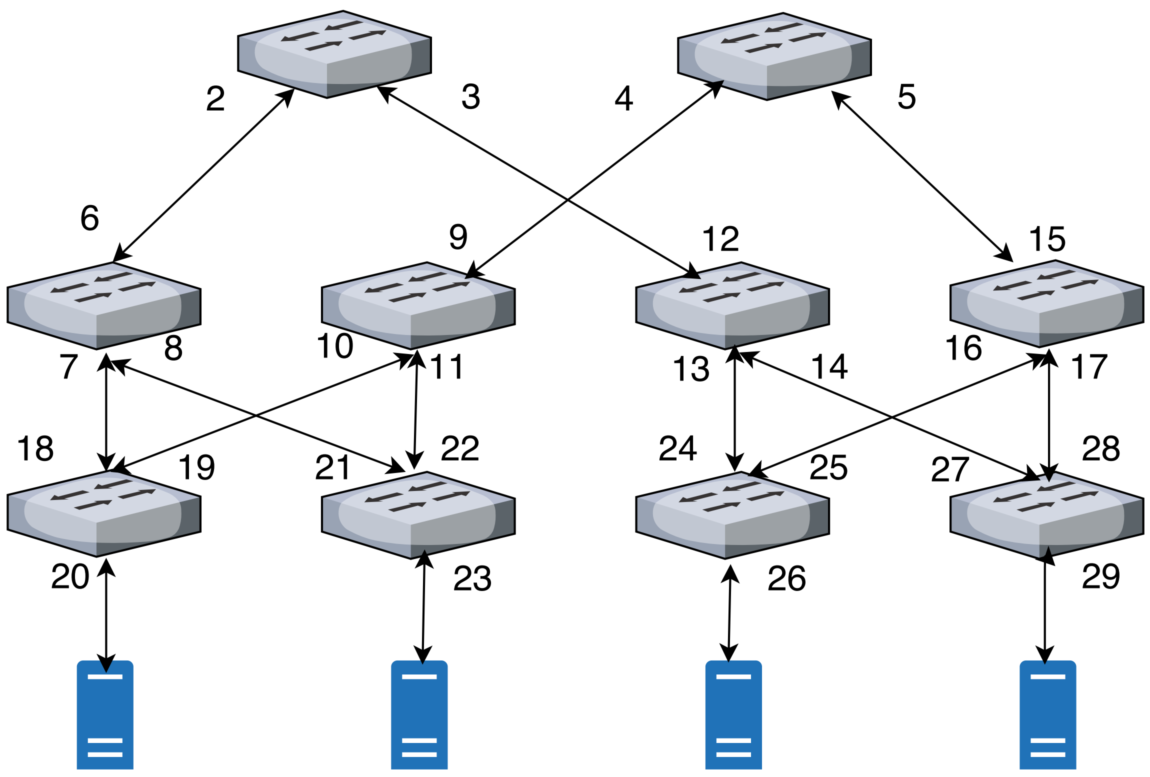

Figure 5.2.: A Fat-tree network topology

new programming language, a programmer must implement a set of high-level abstractions that applications use to receive Kafka events.

- *Performance optimization*: In the Kafka implementation, a programmer must tune parameters in Kafka brokers, producers, and consumers to optimize performance. Using gRPC allows applications access a remote procedure call mechanism that has a high performance potential, but it is not always clear how to optimize performance.

- *Loose Coupling*: Kafka and the publish-subscribe paradigm in general, provides a more loosely coupled communication system than gRPC and point-to-point messaging technologies. In Kafka's publish-subscribe paradigm, producers and consumers do not need to know the existence of other producers and consumers. In gRPC, however, a client needs to know about the services that the event distribution system provides. Using a loosely coupled event distribution system offers the advantages of scalability, resilience, and maintainability.

- *Fault Isolation and Code Base Disaggregation*: Disaggregating the control plane services and using an event distribution mechanism increase fault isolation and allow code base disaggregation regardless of the language used, by keeping components independent of one another.

5.3 Performance Evaluation of Kafka and gRPC Based Event Distribution Systems

We implemented prototypes of the two event distribution systems for the ONOS [4] SDN controller. To measure the two distribution mechanisms, we used the SDN testbed explained above.

5.3.1 Response Time Measurements

This experiment measures response time to evaluate the overhead of external event processing. Our goal is to compare the amount of time that an external app or service needs to process a packet event with the time it takes to process the same packet even inside the monolithic version of ONOS, and to understand the effect on overall response time. In the internal ONOS packet processing, whenever a host sends a ping request for which no forwarding rules have been established, each switch along the path sends the incoming packet to the controller core, which processes the packet internally and returns the packet to the switch to be forwarded. In the disaggregated architecture, whenever a host sends a ping request for which no forwarding rules have been established, each switch along the path sends the incoming packet to the controller core, and the event distribution system in the core (using either Kafka or gRPC) distributes the incoming packet to the set of external apps and services that have subscribed to receive packet events. The external process then uses gRPC to return the packet to the switch for forwarding. To focus measurements on the control plane overhead, we did not install flow rules. Thus, each packet causes a packet event at each switch along the path. We ran the experiment 500 times and measured the ping response time between two end hosts in our SDN testbed that are 5 hops apart. As the graph in Figure 5.3 shows, externalization of packet processing introduces overhead that increases the overall response time. As a baseline, we measured the average response time for internal processing in traditional (i.e., completely aggregated) ONOS as 24 ms. The average response time for a gRPC system is 29 ms, and the average time for a Kafka system is 35 ms. We conclude that using gRPC introduces less overhead than using Kafka based event distribution system.

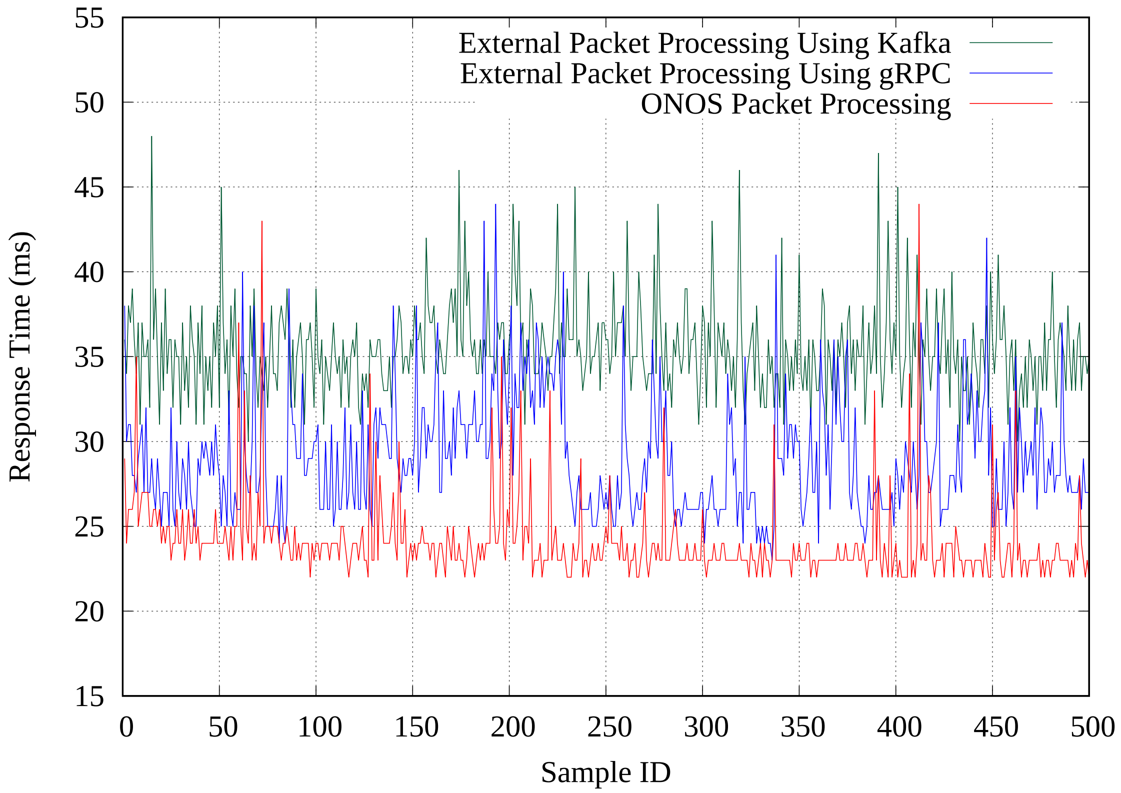

Figure 5.3.: Response times for external packet processing and ONOS internal packet processing

5.3.2 Throughput Measurements

To assess the impact of externalized packet processing and the use of a REST API [80] for flow rule installation on throughput, we compare external reactive forwarding applications that use gRPC and Kafka event distribution systems with an ONOS reactive forwarding application. In all three cases, we use a hard time-out of 10 seconds to remove flow rules (i.e.,the flow rules installed in a switch disappear every 10 seconds, which causes packet events and re-installation of the rules). We use the iperf3 tool to generate TCP [81] traffic, varying the the number of concurrent TCP connections, and running each measurement for 150 seconds. As the results in Figure 5.4 show, the effect of externalized packet processing on throughput is negligible. Furthermore, the overhead of using a REST API to install flow rules is larger than overhead introduced by externalization.

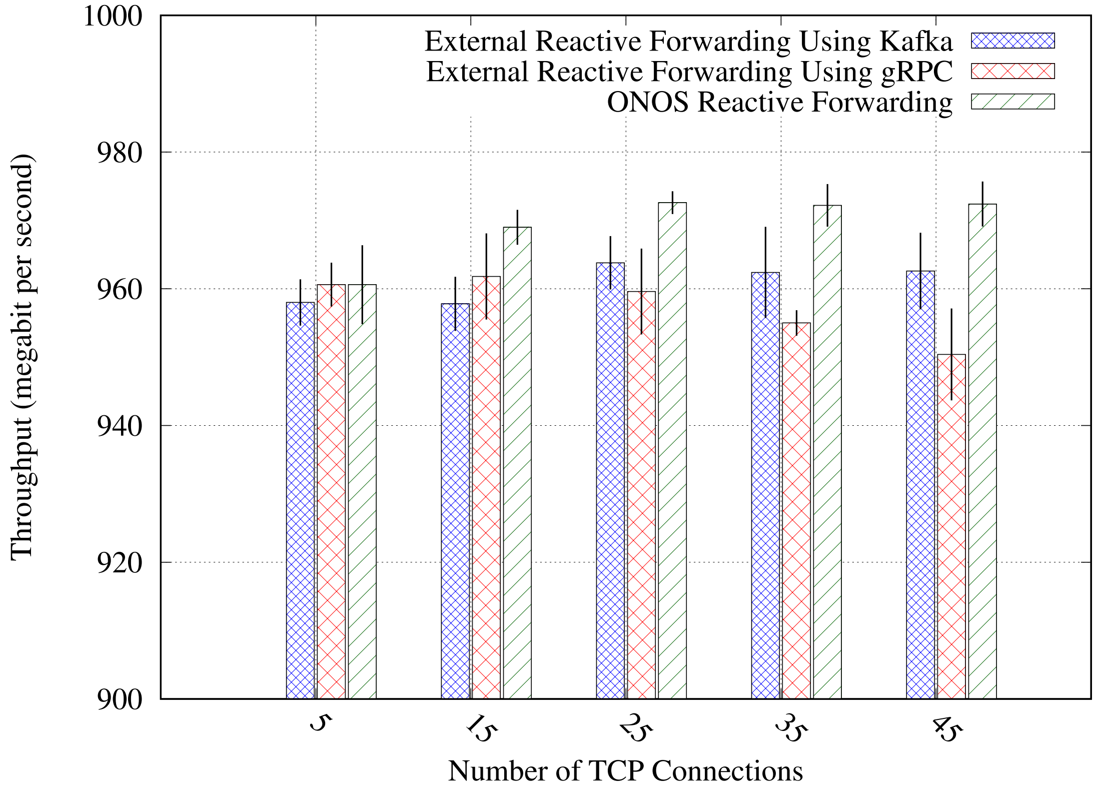

Figure 5.4.: Throughput vs. number of TCP connections for external and ONOS reactive forwarding applications

