Analysis and Visualization of OpenFlow Rule Conflicts

by

Janakarajan Natarajan

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved April 2016 by the
Graduate Supervisory Committee:

Dijiang Huang, Co-Chair
Violet R. Syrotiuk, Co-Chair
Gail-Joon Ahn

ARIZONA STATE UNIVERSITY

May 2016

ABSTRACT

In traditional networks the control and data plane are highly coupled, hindering development. With Software Defined Networking (SDN), the two planes are separated, allowing innovations on either one independently of the other. Here, the control plane is formed by the applications that specify an organization's policy and the data plane contains the forwarding logic. The application sends all commands to an SDN controller which then performs the requested action on behalf of the application. Generally, the requested action is a modification to the flow tables, present in the switches, to reflect a change in the organization's policy. There are a number of ways to control the network using the SDN principles, but the most widely used approach is OpenFlow.

With the applications now having direct access to the flow table entries, it is easy to have inconsistencies arise in the flow table rules. Since the flow rules are structured similar to firewall rules, the research done in analyzing and identifying firewall rule conflicts can be adapted to work with OpenFlow rules.

The main work of this thesis is to implement flow conflict detection logic in OpenDaylight and inspect the applicability of techniques in visualizing the conflicts. A hierarchical edge-bundling technique coupled with a Reingold-Tilford tree is employed to present the relationship between the conflicting rules. Additionally, a table-driven approach is also implemented to display the details of each flow.

Both types of visualization are then tested for correctness by providing them with flows which are known to have conflicts. The conflicts were identified properly and displayed by the views.

DEDICATION

I dedicate this work to my mother, father and brother.

ACKNOWLEDGEMENTS

I am extremely grateful to my advisors Dr. Dijiang Huang and Dr. Violet R. Syrotiuk for their support and encouragement. My work would not have been possible without the help of Sandeep Pisharody and Dr. Chun-Jen Chung. I am glad to have had the opportunity to work with them.

Attending the SNAC talks and learning about others' works was truly an eye opener. I was able to see first-hand some really innovative solutions to complex problems. It was great to have been part of the SNAC and I will take this experience with me wherever I go.

Finally, I would like to thank Dr. Gail-Joon Ahn for providing me with feedback on my work.

TABLE OF CONTENTS

Page

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

1 INTRODUCTION 1

1.1 Conflicting Rules 3

1.2 Related Work 6

1.3 Research Outline 9

2 OFANALYZER ARCHITECTURE 10

2.1 Flow Extraction 11

2.2 Conflict Detection 11

2.3 Conflict Visualization 12

3 BACKGROUND 13

3.1 Overview 13

3.2 OSGi 14

3.3 Apache Karaf 15

3.4 OpenDaylight Bundles 16

3.5 RESTCONF 17

3.5.1 NETCONF 17

3.5.2 YANG 18

3.6 Model Driven-Service Abstraction Layer 19

3.6.1 Service Abstraction Layer 19

3.6.2 Model Centric Applications 20

3.7 MD-SAL Application 20

3.8 Data Trees 22

CHAPTER Page

4 FLOW EXTRACTION 25

4.1 Flow Listener 25

4.2 Flow Structure 27

4.3 Flow Extraction Algorithm 27

5 CONFLICT DETECTION 29

5.1 Conflict Detection Algorithm 29

5.2 Transferring Rules 32

5.3 Send All Conflicts 32

5.4 Use Encoding Scheme 33

5.5 Compilation Process 35

6 CONFLICT VISUALIZATION 37

6.1 Fetch Rules 38

6.2 WebSocket Listener 38

6.3 REST API 39

6.4 Process Retrieved Rules 41

6.5 Transforming the Rules 42

6.5.1 D3 - JavaScript Library 42

6.6 Visualization of OpenFlow Rule Conflicts 44

6.6.1 Tabular Visualization 44

6.6.2 Hierarchical Edge Bundle 45

6.6.3 Reingold-Tilford Trees 46

7 EVALUATION 49

8 CONCLUSION AND FUTURE WORK 53

8.1 Conclusion 53

8.2 Future Work 53

REFERENCES 55

LIST OF TABLES

Table		Page
1.1	Example Flow Table	4
5.1	Rule Conflicts	32
5.2	Types of Rule Conflicts	33
5.3	Conflict Type Assignment	34
5.4	Encoded Table	35
7.1	Test Flows	52

LIST OF FIGURES

Figure		Page
1.1	Traditional Network	1
1.2	OpenFlow Network	2
2.1	OFAnalyzer Architecture	10
3.1	OpenDaylight Lithium Architecture (Reduced)	14
3.2	Apache Karaf	16
3.3	MD-SAL Application Generation	21
3.4	OpenDaylight Trees	23
4.1	Flow Structure	26
6.1	POSTMAN REST Invocation	40
6.2	Landing Page	43
6.3	Tabular View	44
6.4	Hierarchical Edge Bundling	46
6.5	Reingold-Tilford Tree	47
7.1	Test Topology	49
7.2	Test Flows Landing Page	50
7.3	Reingold-Tilford Relationship	51

Chapter 1

INTRODUCTION

A network can comprise of numerous switches where each can have many hosts connecting to it. All packets to and from these hosts have to be routed correctly to ensure proper functioning of the network. The routing decisions are made based on a variety of parameters such as number of hops, traffic along the hops, Quality of Service guarantees etc., along with the equally important business logic parameter which reflects an organization's policies regarding resource use. The switch is tasked with performing the routing based on such parameters.

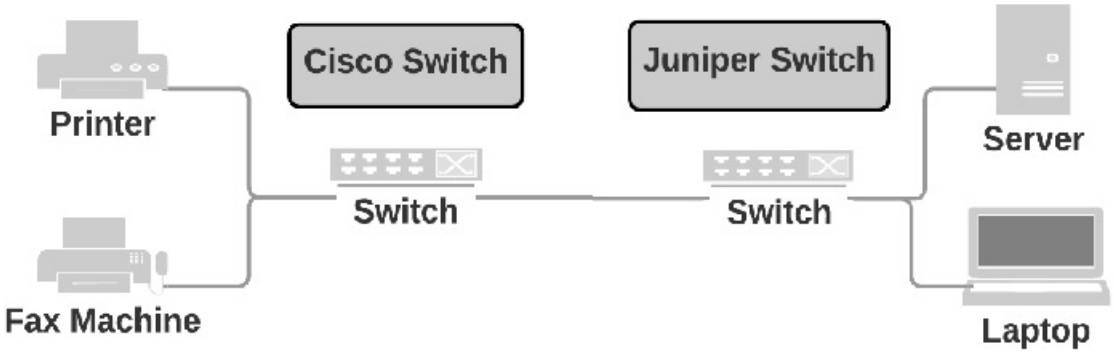


Figure 1.1: Traditional Network

Generally, the business logic is implemented in the control plane. Here the requirements, as specified by an organization, are translated into system level instructions. The actual routing of packets along with the full implementation details of packet handling are part of the data plane. In this way the control plane dictates and governs the access to the network while the data plane handles the actual packet routing.

An example network can be seen in Figure 1. Here the network comprises of a Cisco and a Juniper switch, providing access to the server and printer respectively. In order to prevent the laptop from accessing the server and printer, changes must be made to the Access Control Lists of both the switches separately. Hence, any change in the organization's policy will require work to be duplicated for each switch. This is a big problem as it prevents the organization from having its requirements applied generically throughout its network which may comprise of elements from different vendors. Moreover, since the traditional networks have the two planes tightly coupled, the network administrator will not be able to make changes to the network rapidly.