5.4 Functionality of the Umbrella Framework

As we mentioned earlier, programmers can use the Umbrella framework for writing of SDN management applications using its high-level programming abstractions. To show the functionality of the Umbrella APIs, we implemented two sample applications, including a forwarding and a firewall applications that we explain them briefly as follows:

5.4.1 An Example Forwarding Application

Suppose a programmers wants to write an application to route traffic between all of the hosts that belong to the same subnet. To achieve this goal, the programmer implements an application using Umbrella APIs to generate and install appropriate forwarding rules on each switch between two end points. The Umbrella framework requires three steps:

- First, create an instance of the controller that will be used to execute the application on. The programmers assumes that the name of controller is stored in a config file (e.g. config.properties) and the programmer uses it to initialize the controller.
- Second, obtain a list of current hosts that have been detected by the controller.
- Third, compute the shortest path between each pair of hosts to determine which network switches must be configured between the given hosts. The Umbrella flow abstractions can then be used to generate flow rules based on source and destination MAC addresses, source and destination IP addresses, and Ethernet type (IPv4 in this example), and install the generated flow rules on the network switches.

Appendix E presents a full implementation of the forwarding app using Umbrella APIs.

5.4.2 An Example Firewall Application

Consider the implementation of a basic firewall application that enforces the access control policy given in Table 5.1. Assume a network topology of 4 hosts, with all hosts on the same subnet.

Table 5.1.: Access Control Policy for Firewall Application

	H1 (10.0.0.1)	(H2) 10.0.0.2	(H3) 10.0.0.3	(H4) 10.0.0.4
H1(10.0.0.1)	NONE	DENY	WEB	DENY
H2(10.0.0.2)	DENY	NONE	DENY	ICMP
H3(10.0.0.3)	WEB	DENY	NONE	NONE
H4(10.0.0.4)	DENY	ICMP	NONE	NONE

To implement a firewall, programmers can create an application that uses the Umbrella APIs to generate and install appropriate forwarding rules on the switches between each pair of hosts: (h1,h3) and (h2, h4). Three steps are required:

- First, create an instance of the controller that will be used to execute our application. Programmers assume that the name of controller is stored in a config file (i.e. config.properties) and they use it to initialize the controller.
- Second, obtain the list of current hosts, and find a shortest path between each pair of hosts, (h1, h3) and (h2,h4).
- Third, generate appropriate match-action fields for each pair of hosts based on the type of traffic which is specified in the table 5.1. For the pair (h1, h3), the source and destination TCP ports must specify web traffic (i.e., port 80), and for the pair (h2, h4) the ICMP type and code fields must specify ICMP traffic (i.e. type=8 (echo request), type=0 (echo reply) and code=0). Note that for both types of traffic, the match fields also specify source and destination MAC addresses, source and destination IP addresses, Ethernet type, and IP protocol number as the match fields.

Appendix F contains a full implementation of the firewall application using Umbrella APIs.

5.5 Performance Evaluation of the Umbrella Framework

5.5.1 Experimental Setup and Results

Umbrella provides driver modules for the ONOS and ODL SDN controllers. We then used the Umbrella high-level API to write a controller-independent application that installs one-directional flow rules that forward traffic between a sender and a receiver. We ran a script that performs a total of 10 experiments using Mininet with linear topology of size 10, 20, 30... 100. Each experiment is performed 5 times and includes the following steps: 1) Create a Mininet instance to setup the topology with a specific size. 2) Have the sender transmit packets at the rate of 1 pkt/ms, and after 2 seconds, have our application install flow rules. 3) Arrange for the receiver (i.e. the destination host in the topology) to receive packets throughout the experiment and to report the number of packets received. Packet loss is then used to compute the flow rule setup time. Figure 5.5 illustrates the average flow rule setup time vs. total number of switches for both the OpenDayLight and ONOS controllers. As the results show, the flow rule setup time increases as the network span (i.e., number of switches between the two endpoints) increases. In addition, the simulation results show that although OpenDayLight outperforms ONOS in smaller network topologies, ONOS has a better performance for flow rule installation in larger network topologies. The Umbrella framework makes direct comparison of controller performance straightforward.

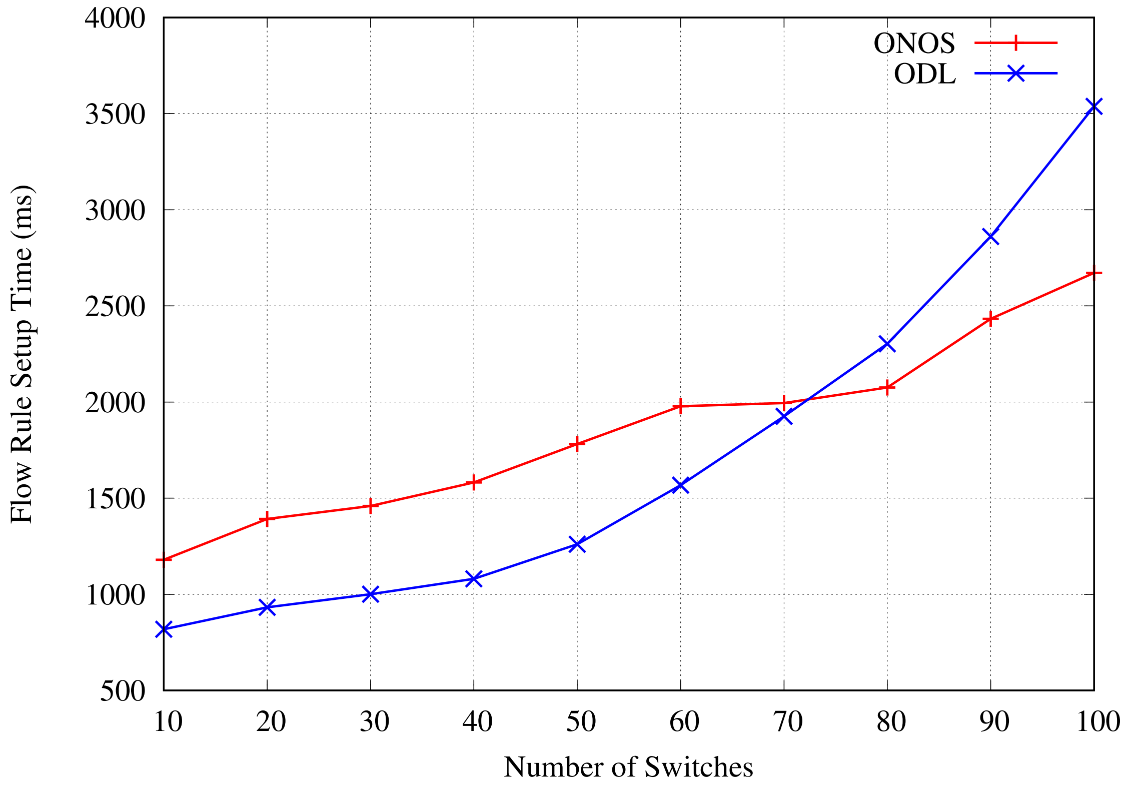

Figure 5.5.: Flow rule setup time

5.6 Assessment and Performance Evaluation of OSDF

5.6.1 Prototype and Experimental Setup

An early version of OSDF is implemented in the ONOS SDN controller, an open source SDN controller that, like other SDN controllers, provides services and APIs that programmers use to write applications and modules internal to ONOS. We use the ONOS APIs, such as the topology and flow rule APIs, to implement each network operation. Other modules, such as the policy parser, policy store, and policy conflict detection modules, have been implemented from scratch. To provide a broader evaluation of OSDF functionality and performance, we use both simulation and testbed measurements. That is, in addition to the SDN testbed described earlier, we used the *Mininet* [79] network emulator to evaluate our framework at a larger scale. We ran

Mininet on a Linux host with a Intel Core i7 processor and 8GB RAM. To generate various traffic patterns for the simulation, we used *iperf* [82], a tool for measuring the throughput of IP networks.

5.6.2 Simulation and Experimental Results

This section uses typical applications such as *Network Traffic Isolation*, an *Inter Domain Rate Limiter* to provide QoS, *Inbound Traffic Engineering*, and *Forwarding Resiliency*; OSDF allows each to be implemented easily.

1. ***Traffic Isolation***: Consider the leaf-spine topology that Figure 5.6 illustrates. We will call the configuration *site A*.

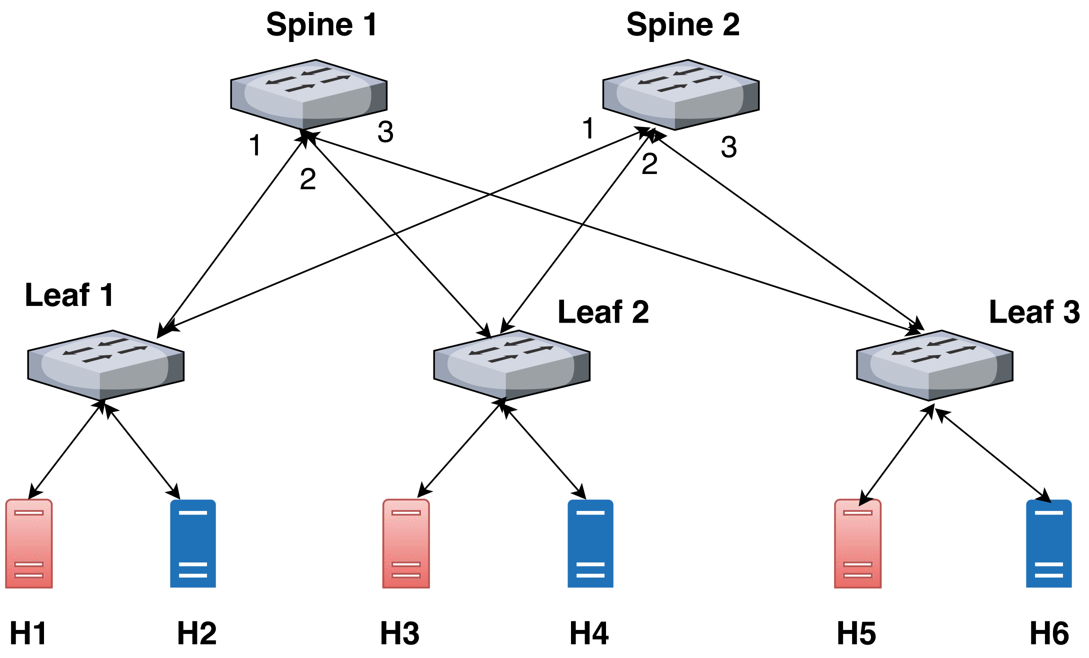

Figure 5.6.: Site A: a leaf-pine network topology used for traffic isolation experiments

Assume the network represents a multi-tenant network where hosts H1, H3, and H5 belong to tenant 1, and H2,H4, and H6 belong to tenant 2. We will configure the network using OSDF such that hosts owned by a given tenant are able to communicate with each other but are blocked from communicating with hosts owned by other tenants. Suppose we want to route web traffic for tenant 1, and route video streaming traffic for tenant 2. In addition, suppose all of the network switches in the given topology belong to the same region (site A). To achieve the goal, we define the following OSDF network policies:

- Route web traffic in site A between (H1, H3), (H1, H5), and (H3,H5) using the default priority.
- Route video traffic in site A between (H2, H4), (H2, H6), and (H4,H6) using the default priority.