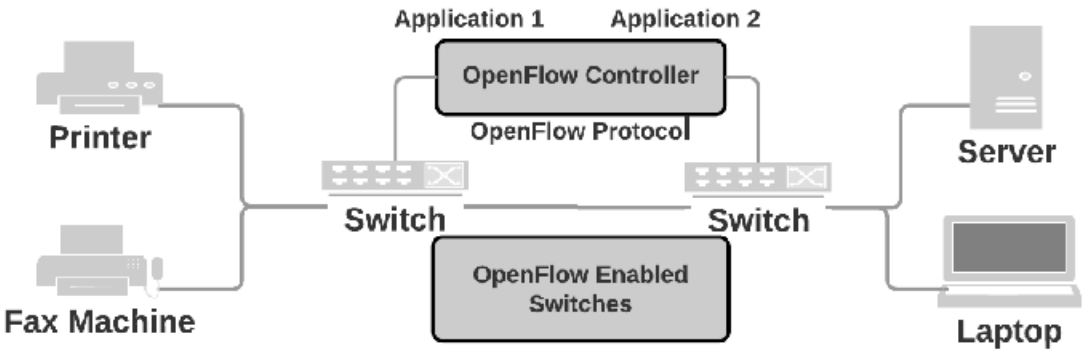


Figure 1.2: OpenFlow Network

OpenFlow [1] aims to solve this issue. It does this by decoupling the control and data plane thereby allowing each to be manipulated independent of the other. The data plane, comprising of the switch flow tables, is controlled by applications that form the control plane. The communication with the flow table takes place through the OpenFlow protocol. Generally, the application specifies its requirements to an OpenFlow controller which then uses the protocol to push the requests down to the switches. An example of an OpenFlow enabled network is shown in Figure 2.

Here the applications 1 and 2 run on top of the OpenFlow controller and directly

indicate their requirement to it. The controller then uses the OpenFlow protocol to push the necessary changes to the flow table. The switches under the controller are *OpenFlow enabled* which indicates that the flow table is accessible from the outside world.

1.1 Conflicting Rules

With the applications now having access to the internal flow tables of the switches, there is a high probability that the flows may conflict with one another. This can be caused by a number of reasons such as logical errors in the application, misbehaving programs and applications being generally unaware of the intentions of its peers.

It is preferred to always have a conflict-free flow table to ensure the network reflects the requirements set by the organization. For example, packets from legitimate clients to use a resource may be dropped due to a conflicting rule. Another scenario may involve creating security holes in the network wherein unauthorized clients may gain privileged access to services offered.

Conflicts come in a variety of flavours. A number of works such as [1], [2] have classified the conflicts present in firewalls into five categories. They are:

- Shadowing
- Generalization
- Correlation
- Redundancy
- Overlap

Since the flows in an OpenFlow architecture are a superset of the firewall rules, these classifications are applicable here as well.

Rule	Priority	Src IP	Dst IP	Prot	Src Port	Dst Port	Action
1	10	10.10.*.*	11.11.1.*	TCP	6001	6002	Forward
2	99	10.10.1.*	11.11.1.2	TCP	7867	7868	Deny
3	100	10.11.1.2	11.12.1.1	TCP	23	23	Deny
4	101	10.11.1.*	11.12.1.*	TCP	*	*	Forward
5	102	10.10.1.*	11.11.1.*	*	*	*	Deny
6	200	10.10.1.1	11.11.1.1	UDP	40	40	Deny

Table 1.1: Example Flow Table

A point of interest is that in firewalls the order of rules in the table inherently indicates the priority of the rules. Hence a rule r_x has a higher priority than a rule r_y if r_x is higher in the table than r_y. In OpenFlow there is a separate field called *priority* that is used to indicate the priority of each rule. This *Priority* field is explicitly used when classifying the OpenFlow rules into one of the following five types:

Shadowing

A rule r_y is said to have a *shadowing conflict* with r_x if r_x matches the same packet space as r_y but has a different action and a higher priority compared to r_y. In Table 1, rule 2 is shadowed by rule 1. Here the packet space of rule 1 is a superset of rule 2. All packets belonging to the space 10.10.1.* will also be under 10.10.*.*. Hence rule 2 will never be utilized. This is a conflict because the action of rule 2 is to Deny while rule 1 will allow it. This can create a security hole if the network administrator is not careful.

Generalization

A rule r_y is said to have a *generalization conflict* with r_x if r_x matches a subset of r_y's packet space and r_x has a higher priority and different action compared to r_y. In Table 1, rule 3 matches a subset of rule 4 and has a different action when compared to rule 4. Since 10.11.1.2 is a subset of the packet space of 10.11.1.* and the rule has a different action when compared to rule 4, this is a generalization conflict. This type of conflict may not be an error. Network administrators use this type of flow to blacklist particular hosts from accessing the network. In this case, all hosts except 10.11.1.2 are allowed to reach their destination.

Correlation

A rule r_y correlates with another rule r_x if r_y's packet space intersects with r_x's packet space but r_ys' action differs from r_xs'. Hence any packet that matches this intersection will be either allowed or denied based on the priority of the correlating rules. In Table 1, rule 1 and rule 5 have a correlation conflict. Rule 5 denies all packets from 10.10.1.* to 11.11.1.* with any protocol and source, destination ports. However, rule 1 allows all packets from 10.11.*.* to 11.11.1.* with TCP protocol and 6001 and 6002 as source and destination ports respectively. If rule 5 had a higher priority then all packets from 10.10.1.* to 11.11.1.* would be denied. Since rule 1 has higher priority all TCP packets from 10.10.1.* to 11.11.1.* are allowed in, again creating a security hole.

Redundancy

A rule r_y is redundant if there exists another rule r_x which matches the same packet space or is a more general rule of r_y and has the same action. In Table 1, rule

5 and rule 6 have a redundancy conflict. Rule 5 is much more general than rule 6, hence all packets matched by rule 5 will also be matched by rule 6. Since both rules have the same action, the packets will not be affected by the priority.

Overlap

A rule r_y overlaps another rule r_x if r_x matches the same address space of r_y and both their actions are the same. Overlap conflicts are , in essence, correlation conflicts when the actions are the same between the two compared rules.

1.2 Related Work

The OpenFlow rules have a number of fields such as *Priority, VLAN-ID, Source MAC, Destination MAC, Source IP, Destination IP, Source Port, Destination Port, Protocol and Action* with many types of actions supported in each version of OpenFlow. This is startlingly similar to firewall rules. Since OpenFlow rules are a super-set of firewall rules, the techniques used to identify conflicts among the firewall rules can be adapted to work with OpenFlow as well.

One of the earliest works on identifying conflicts or errors in the firewall is the work done by Mayer et al. in [2]. Their tool called *Fang* reads in the vendor specific configuration files and converts it into an internal representation. It also presents a GUI to the network administrator, where queries can be presented to the system and the results displayed in a tabular format. From an OpenFlow perspective Fang can be considered, but it would not fit the need to display the relation between conflicting rules. Since the onus is on the user to query the system, conflicts will show up only if the right query is presented.

Al-Shaer and Hamed in [3] provide one of the most comprehensive tools to identify conflicts in firewalls. Here the rules are internally represented in a *policy tree* which

aids in the conflict identification process. Their classification of the firewall conflicts forms the basis of a number of further works in this area and can be applied to OpenFlow as well. A major drawback of their tool *Firewall Policy Advisor* is that the conflict information is displayed as text in their GUI. Presenting details in a textual format can make it difficult to visualize the relationship between the conflicting rules.

In [4] Hu et al. describe a tool called *FAME* which identifies the conflicts among the firewall rules and displays them in a tabular format. The conflicts are placed into segments, grouping of conflicts based on their type. This segmentation aids in the visualization of the relationship between the conflicting rules. Moreover, this tool also provides conflict resolution based on *action constraint generation* and *rule reordering*. This thesis models the tabular format view for the OpenFlow rules after the approach followed by FAME.