The network administrator enters above policies into the system using a command line interface or a script, and OSDF installs OpenFlow rules in appropriate network switches to meet the requirements. As the example shows, instead of requiring managers to install flow rules, OSDF allows a manager to specify traffic isolation at the policy level.

2. ***Inter-site QoS Provisioning:*** The main goal of this scenario is to experiment with using the abstractions defined above to provision QoS for inter-site routing. OpenFlow 1.3 introduces meters which can be used to measure and control the ingress rate of traffic. If the packet rate or byte rate passing through the meter exceeds a predefined threshold, a meter band will be triggered. A *Rate Limiter* is a meter which drops packets when a meter band is reached. To achieve the goal, the underlying hardware must support metering. Because Open vSwitch does not support metering directly, we decided to test our QoS provisioning abstractions using our SDN testbed. Suppose we want to configure the network illustrated in Figure 5.1 with the following QoS requirements:

    - Route web traffic between IT and the sales department using the default priority, and limit each flow to 200 Mbps.

- Route video traffic between IT and the sales department using the default priority, and limit each flow to 500 Mbps.

We ran an experiment between the IT and sales department computers to test inter-site routing with rate limiting. In this scenario, we initiated four TCP connections (an aggregate of 800 Mbps) to generate web traffic and two TCP connections (an aggregate of 1000 Mbps) to generate video traffic. Figure 5.7 summarizes the results, and shows the throughput for each of the web traffic and video traffic flows is limited to 200 Mbps and 500 Mbps, respectively.

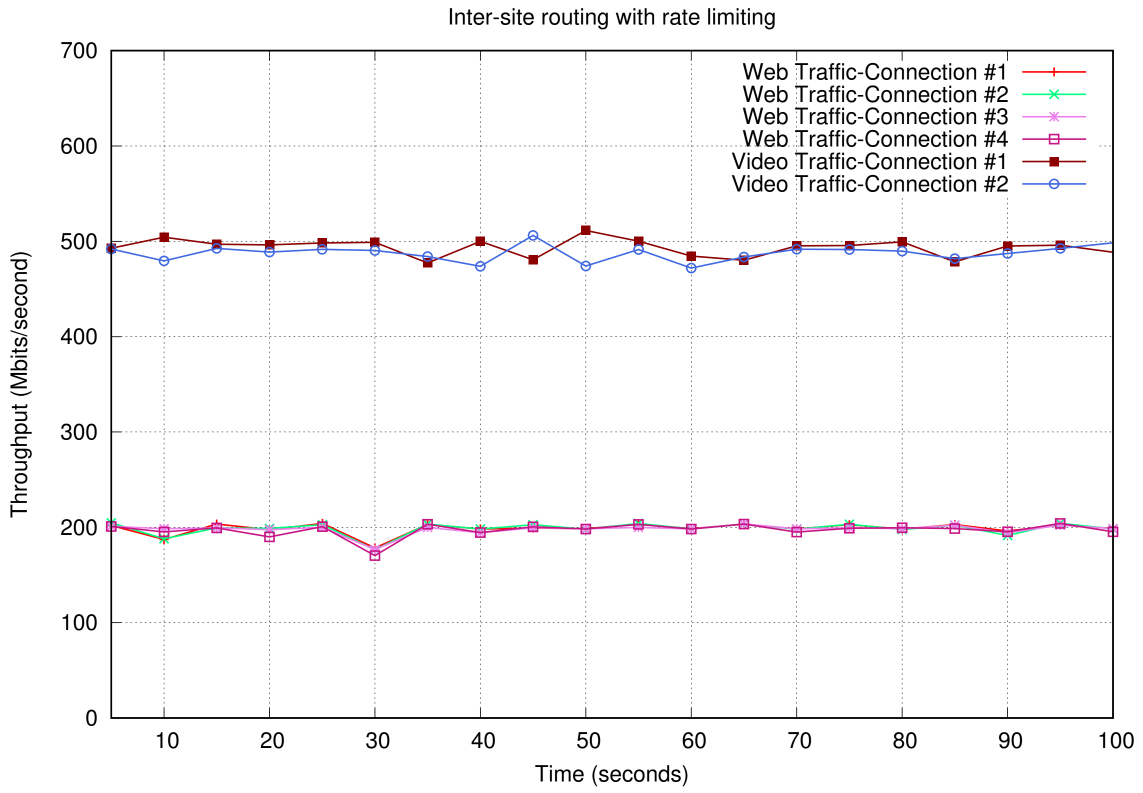

Figure 5.7.: Inter-site routing with rate limiting

To examine how well QoS provisioning works, we measured inter-site routing without QoS. To do so, we entered the same basic network policies for web and video traffic, omitting the rate limiting policies:

- Route web traffic between IT and sales departments using the default priority.
- Route video traffic between IT and sales departments using the default priority.

As the results in Figure 5.8 show, without rate limiting, video and web traffic are not guaranteed specific limits, and the throughput fluctuates more than when QoS is applied (i.e., more than in Fig.5.7).

Another proposed programming language, Merlin, addresses QoS provisioning by providing high level abstractions that can be used to express bandwidth constraints such as minimum and maximum bandwidth limits. Compared to our approach, Merlin shares the same weakness as other network programming languages (e.g., NetKAT) because a programmer must specify low-level packet match fields to define a policy.

3. ***Inbound Traffic Engineering***: This service refers to splitting incoming traffic over multiple peering links. Consider, for example, the internal representation of the IT and Sales departments in our SDN testbed, as illustrated in Figure 5.9. Suppose a network administrator wants to configure the two departments such that the incoming traffic to switch *Sales-S1* should be split among the outgoing links. That is, web traffic from the IT department should be forwarded to port 2 of switch Sales-S1, and video streaming traffic should be forwarded to port 3 of the same switch.

   To achieve the goal, we can define high-level policies as follows:

   - Route web traffic between IT and Sales departments via Sales-S1:2 using the default priority.
   - Route video traffic between IT and Sales departments via Sales-S1:3 using the default priority.

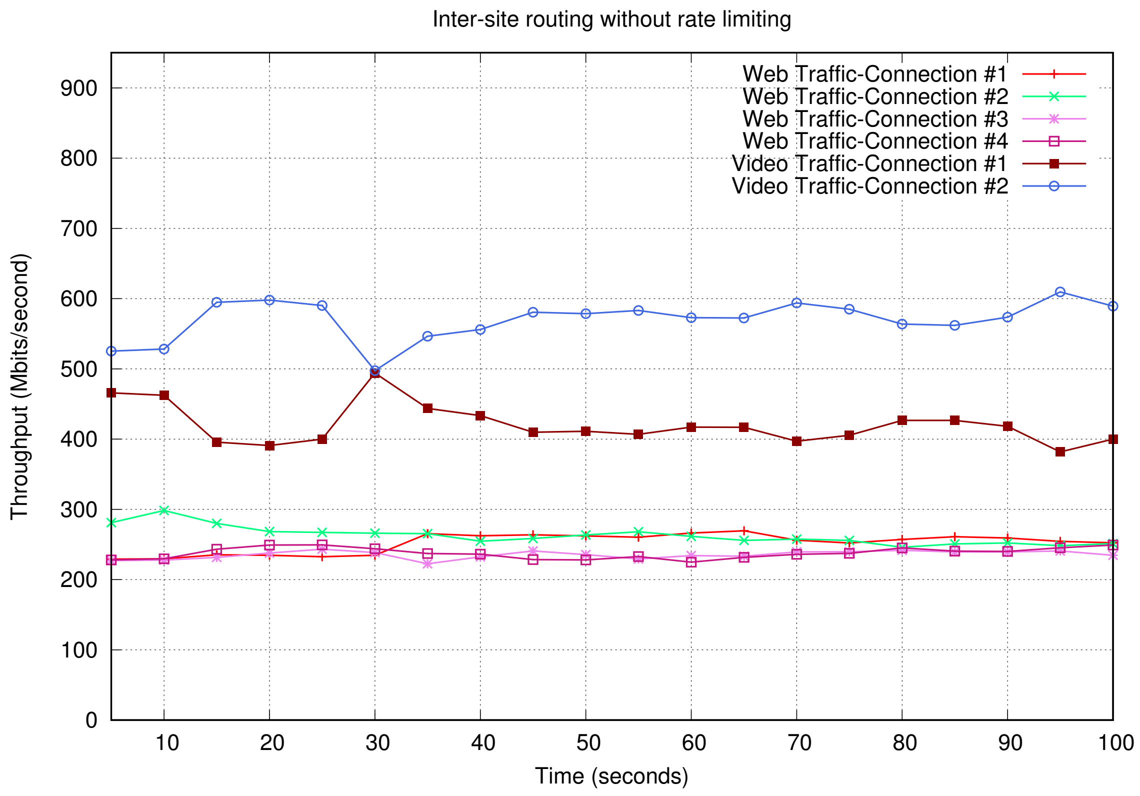

Figure 5.8.: Inter-site routing without rate limiting

4. ***Forwarding Resiliency***: Consider the network topology that Figure 5.10 illustrates; call it site C. We report an experiment that shows how OSDF can use redundant policies with differing priorities to handle the case of rerouting traffic (web traffic in the example) from a primary path to a backup path during failures or for maintenance purposes. In the first step, we define high-level policies for both a primary path and a backup path between two end-points:

   - Route web traffic in site C using priority 100 via S3:2.
   - Route web traffic in site C using priority 50 via S6:4.

   Based on the above policies, if host H1 initiates web traffic to host H2, the policy with higher priority will be triggered, and OpenFlow rules with priority

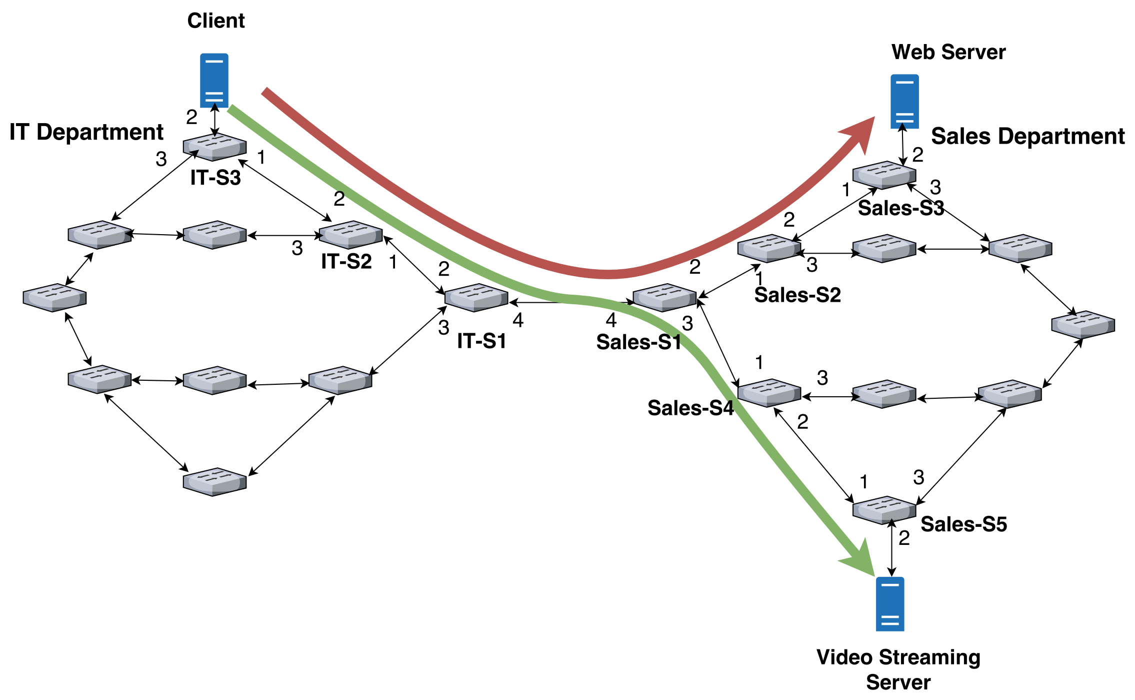

Figure 5.9.: Inbound traffic engineering scenario

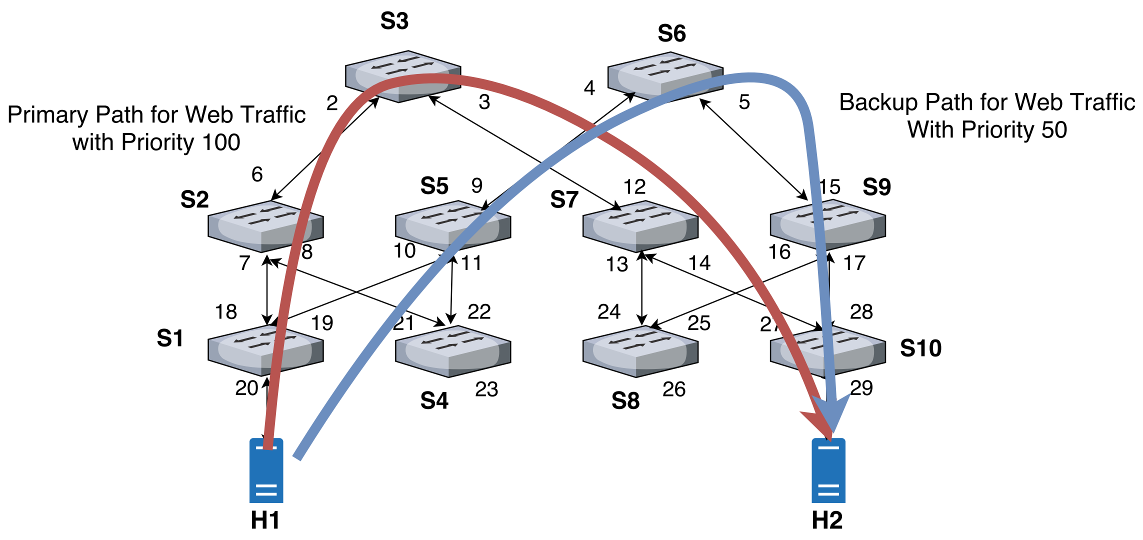
S3
S6
Primary Path for Web Traffic with Priority 100
Backup Path for Web Traffic With Priority 50
S2
S5
S7
S9
S1
S4
S8
S10
H1
H2

Figure 5.10.: A multipath network topology (site C)

100 will be added to network switches along the primary path (i.e. H1, S1, S2, S3, S7, S10, H2).

To show how OSDF reacts to adding and removing policies, we wrote a script that forces a change from the primary to a backup path by removing and adding the policies every N (Interval) seconds in simulation of 150 seconds. One aspect of any networking system arises from the need to re-establish valid routes after a failure with minimal interference. To demonstrate that our framework supports rerouting without negative impact on traffic, we conducted an experiment to compare the throughput of a flow in the presence of rerouting to the throughput of the same flow with no rerouting. To conduct the experiment, we ran iperf as a server using port 80 on H2, and ran iperf as a client on H1 with the default Linux configuration. We repeated the experiment for various numbers of parallel connections. As the results in Figure 5.11 show, when switching between primary path and backup path is not frequent (e.g the interval is 40 seconds) the throughput remains close to the maximum possible throughput, which is around 940 Mbps (achieved when just the primary path is used). When we change between the primary and backup paths frequently (e.g., every 2 seconds) the throughput drops significantly, specially when multiple, parallel TCP connections (e.g., 50). Multiple factors explain the decrease. First, as the number of connections increases from 1 to 50, the setup time (i.e., time required to install OpenFlow rules) will increase because the time is linearly proportional to number of rules. Second, switching paths introduces delay, which drives TCP into congestion avoidance. Frequent changes means lower throughput because TCP will spend more time recovering, and less time in a stable state.

5.6.3 Simulation Scenarios

To test our framework from functional and performance points of view and to examine the effects of scale, we ran simulation scenarios.

- ***Response time***: The simulation uses a worst-case linear network topology with 40 switches that we call site D. Figure 5.12 illustrates the topol-

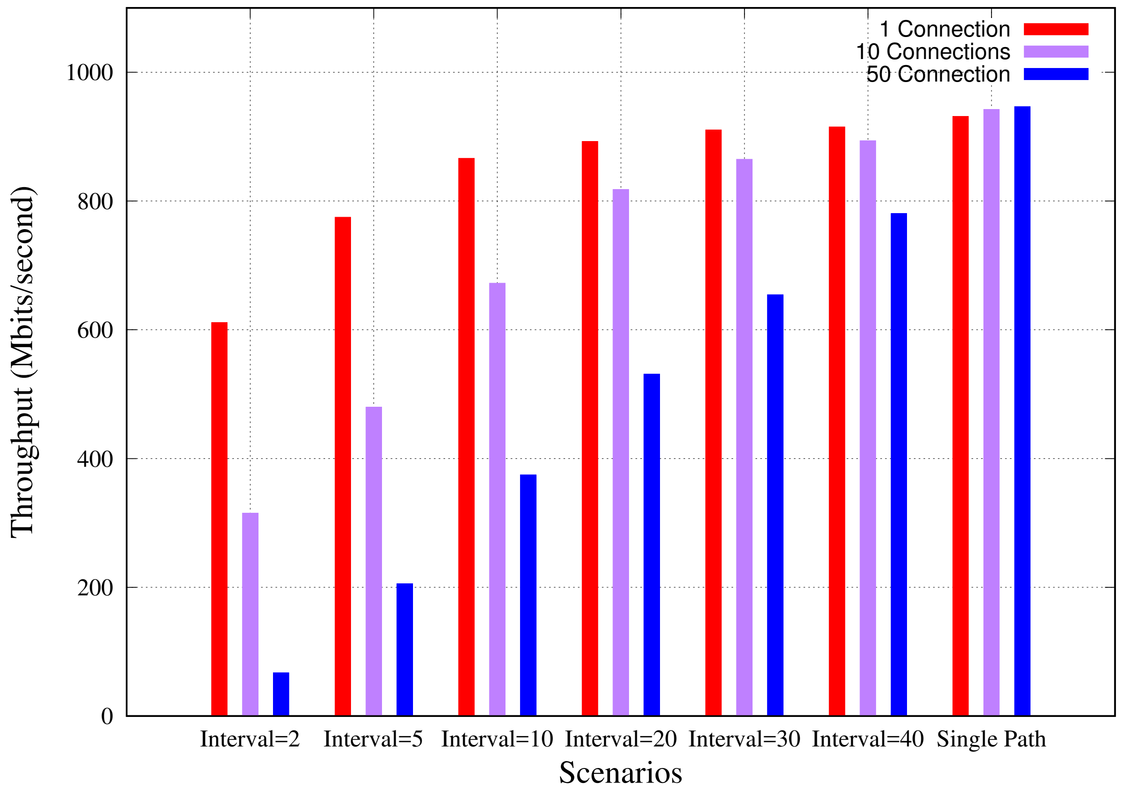

Figure 5.11.: Experimental results for forwarding resiliency scenarios

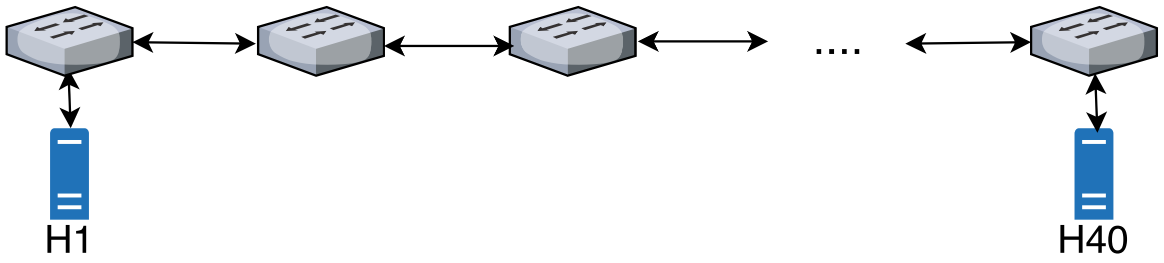

Figure 5.12.: Site D: A worst-case linear network topology

ogy. To configure the network to route web traffic between H1 and H40, we use a high-level policy:

– Route web traffic in site D using the default priority.

To evaluate setup time, we initiate various numbers of TCP connections on port 80 between H1 and H40, and measure the response time for each.

We define the response time as the amount of time needed to establish the entire set of the connections between the two end-points. The time includes the amount of time needed to read and parse a network policy, generate flow rules, and install generated rules on the network devices for all of the connections between the two end-points. By increasing the number of TCP connections established between two end-points, we increase the number of OpenFlow rules that must be installed in network switches along the path. Consequently, we measured the response time as a function of number of flows and compared the response time of OSDF with reactive forwarding approach implemented by ONOS, as the results in Figure 5.13 illustrate. As the results show, the response time increases linearly as number of connections increase between the two end-points. The reason that our system substantially outperforms the ONOS reactive approach arises from an optimization in which our system pre-installs OpenFlow rules in all switches across the entire end-to-end path when the first *PACKET_IN* message arrives at a controller. The ONOS reactive forwarding approach waits until a *PACKET_IN* message arrives from a switch before installing a forwarding rule in the switch.

To validate the simulation results, we ran the same experiment on our testbed; Figure 5.14 shows the results. The experimental measurement confirms that the simulation results are valid. Note that the TCAM table on network switches in our testbed can only hold 2K entries when more than 2 match fields are used by flow rules. Therefore, the number of connections measured across the testbed is limited by the hardware.