Al-Shaer and Hamed have recognized the viability of using firewall conflict detection techniques for OpenFlow conflict detection. They have provided a tool *FlowChecker* in [5]. Here the OpenFlow rules are internally represented using a structure called *Binary Decision Diagram* (BDD). The BDD encodes the location of the packet in the network along with each field of the flow rule. With this representation, reducing the BDD into a *Reduced Ordered Binary Decision Diagram* assists in the detection of conflicts among the flows. However, no effective visualization technique is discussed to give the network administrator an overall picture of the network.

Another tool that uses BDDs for identifying firewall rule inconsistencies is [6]. The tool takes into consideration the blacklists and whitelists that the network administrator may have setup and models all paths that a packet may take in a network. Since many networks have firewalls from different vendors, the configuration files are represented internally in a vendor-neutral manner. This tool also considers the firewalls which are stand-alone as well as those that are distributed firewalls.

Finally, different conflict detection techniques for OpenFlow rules are discussed in [7]. Two different approaches are analyzed. The first approach is to internally represent the flows using a combination of a hash and a radix trie. Apart from the *Source and Destination IP address* fields the remaining fields in an OpenFlow rule are encoded in a hash trie. The hash trie is used for this purpose as these fields are exact match fields. A radix trie is chosen for the IP headers because these headers are prefix-based and according to [8] is the preferred data structure for representing prefix-based fields. The advantage of this structure is the highly compressed nature of the nodes, where a large amount of information is presented with minimal tree elements. Moreover, this trie is used internally in routers to match packets to their respective flows when performing Quality of Service control. The output of the hash and radix tries is then utilized to identify the conflicts among the flow rules. The second approach is an *Ontology based Conflict Detection*. According to [7] a logic system is used to provide an effective way to infer the conflicts from the added flow rules.

The main aim of this thesis is to provide an array of options to visualize the relationship between the conflicting flow rules. A number of works have focused on the visualization of conflicts as well. PolicyVis [9] also utilizes Binary Decision Diagrams to segment the rules based on accepted or denied actions. A two dimensional graph is used to visualize the firewall rules, where the co-ordinate axes can be controlled. A third dimension is used by placing special shapes inside the graphs, signifying another value of the firewall rule e.g. TCP, UDP etc. This helps the user to choose the required scope when viewing the firewall rules or segments. PolicyVis also provides support for viewing distributed firewall rules and segments. The major drawback of this approach is that the conflict visualization depends on the user choosing the right dimension for the graph axes.

Frequently, a large number of factors are considered when making predictions for any event. The same is true for college football. In [10], a website from *Ed Fang* visualizes the predictions for the NCAA championships using a unique circular tree structure. This makes it easier to identify the relationship between the predictions and their probability to win the championship. This thesis uses a similar approach to visualizing the flow rule conflicts by employing the Reingold-Tilford tree and hierarchical edge bundle.

1.3 Research Outline

With a basic understanding of the OpenFlow architecture, the types of conflicts that may arise and the work done to minimize, correct and in some cases visualize them the next chapters deal with architecture of the tool presented in this thesis called *OFAnalyzer*. Chapter 2 takes a closer look at the overall architecture of the tool.

In Chapter 3, the open-source OpenFlow controller - OpenDaylight - is elaborated upon. An introduction is provided of its internal workings along with its associated tools and technologies. In subsequent chapters the three major tasks, i.e., flow extraction, flow conflict detection and flow visualization is explained in detail. Chapter 4 discusses the flow extraction process in detail. In Chapter 5, the conflict detection technique used in OFAnalyzer is described. The types of conflicts and the data structures that are used are explained in detail.

In Chapter 6, the visualization of the conflicts is presented. In Chapter 7, the tool is evaluated and its function is checked against a set of rules that are known to have conflicts among them. Chapter 8 presents the future work, where the many ways this tool can be used is described.