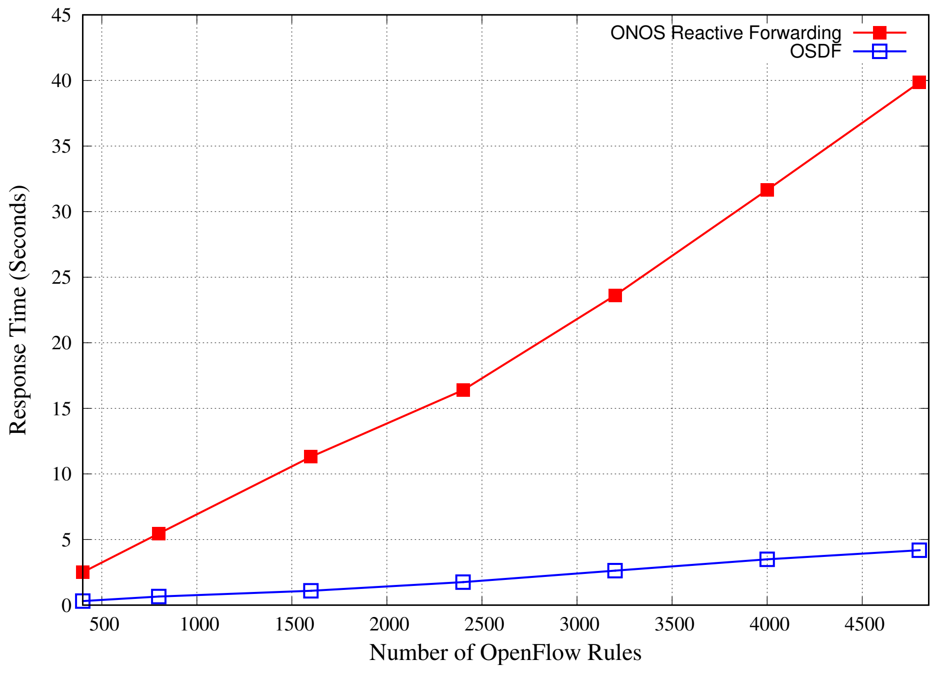

Figure 5.13.: Response time: ONOS reactive forwarding vs. OSDF for a linear network topology (the simulation scenario)

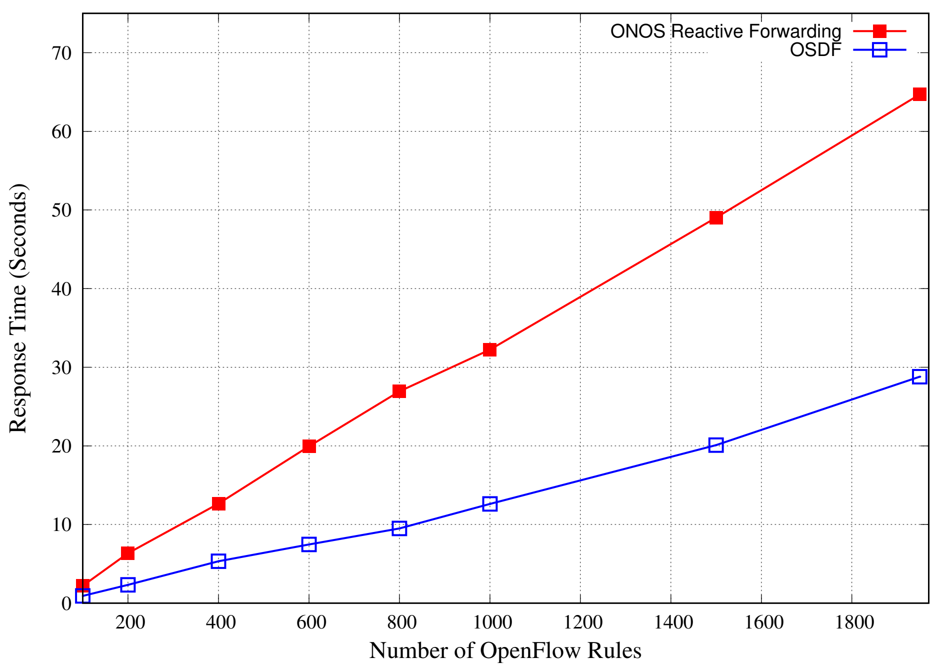

Figure 5.14.: Response time: ONOS reactive forwarding vs. OSDF for the testbed

6 FUTURE WORK

Migrating SDN controllers from a monolithic architecture to a microservice based architecture can potentially open new areas of research. The following provides a brief list:

- This dissertation proposes a new architecture for the next generation of SDN controllers based on a microservice architecture. However, designing and implementing SDN controller subsystems for the new architecture will require further investigation. Specifically, topology, control, and configuration subsystems must be studied. The following explains some of the research questions:
  - As we discussed earlier, each subsystem provides a set of interservice and NB APIs to communicate with other subsystems and applications. One of the critical question that researchers should answer is how such APIs should be designed? Should we design these APIs using a single design pattern to allow easy porting from one subsystem to another subsystem? For example, does employing gRPC for communication between all apps and services allow the use of a single set of generic methods that can then be customized to define the new protobuf messages that are required for each subsystem? What are the tradeoffs of using generic methods instead of using more specific methods that tailor the functionality to each specific subsystem?
  - The design and implementation of SDN management applications must be evaluated for both functionality and performance in a microservice architecture. For example, if a controller adopts gRPC for the communication between microservices, what is the impact of the communication overhead on network performance (e.g., on latency)?

- In a microservice architecture, the codebase for each subsystem is organized in a separate repository which makes the development of each subsystem independent of other subsystems but it does not mean the subsystems are not dependent to each other in terms of functionality. Thus, to have a fully functional SDN controller, all of the required subsystems and applications need to be deployed in a cluster that can communicate with each other to provide the required functionalities. One of the potential solution that programmers can use to collectively facilitate reliable, secure communications between microservices is service mesh. A service mesh acts as a proxy that provides critical services like load balancing, service discovery, authentication, operational monitoring, etc. Despite the fact that a service mesh has advantages but it also has some drawbacks. For example, using a service mesh increases the latency of service to service communication as each call out to another service must go through the proxy and load balancer. Studying the use of a service mesh for a microservice based SDN controller is an open research problem that should be addressed to answer the following questions: How does a service mesh affect on performance of a disaggregated SDN controller? Does a service mesh increase the complexity of a disaggregated SDN controller? What are the alternatives that developers can use in the context of SDN to achieve the goals and avoid increasing delay for performing network operations?

- As discussed earlier, the implementation of OSDF reported in this dissertation consists of an internal application for the ONOS SDN controller, and it uses the Java programming language. However, migrating to a disaggregated control plane will allow OSDF or similar intent-based SDN programming frameworks to be implemented outside of the controller core using an arbitrary programming language. The question is how should an intent-based SDN framework be designed to be agnostic of the programming language?

- Externalization of event processing, especially packet events, has the potential to overwhelm clients that consume the events. In many cases, a given client only needs to consume a subset of the events. For example, a topology subsystem that is responsible for building a network topology only needs to process LLDP and ARP packet events. The question arises: how and where should event filtering be done to limit the events that a client receives? Some potential options include:
  - *Client Side Filtering*: In the client side filtering, all of the events will be sent to the external apps and services and application and services are responsible to filter events according to their requirements.
  - *Server Side Filtering*: In the server side filtering, the event distribution system filters events based on type before publishing them on a broker or sending them directly to the application and services using a point-to-point channel. In addition, in publish-subscribe model, programmers can equip a broker to filter events based on type.
- External apps use NB interfaces, such as the REST API that current SDN controllers provide to the programmers for installing, removing, and updating flow rules. A REST API does not provide an efficient approach for installing of flow rules in external reactive-based applications because it introduces a notable delay. Replacing the REST APIs with a new NB interface, such as a gRPC-based NB interface, can be considered as a potential solution to speed up flow rule installation in external applications. The proposed architecture opens new areas of research in designing of new NB interfaces for SDN controllers to meet new requirements.

7 SUMMARY AND CONCLUSION

A monolithic architecture for an SDN controller aggregates all control plane subsystems into a single, gigantic program. An SDN controller that adopts the monolithic approach restricts programmers who write management applications to the programming interfaces and services that the controller provides. In this dissertation, we propose a distributed architecture that disaggregates controller software into a small controller core and a set of cooperative microservices to overcome the limitations inherent in the monolithic architecture. Disaggregation allows a programmer to choose a programming language that is appropriate for each microservice. We explain the steps that are required to migrate from a monolithic architecture to a microservice architecture. One of the key steps in migrating from a monolithic architecture to a microservice based SDN controller requires devising a mechanism to distribute network events to external processes. To address the problem, we consider two event distribution mechanisms based on two generic messaging paradigms: publish-subscribe and point-to-point models. We discuss the advantages and disadvantages of the two event distribution systems. In addition, we use two popular technologies, gRPC and Apache Kafka, to devise prototypes of the event distribution systems and evaluate their functionality and performance. Our experimental results show that externalizing event processing introduces some overhead, and the overhead resulting from gRPC is lower than the overhead resulting from Kafka. Externalizing event processing has a negligible effect on throughput, and the cost is considered small when compared with the advantages of portability, flexibility, and support for multilanguage management applications.

In addition, the dissertation presents a unified software defined network programming framework called Umbrella that can be used to implement SDN applications independent of NB APIs that different SDN controllers provide. Umbrella employs the event distribution systems that we proposed as part of the disaggreagated control plane architecture, and includes support for both reactive and proactive applications. Umbrella increases the portability of SDN applications across SDN controllers and makes it easy for the developers to evaluate a single applications on multiple SDN controllers. In addition, Umbrella reduces the programming complexity by providing high-level programming abstractions that allows a programmer to write SDN applications without requiring the programmer to master low level details of each SDN controller. The dissertation explains the Umbrella APIs and illustrates their use with two SDN applications: a firewall and forwarding. Furthermore, the dissertation reports an evaluation of the performance of SDN applications in terms of flow rule setup time on two SDN controllers; ONOS and OpenDayLight.

Finally, the dissertation presents an SDN-based network programming framework, OSDF. OSDF provides a high-level API that can be used by managers and network administrators to express network requirements for applications and policies for multiple domains. OSDF also provides a set of high-level network operation services that handle common network configuration, monitoring, and QoS provisioning. OSDF is equipped with a policy conflict management module to help a network administrator detect and resolve policy conflicts. The dissertation explains the OSDF policy specification, and implements its use to implement typical SDN management applications, such as network traffic isolation, inbound traffic engineering, intra and inter domain QoS provisioning, and forwarding resiliency. The dissertation also evaluates the OSDF framework in terms of functionality and performance using simulation and experimental results.

REFERENCES

[1] D. Kreutz, F. M. V. Ramos, P. E. Verssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, Jan 2015.

[2] Open network operating system, 2019. `http://onosproject.org/`.

[3] Opendaylight sdn controller, 2019. `https://opendaylight.org/`.

[4] 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.

[5] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[6] Open Networking Foundation (ONF). OpenFlow Switch Specification: Version 1.4, 2015. `https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.4.1.pdf`.

[7] B. Pfaff and B. Davie. The open vswitch database management protocol. RFC 7047, RFC Editor, December 2013. `http://www.rfc-editor.org/rfc/rfc7047.txt`.

[8] A. Doria, J. Hadi Salim, R. Haas, H. Khosravi, W. Wang, L. Dong, R. Gopal, and J. Halpern. Forwarding and control element separation (forces) protocol specification. RFC 5810, RFC Editor, March 2010. `http://www.rfc-editor.org/rfc/rfc5810.txt`.

[9] Haoyu Song. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 127–132, New York, NY, USA, 2013. ACM.

[10] Giuseppe Bianchi, Marco Bonola, Antonio Capone, and Carmelo Cascone. Openstate: Programming platform-independent stateful openflow applications inside the switch. *SIGCOMM Comput. Commun. Rev.*, 44(2):44–51, April 2014.

[11] M. Su, V. Alvarez, T. Jungel, U. Toseef, and K. Pentikousis. An openflow implementation for network processors. In *2014 Third European Workshop on Software Defined Networks*, pages 123–124, Sep. 2014.

[12] B. Belter, D. Parniewicz, L. Ogrodowczyk, A. Binczewski, M. Stroiski, V. Fuentes, J. Matias, M. Huarte, and E. Jacob. Hardware abstraction layer as an sdn-enabler for non-openflow network equipment. In *2014 Third European Workshop on Software Defined Networks*, pages 117–118, Sep. 2014.

[13] B. Belter, A. Binczewski, K. Dombek, A. Juszczyk, L. Ogrodowczyk, D. Parniewicz, M. Stroiski, and I. Olszewski. Programmable abstraction of datapath. In *2014 Third European Workshop on Software Defined Networks*, pages 7–12, Sep. 2014.

[14] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[15] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4fpga: A rapid prototyping framework for p4. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 122–135, New York, NY, USA, 2017. ACM.

[16] P4 specification, 2019. `https://p4.org/p4-spec/`.

[17] Netconf: Network configuration protocol, 2019. `https://tools.ietf.org/html/rfc4741`.

[18] Openconfig, 2019. `http://openconfig.net/`.

[19] grpc network management interface (gnmi) specification. `https://github.com/openconfig/reference/blob/master/rpc/gnmi/gnmi-specification.md`.

[20] F. Bannour, S. Souihi, and A. Mellouk. Distributed sdn control: Survey, taxonomy, and challenges. *IEEE Communications Surveys Tutorials*, 20(1):333–354, Firstquarter 2018.

[21] David Erickson. The beacon openflow controller. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 13–18, New York, NY, USA, 2013. ACM.

[22] Ryu sdn framework, 2019. `https://osrg.github.io/ryu/`.

[23] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, and Rob Sherwood. On controller performance in software-defined networks. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*, Hot-ICE'12, pages 10–10, Berkeley, CA, USA, 2012. USENIX Association.

[24] Floodlight project, 2019. `http://projectfloodlight.org`.

[25] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A distributed control platform for large-scale production networks. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 351–364, Berkeley, CA, USA, 2010. USENIX Association.

[26] Amin Tootoonchian and Yashar Ganjali. Hyperflow: A distributed control plane for openflow. In *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[27] 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.

[28] Soheil Hassas Yeganeh and Yashar Ganjali. Kandoo: A framework for efficient and scalable offloading of control applications. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 19–24, New York, NY, USA, 2012. ACM.

[29] 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.

[30] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, Victor Lin, Colin Rice, Brian Rogan, Arjun Singh, Bert Tanaka, Manish Verma, Puneet Sood, Mukarram Tariq, Matt Tierney, Dzevad Trumic, Vytautas Valancius, Calvin Ying, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 432–445, New York, NY, USA, 2017. ACM.

[31] C. Trois, M. D. Del Fabro, L. C. E. de Bona, and M. Martinello. A survey on sdn programming languages: Toward a taxonomy. *IEEE Communications Surveys Tutorials*, 18(4):2687–2712, Fourthquarter 2016.

[32] Nate Foster, Michael J. Freedman, Rob Harrison, Jennifer Rexford, Matthew L. Meola, and David Walker. Frenetic: A high-level language for openflow networks. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '10, pages 6:1–6:6, New York, NY, USA, 2010. ACM.

[33] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. Modular SDN Programming with Pyretic. *USENIX ;login:*, 38:40–47, 2013.

[34] Robert Soulé, Shrutarshi Basu, Robert Kleinberg, Emin Gün Sirer, and Nate Foster. Managing the network with merlin. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 24:1–24:7, New York, NY, USA, 2013. ACM.

[35] Andreas Voellmy, Hyojoon Kim, and Nick Feamster. Procera: A language for high-level reactive network control. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 43–48, New York, NY, USA, 2012. ACM.

[36] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 59–72, Oakland, CA, 2015. USENIX Association.

[37] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying sdn programming using algorithmic policies. *SIGCOMM Comput. Commun. Rev.*, 43(4):87–98, August 2013.

[38] Andreas Voellmy and Paul Hudak. Nettle: Taking the sting out of programming network routers. In *Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages*, PADL'11, pages 235–249, Berlin, Heidelberg, 2011. Springer-Verlag.

[39] Bruno Batista and Marcial Fernandez. Ponderflow: A new policy specification language to sdn openflow-based networks. *International Journal On Advances in Networks and Services*, 7(3 and 4):163–172, dec 2014.

[40] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, POLICY '01, pages 18–38, London, UK, UK, 2001. Springer-Verlag.

[41] Z. Shu, J. Wan, J. Lin, S. Wang, D. Li, S. Rho, and C. Yang. Traffic engineering in software-defined networking: Measurement and management. *IEEE Access*, 4:3246–3256, 2016.

[42] Brandon Heller, Srini Seetharaman, Priya Mahadevan, Yiannis Yiakoumis, Puneet Sharma, Sujata Banerjee, and Nick McKeown. Elastictree: Saving energy in data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association.

[43] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic flow scheduling for data center networks. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[44] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. Microte: Fine grained traffic engineering for data centers. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 8:1–8:12, New York, NY, USA, 2011. ACM.

[45] 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.

[46] K. Jeong, J. Kim, and Y. Kim. Qos-aware network operating system for software defined networking with generalized openflows. In *2012 IEEE Network Operations and Management Symposium*, pages 1167–1174, April 2012.

[47] Michael Scharf, Vijay Gurbani, Thomas Voith, Manuel Stein, W. Roome, Greg Soprovich, and Volker Hilt. Dynamic vpn optimization by alto guidance. In *Proceedings of the 2013 Second European Workshop on Software Defined Networks*, EWSDN '13, pages 13–18, Washington, DC, USA, 2013. IEEE Computer Society.

[48] M. Said Seddiki, Muhammad Shahbaz, Sean Donovan, Sarthak Grover, Miseon Park, Nick Feamster, and Ye-Qiong Song. Flowqos: Qos for the rest of us. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 207–208, New York, NY, USA, 2014. ACM.

[49] Julius Schulz-Zander, Lalith Suresh, Nadi Sarrar, Anja Feldmann, Thomas Hühn, and Ruben Merz. Programmatic orchestration of wifi networks. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 347–358, Berkeley, CA, USA, 2014. USENIX Association.

[50] Manu Bansal, Jeffrey Mehlman, Sachin Katti, and Philip Levis. Openradio: A programmable wireless dataplane. In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 109–114, New York, NY, USA, 2012. ACM.

[51] Amin Tootoonchian, Monia Ghobadi, and Yashar Ganjali. Opentm: Traffic matrix estimator for openflow networks. In *Proceedings of the 11th International Conference on Passive and Active Measurement*, PAM'10, pages 201–210, Berlin, Heidelberg, 2010. Springer-Verlag.

[52] S. R. Chowdhury, M. F. Bari, R. Ahmed, and R. Boutaba. Payless: A low cost network monitoring framework for software defined networks. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–9, May 2014.

[53] Junho Suh, Ted Taekyoung Kwon, Colin Dixon, Wes Felter, and John Carter. Opensample: A low-latency, sampling-based measurement platform for commodity sdn. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ICDCS '14, pages 228–237, Washington, DC, USA, 2014. IEEE Computer Society.

[54] Syed Akbar Mehdi, Junaid Khalid, and Syed Ali Khayam. Revisiting traffic anomaly detection using software defined networking. In *Proceedings of the 14th International Conference on Recent Advances in Intrusion Detection*, RAID'11, pages 161–180, Berlin, Heidelberg, 2011. Springer-Verlag.

[55] Rodrigo Braga, Braga, Edjard Mota, Mota, and Alexandre Passito, Passito. Lightweight ddos flooding attack detection using nox/openflow. In *Proceedings of the 2010 IEEE 35th Conference on Local Computer Networks*, LCN '10, pages 408–415, Washington, DC, USA, 2010. IEEE Computer Society.

[56] Eric Keller, Soudeh Ghorbani, Matt Caesar, and Jennifer Rexford. Live migration of an entire network (and its hosts). In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 109–114, New York, NY, USA, 2012. ACM.

[57] A. Arefin, V. K. Singh, G. Jiang, Y. Zhang, and C. Lumezanu. Diagnosing data center behavior flow by flow. In *2013 IEEE 33rd International Conference on Distributed Computing Systems*, pages 11–20, July 2013.

[58] Enterprise integration patterns, 2019. `https://enterpriseintegrationpatterns.com/`.

[59] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[60] Docker, 2019. `https://docker.com/`.

[61] Kubernetes (k8s): Production-grade container orchestration, 2019. `https://kubernetes.io`.

[62] D. Comer and A. Rastegarnia. Externalization of packet processing in software defined networking. *IEEE Networking Letters*, pages 1–1, 2019.

[63] Douglas Comer, Rajas H. Karandikar, and Adib Rastegarnia. Umbrella: A unified software defined development framework. In *Proceedings of the 2018 Symposium on Architectures for Networking and Communications Systems*, ANCS '18, pages 148–150, New York, NY, USA, 2018. ACM.

[64] W. Cerroni, C. Buratti, S. Cerboni, G. Davoli, C. Contoli, F. Foresta, F. Callegati, and R. Verdone. Intent-based management and orchestration of heterogeneous openflow/iot sdn domains. In *2017 IEEE Conference on Network Softwarization (NetSoft)*, pages 1–9, July 2017.

[65] D. Comer and A. Rastegatnia. Osdf: An intent-based software defined network programming framework. In *2018 IEEE 43rd Conference on Local Computer Networks (LCN)*, pages 527–535, Oct 2018.

[66] D. Comer and A. Rastegarnia. Osdf: A framework for software defined network programming. In *2018 15th IEEE Annual Consumer Communications Networking Conference (CCNC)*, pages 1–4, Jan 2018.

[67] S. Pisharody, J. Natarajan, A. Chowdhary, A. Alshalan, and D. Huang. Brew: A security policy analysis framework for distributed sdn-based cloud environments. *IEEE Transactions on Dependable and Secure Computing*, PP(99):1–1, 2017.

[68] Ehab S Al-Shaer and Hazem H Hamed. *Firewall Policy Advisor for Anomaly Discovery and Rule Editing.* Springer US, Boston, MA, 2003.

[69] Apache kafka: A distributed streaming platform, 2019. `https://kafka.apache.org/`.

[70] grpc: A high-performance, open-source universal rpc framework, 2019. `https://grpc.io`.

[71] Http2: Hypertext transfer protocol version 2, 2019. `https://http2.github.io/`.

[72] Protocol buffers, 2019. `https://developers.google.com/protocol-buffers/`.

[73] Golang, 2019. `https://golang.org/`.

[74] Python programming langauge, 2019. `https://www.python.org/`.

[75] Extensible markup language (xml), 2019. `https://w3.org/XML/`.

[76] Umbrella project, 2019. `https://github.com/umbrella-project/umbrella`.

[77] Osdf application for onos sdn controller, 2019. `https://github.com/OpenSDF/OSDF-REACTIVE-APP`.

[78] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, pages 63–74, New York, NY, USA, 2008. ACM.

[79] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[80] Leonard Richardson, Mike Amundsen, and Sam Ruby. *RESTful Web APIs*. O'Reilly Media, Inc., 2013.

[81] Jon Postel. Transmission control protocol, 1981. `https://tools.ietf.org/html/rfc793`.

[82] iperf - the ultimate speed test tool for tcp, udp and sctp, 2019. `https://iperf.fr`.

A PROTOBUF MESSAGES AND SERVICES FOR THE GRPC BASED EVENT DISTRIBUTION SYSTEM

```
syntax = "proto3";

// Corresponds to Registration Request protobuf message
message RegistrationRequest {
    string clientId = 1;
}
// Corresponds to Registration Response protobuf message
message RegistrationResponse {
    string clientId = 1;
    string serverId = 2;
}
// An enumeration for different types of events
enum topicType {
    PACKET_EVENT = 0;
    LINK_EVENT = 1;
}
// Corresponds to Topic protobuf message
message Topic {
    string clientId = 1;
    topicType type = 2;
}
// Corresponds to Notification protobuf message
message Notification {
    string clientId = 1;
    string serverId = 2;
    topicType type = 3;
    oneof event {
        net.packet.PacketContextProto packetContext = 4;
        net.link.LinkNotificationProto linkEvent = 5;

    }
}

// Corresponds to EventNotification service
service EventNotification {
    rpc register (RegistrationRequest) returns (RegistrationResponse);
    rpc onEvent (Topic) returns (stream Notification);

}
```

B UMBRELLA FLOW SERVICE API USING JAVA PROGRAMMING LANGUAGE

```
/*
 * Copyright ${2018} [Adib Rastegarnia, Douglas Comer]
 *
 *    Licensed under the Apache License, Version 2.0 (the "License");
 *    you may not use this file except in compliance with the License.
 *    You may obtain a copy of the License at
 *
 *        http://www.apache.org/licenses/LICENSE-2.0
 *
 *    Unless required by applicable law or agreed to in writing, software
 *    distributed under the License is distributed on an "AS IS" BASIS,
 *    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *    See the License for the specific language governing permissions and
 *    limitations under the License.
 */

package api.flowservice;

import java.util.List;

/**
 * Represents a flow including a list of match fields and  actions.
 */
public class Flow implements FlowInterface {


    /**
     * An object of match fields.
     */
    protected FlowMatch flowMatch;
    /**
     * A List of flow actions.
     */
    protected List<FlowAction> flowActions;

    /**
     * Priority of a flow.
     */
    protected Integer priority;

    /**
     * Flow rule table ID.
     */
    protected Integer tableID;
    /**
     * Flow rule device ID.
     */
    protected String deviceID;

    /**
     * Flow rule time out.
     */
    protected Integer timeOut;
    /**
     *
     */
    protected boolean isPermanent;
    /**
     * Flow rule cookie.
     */
    protected Integer cookie;
    /**
     * An application ID for a flow rule.
     */
    protected String appId;
    /**
     * flow rule ID.
     */
    protected String flowID;

    /**
     * Default constructor
     */
    protected Flow() {
    }

    /**
     * Constructor based on a builder
     *
     * @param builder : Builder object to initialize the flow
     */
    protected Flow(Builder builder) {
        this.deviceID = builder.deviceID;
        this.flowMatch = builder.flowMatch;
        this.flowActions = builder.flowActions;
        this.priority = builder.priority;
        this.tableID = builder.tableID;
        this.timeOut = builder.timeOut;
        this.isPermanent = builder.isPermanent;
        this.cookie = builder.cookie;
        this.appId = builder.appId;
    }


    /**
     * Returns a builder object.
     * @return Builder object.
     */
    public static Builder builder() {
        return new Builder();
    }

    /**
     * Determine a flow is permenant or not.
     *
     * @return a boolean
     */
    public boolean isPermanent() {
        return this.isPermanent;
    }

    /**
     * Returns the priority.
     *
     * @return priority.
     */
    public Integer getPriority() {
        return this.priority;
    }

    /**
     * Returns the table ID.
     * @return table ID.
     */
    public Integer getTableID() {
        return this.tableID;
    }


    /**
     * Sets table ID.
     * @param tableID flow rule table ID.
     */
    public void setTableID(int tableID) {
        this.tableID = tableID;
    }

    /**
     * Returns the flow time out.
     *
     * @return time out
     */
    public Integer getTimeOut() {
        return this.timeOut;
    }

    /**
     * Returns the application ID.
     *
     * @return app ID.
     */
    public String getAppId() {
        return this.appId;
    }

    /**
     * Returns the device ID.
     *
     * @return device ID.
     */
    public String getDeviceID() {
        return this.deviceID;
    }

    /**
     * Returns the cookie.
     *
     * @return cookie.
     */
    /**
     * Returns flow rule ID.
     * @return flow rule ID.
     */
    public String getFlowID() {
        return this.flowID;
    }

    /**
     * Sets flow rule ID.
     * @param flowID flow rule ID.
     */
    public void setFlowID(String flowID) {
        this.flowID = flowID;
    }

    /**
     * Returns flow rule cookie.
     * @return flow rule cookie.
     */
    public Integer getCookie() {
        return this.cookie;
    }

    /**
     * Returns the flow match fields object.
     * @return flow match object.
     */
    public FlowMatch getFlowMatch() {
        return this.flowMatch;
    }

    /**
     * Returns the list of flow action objects.
     *
     * @return flow action objects.
     */
    public List<FlowAction> getFlowActions() {
        return this.flowActions;
    }

    /**
     * Adds a flow match object to a flow rule.
     *
     * @param match match object.
     */
    public void addFlowMatch(FlowMatch match) {
        this.flowMatch = match;
    }

    /**
     * Adds a flow action object to a flow rule.
     * @param action flow action object.
     */
    public void addFlowAction(FlowAction action) {
        this.flowActions.add(action);
    }

    /**
```

```
     * Adds a list of flow actions object to a flow rule.
     * @param actions a list of flow action objects.
     */
    public void addFlowActions(List<FlowAction> actions) {
        this.flowActions.addAll(actions);
    }

    /**
     * Builder class to generate a Flow object.
     */
    public static class Builder {

        private FlowMatch flowMatch;
        private List<FlowAction> flowActions;
        private Integer priority;
        private Integer tableID;
        private String deviceID;
        private Integer timeOut;
        private boolean isPermanent;
        private Integer cookie;
        private String appId;

        public Builder() {
        }

        public Builder flowMatch(FlowMatch match) {
            this.flowMatch = match;
            return this;

        }

        public Builder priority(int priority) {
            this.priority = new Integer(priority);
            return this;
        }

        public Builder tableID(int tableID) {
            this.tableID = new Integer(tableID);
            return this;
        }

        public Builder deviceID(String deviceID) {
            this.deviceID = deviceID;
            return this;
        }

        public Builder timeOut(int timeOut) {
            this.timeOut = new Integer(timeOut);
            return this;
        }

        public Builder isPermanent(boolean isPermanent) {
            this.isPermanent = isPermanent;
            return this;
        }

        public Builder flowActions(List<FlowAction> actions) {
            this.flowActions = actions;
            return this;
        }
        public Builder cookie(int cookie) {
            this.cookie = new Integer(cookie);
            return this;
        }

        public Builder appId(String appId) {
            this.appId = appId;
            return this;
        }

        /**
         * Instantiates an object of type Flow.
         *
         * @return : Flow object.
         */
        public Flow build() {
            return new Flow(this);
        }
    }
}
```

C UMBRELLA TOPO VERTEX ABSTRACTION USING JAVA PROGRAMMING LANGUAGE

```java
/*
 * Copyright ${2018} [Adib Rastegarnia, Douglas Comer]
 *
 *    Licensed under the Apache License, Version 2.0 (the "License");
 *    you may not use this file except in compliance with the License.
 *    You may obtain a copy of the License at
 *
 *        http://www.apache.org/licenses/LICENSE-2.0
 *
 *    Unless required by applicable law or agreed to in writing, software
 *    distributed under the License is distributed on an "AS IS" BASIS,
 *    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *    See the License for the specific language governing permissions and
 *    limitations under the License.
 */

package api.topostore;

import java.util.Objects;

/**
 * Representation of nodes in a topology graph.
 */
public class TopoVertex {

    protected String ID;
    protected TopoVertexType type;

    public TopoVertex(TopoVertexType type) {
        this.type = type;
    }

    public TopoVertex(TopoVertexType type, String ID) {
        this.type = type;
        this.ID = ID;
    }

    /**
     * Return topo vertex ID.
     *
     * @return topo vertex ID.
     */
    public String getID() {
        return this.ID;
    }

    /**
     * Set topo vertex ID.
     *
     * @param ID topo vertex ID.
     */

    public void setID(String ID) {
        this.ID = ID;
    }

    /**
     * Return topo vertex type.
     *
     * @return topo vertex type.
     */
    public TopoVertexType getType() {
        return this.type;
    }

    /**
     * Set topo vertex type.
     *
     * @param type topo vertex type.
     */
    public void setType(TopoVertexType type) {
        this.type = type;
    }

    @Override
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }

        //System.out.println("TopoVertex:equals");
        TopoVertex topoVertex = (TopoVertex) obj;

        if (topoVertex.getID().equals(this.ID)) {
            return true;
        }

        return false;
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.ID);
    }
}
```

D UMBRELLA TOPO EDGE ABSTRACTION USING JAVA PROGRAMMING LANGUAGE

```java
/*
 * Copyright ${2018} [Adib Rastegarnia, Douglas Comer]
 *
 *    Licensed under the Apache License, Version 2.0 (the "License");
 *    you may not use this file except in compliance with the License.
 *    You may obtain a copy of the License at
 *
 *        http://www.apache.org/licenses/LICENSE-2.0
 *
 *    Unless required by applicable law or agreed to in writing, software
 *    distributed under the License is distributed on an "AS IS" BASIS,
 *    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 *    See the License for the specific language governing permissions and
 *    limitations under the License.
 */

package api.topostore;

import org.jgrapht.graph.DefaultEdge;

import java.util.Objects;

/**
 * Representation of network links in a network topology.
 */
public class TopoEdge extends DefaultEdge implements TopoEdgeInterface {

    /**
     * Source network device.
     */
    private String src;
    /**
     * Destination network device.
     */
    private String dst;
    /**
     * Source device port.
     */
    private String srcPort;
    /**
     * Destination device port.
     */
    private String dstPort;
    /**
     * Topology link state.
     */
    private String state;
    /**
     * Topology link type.
     */
    private TopoEdgeType type;
    /**
     * Topology link weight.
     */
    private int weight;
    /**
     * Topology link label.
     */
    private String label;

    public TopoEdge() {
        src = null;
        dst = null;
        srcPort = null;
        dstPort = null;
        state = null;
        type = null;
        weight = 0;
        label = null;
    }

    /**
     * Returns type of a link.
     *
     * @return type of a link.
     */
    public TopoEdgeType getType() {
        return this.type;
    }

    /**
     * Sets type of a link.
     *
     * @param type TopoEdgeType
     */
    public void setType(TopoEdgeType type) {
        this.type = type;
    }

    /**
     * Returns destination attachment point in a topology link.
     *
     * @return destination attachment point.
     */
    public String getDst() {
        return this.dst;
    }

    /**
     * Sets destination attachment point in a topology link.
     *
     * @param dst destination attachment point.
     */
    public void setDst(String dst) {
        this.dst = dst;
    }


    /**
```

```
     * Returns link destination port.
     * @return destination port.
     */
    public String getDstPort() {
        return this.dstPort;
    }

    /**
     * Sets link destination port.
     * @param dstPort destination port.
     */
    public void setDstPort(String dstPort) {
        this.dstPort = dstPort;
    }

    /**
     * Returns source network device of a link.
     * @return source network device ID.
     */
    public String getSrc() {
        return this.src;
    }

    /**
     * Sets source network device of a link.
     * @param src source network device ID.
     */
    public void setSrc(String src) {
        this.src = src;
    }

    /**
     * Returns link source port.
     * @return source port.
     */
    public String getSrcPort() {
        return this.srcPort;
    }

    /**
     * Sets link source port.
     * @param srcPort source port.
     */
    public void setSrcPort(String srcPort) {
        this.srcPort = srcPort;
    }

    /**
     * Returns state of a link in the topology.
     * @return state of a link.
     */
    public String getState() {
        return this.state;
    }

    /**
     * Sets state of a link in the topology.
     * @param state
     */
    public void setState(String state) {
```

```java
        this.state = state;
    }

    /**
     * Returns weight of a link.
     * @return weight of a link.
     */
    public int getWeight() {
        return this.weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }

    public String getLabel() {
        return this.label;
    }


    public void setLabel(String label) {
        this.label = label;
    }

    @Override
    public boolean equals(Object o) {
        if (o == this) {
            return true;
        }
        if (!(o instanceof TopoEdge)) {
            return false;
        }
        TopoEdge topoEdge = (TopoEdge) o;
        return Objects.equals(src, topoEdge.src) && Objects.equals(dst, topoEdge.dst)
                && Objects.equals(srcPort, topoEdge.srcPort) && Objects.equals(dstPort, topoEdge.dstPort);
    }

    @Override
    public int hashCode() {
        return Objects.hash(src, srcPort, dst, dstPort);
    }
}
```

E IMPLEMENTATION OF FORWARDING APPLICATION USING UMBRELLA

```java
package apps;

import api.flowservice.Flow;
import api.flowservice.FlowAction;
import api.flowservice.FlowActionType;
import api.flowservice.FlowMatch;
import api.topostore.TopoEdge;
import api.topostore.TopoEdgeType;
import api.topostore.TopoHost;
import config.ConfigService;
import drivers.controller.Controller;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;

public class Forwarding {
    public static void main(String[] args) {


        String controllerName;

        Controller controller = null;
        ConfigService configService = new ConfigService();
        controllerName = configService.getControllerName();

        controller = configService.init(controllerName);

        Set<TopoHost> srchosts = controller.topoStore.getHosts();
        Set<TopoHost> dsthosts = controller.topoStore.getHosts();

        List<TopoEdge> path = null;


        for (TopoHost srcHost : srchosts) {
            for (TopoHost dstHost : dsthosts) {
                if (!srcHost.equals(dstHost)) {
                    String srcMac = srcHost.getHostMac();
                    String dstMac = dstHost.getHostMac();


                    path = controller.topoStore.getShortestPath(srcHost.getHostID(), dstHost.getHostID());
                    controller.printPath(path);

                    for (TopoEdge edge : path) {

                        if (edge.getType() == TopoEdgeType.HOST_SWITCH) {
                            continue;
                        }
                        FlowMatch flowMatch = FlowMatch.builder()
                                .ethSrc(srcMac)
                                .ethDst(dstMac)
                                .ipv4Src(srcHost.getHostIPAddresses().get(0) + "/32")
                                .ipv4Dst(dstHost.getHostIPAddresses().get(0) + "/32")
                                .ethType(2048)
                                .build();

                        FlowAction flowAction = new FlowAction(FlowActionType.OUTPUT,
                                Integer.parseInt(edge.getSrcPort()));

                        ArrayList<FlowAction> flowActions = new ArrayList<FlowAction>();
                        flowActions.add(flowAction);

                        Flow flow = Flow.builder()
                                .deviceID(edge.getSrc())
                                .tableID(0)
                                .flowMatch(flowMatch)
                                .flowActions(flowActions)
                                .priority(1000)
                                .appId("TestForwarding")
                                .timeOut(50)
                                .build();

                        controller.flowService.addFlow(flow);
                    }


                }
            }
        }
    }
}
```

F IMPLEMENTATION OF FIREWALL APPLICATION USING UMBRELLA

```
package apps;

import api.flowservice.Flow;
import api.flowservice.FlowAction;
import api.flowservice.FlowActionType;
import api.flowservice.FlowMatch;
import api.topostore.TopoEdge;
import api.topostore.TopoEdgeType;
import api.topostore.TopoHost;
import config.ConfigService;
import drivers.controller.Controller;

import java.util.ArrayList;
import java.util.List;
import java.util.Set;

public class Firewall {

    public static void main(String[] args) {

        String controllerName;

        Controller controller = null;
        ConfigService configService = new ConfigService();
        controllerName = configService.getControllerName();

        controller= configService.init(controllerName);


        Set<TopoHost> srchosts = controller.topoStore.getHosts();

        ArrayList<TopoHost> hosts = new ArrayList<>(srchosts);

        List<TopoEdge> fwPath = null;
        List<TopoEdge> rvPath = null;


        for (TopoHost srcHost: hosts) {
            for (TopoHost dstHost: hosts) {
                if(!srcHost.equals(dstHost)){

                String srcMac = srcHost.getHostMac();
                String dstMac = dstHost.getHostMac();

                String srcIP = srcHost.getHostIPAddresses().get(0);
                String dstIP = dstHost.getHostIPAddresses().get(0);

                fwPath = controller.topoStore.getShortestPath(srcHost.getHostID(), dstHost.getHostID());
                rvPath = controller.topoStore.getShortestPath(dstHost.getHostID(), srcHost.getHostID());

                if ((srcIP.equals("10.0.0.1") && dstIP.equals("10.0.0.3"))
                || (srcIP.equals("10.0.0.3") && dstIP.equals("10.0.0.1"))) {


            FlowMatch flowMatch = null;

            for (TopoEdge edge : fwPath) {

                if (edge.getType() == TopoEdgeType.HOST_SWITCH) {
                    continue;
                }

                flowMatch = FlowMatch.builder()
                        .ethSrc(srcMac)
                        .ethDst(dstMac)
                        .ipv4Src(srcHost.getHostIPAddresses().get(0) + "/32")
                        .ipv4Dst(dstHost.getHostIPAddresses().get(0) + "/32")
                        .ethType(2048)
                        .ipProto(6)
                        .tcpDst(80)
                        .build();


                FlowAction flowAction = new FlowAction(FlowActionType.OUTPUT,
                        Integer.parseInt(edge.getSrcPort()));

                ArrayList<FlowAction> flowActions = new ArrayList<FlowAction>();
                flowActions.add(flowAction);

                Flow flow = Flow.builder()
                        .deviceID(edge.getSrc())
                        .tableID(0)
                        .flowMatch(flowMatch)
                        .flowActions(flowActions)
                        .priority(100)
                        .appId("Firewall")
                        .timeOut(100)
                        .build();

                controller.flowService.addFlow(flow);

            }

            // Reverse Path

            for (TopoEdge edge : rvPath) {

                if (edge.getType() == TopoEdgeType.HOST_SWITCH) {
                    continue;
                }

                flowMatch = FlowMatch.builder()
                        .ethSrc(dstMac)
                        .ethDst(srcMac)
                        .ipv4Src(dstHost.getHostIPAddresses().get(0) + "/32")
                        .ipv4Dst(srcHost.getHostIPAddresses().get(0) + "/32")
                        .ethType(2048)
                        .ipProto(6)
                        .tcpSrc(80)
                        .build();

                FlowAction flowAction = new FlowAction(FlowActionType.OUTPUT,
                        Integer.parseInt(edge.getSrcPort()));

                ArrayList<FlowAction> flowActions = new ArrayList<FlowAction>();
                flowActions.add(flowAction);

                Flow flow = Flow.builder()
                        .deviceID(edge.getSrc())
                        .tableID(0)
                        .flowMatch(flowMatch)
                        .flowActions(flowActions)
                        .priority(100)
                        .appId("Firewall")
                        .timeOut(100)
                        .build();

                controller.flowService.addFlow(flow);


            }


        }

        if ((srcIP.equals("10.0.0.2") && dstIP.equals("10.0.0.4"))
                || (srcIP.equals("10.0.0.4") && dstIP.equals("10.0.0.2"))) {


            FlowMatch flowMatch = null;

            for (TopoEdge edge : fwPath) {

                if (edge.getType() == TopoEdgeType.HOST_SWITCH) {
                    continue;
                }


                flowMatch = FlowMatch.builder()
                        .ethSrc(srcMac)
                        .ethDst(dstMac)
                        .ipv4Src(srcHost.getHostIPAddresses().get(0) + "/32")
                        .ipv4Dst(dstHost.getHostIPAddresses().get(0) + "/32")
                        .ipProto(0x01)
                        .ethType(2048)
                        .icmpv4_code(0x0)
                        .icmpv4_type(0x08)
                        .build();


                FlowAction flowAction = new FlowAction(FlowActionType.OUTPUT,
                        Integer.parseInt(edge.getSrcPort()));

                ArrayList<FlowAction> flowActions = new ArrayList<FlowAction>();
                flowActions.add(flowAction);

                Flow flow = Flow.builder()
                        .deviceID(edge.getSrc())
                        .tableID(0)
                        .flowMatch(flowMatch)
                        .flowActions(flowActions)
                        .priority(100)
                        .appId("Firewall")
                        .timeOut(100)
                        .build();

                controller.flowService.addFlow(flow);


            }
            // Reverse Path

            for (TopoEdge edge : rvPath) {

                if (edge.getType() == TopoEdgeType.HOST_SWITCH) {
                    continue;
                }


                flowMatch = FlowMatch.builder()
                        .ethSrc(dstMac)
                        .ethDst(srcMac)
                        .ipv4Src(dstHost.getHostIPAddresses().get(0) + "/32")
                        .ipv4Dst(srcHost.getHostIPAddresses().get(0) + "/32")
                        .ipProto(0x01)
                        .ethType(2048)
                        .icmpv4_code(0x0)
                        .icmpv4_type(0x0)
                        .build();


                FlowAction flowAction = new FlowAction(FlowActionType.OUTPUT,
                        Integer.parseInt(edge.getSrcPort()));

                ArrayList<FlowAction> flowActions = new ArrayList<FlowAction>();
                flowActions.add(flowAction);

                Flow flow = Flow.builder()
                        .deviceID(edge.getSrc())
                        .tableID(0)
                        .flowMatch(flowMatch)
                        .flowActions(flowActions)
                        .priority(100)
                        .appId("Firewall")
                        .timeOut(100)
                        .build();

                controller.flowService.addFlow(flow);


            }


        }
      }
     }
    }
  }
}
```

VITA

Adib Rastegarnia was born in Isfahan, a city in central Iran, known for its Persian architecture. He earned his Bachelors degree in Information Technology (IT) Engineering in 2010 from Urmia University of Technology. He received a Master's degree in IT Engineering in 2014 from University of Tehran. Adib also earned a Masters degree in Computer Science from Purdue University along his PhD studies.

Adib joined System Research group at Computer Science Department of Purdue University. His current research interests span the areas of computer networks, operating systems, Software Defined Networking (SDN), and Internet of Things (IoT) with a focus on designing and implementing working prototypes of large, complex software systems and conducting real world measurements. Since Adib joined Purdue, he received various awards such as Purdue CS Teaching Fellowship Award, Purdue CS Department Corporate Partners Research Award, and CS Research/Teaching Assistant Award.

Adib served as a reviewer for well known networking journals and conferences such as IEEE Communication Magazine, IEEE Networking Letters, IEEE Potential Magazine, and Internet Measurement Conference (IMC). He served as the Vice Chair of IEEE Region 4 Student Activities Committee (SAC) between 2017 and 2019. As part of his volunteer activities, Adib served as organizing committee member of IEEE Student Leadership Conference (SLC) 2018 and as the workshop manager for 2017 Humanitarian Technology Day Conference.

Adib designed and implemented multiple open source SDN programming frameworks and experimental testbeds during his PhD studies. He actively contributes in open source SDN projects such as next generation of ONOS SDN controller that is based on a microservice architecture.