Chapter 2

OFANALYZER ARCHITECTURE

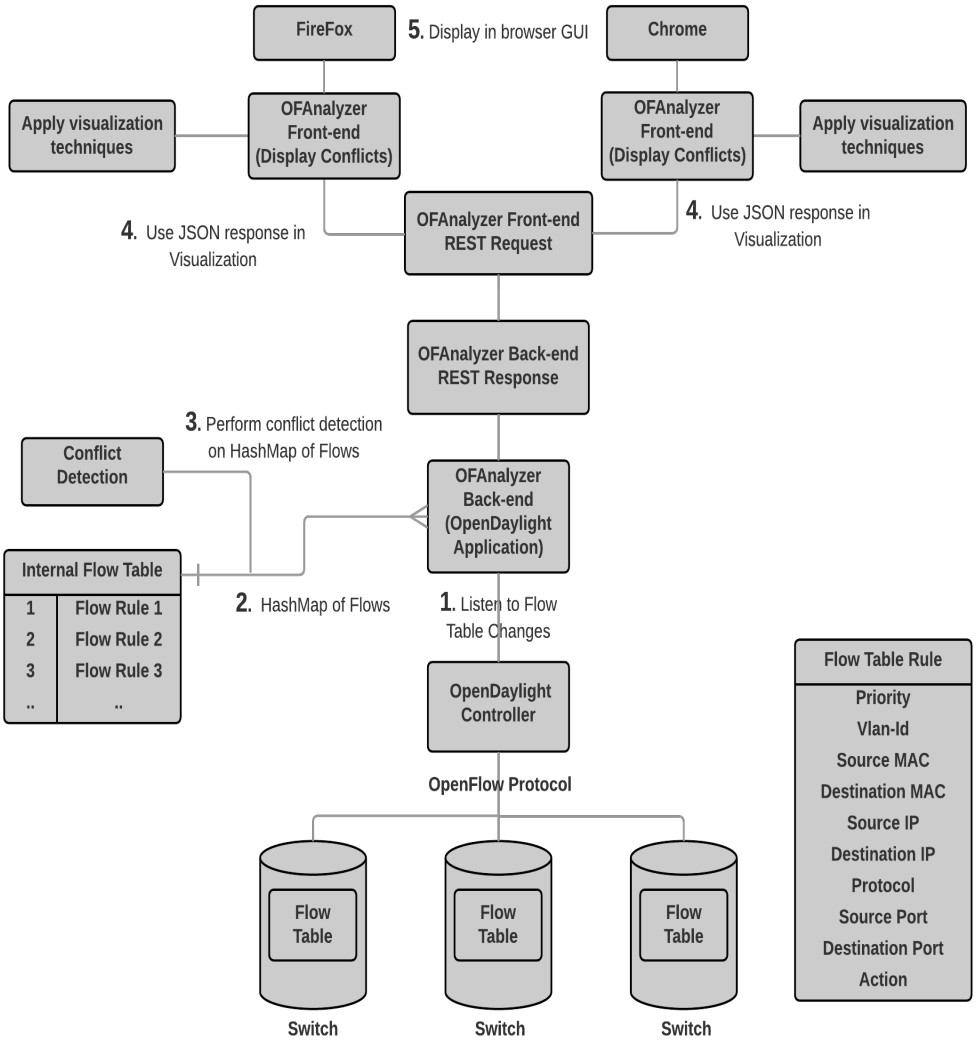


Figure 2.1: OFAnalyzer Architecture

The architecture of the OFAnalyzer is shown in 2.1. The OFAnalyzer performs

three important tasks. They are:

- Flow Extraction
- Conflict Detection and
- Conflict Visualization

The following sections briefly describe each task.

2.1 Flow Extraction

The OpenDaylight controller is connected to the OpenFlow switches. As packets arrive at these switches, if a flow rule is not present then the packet is forwarded to the controller to be processed. The controller then injects flow rules into the switch so that future packets that match the flow are processed by the switch itself. The flow rules can also be proactively installed by applications running on the controller.

The flow rules installed from any source are extracted by the OFAnalyzer back-end as the first step. It does this by listening to the flow rule tree maintained by the OpenDaylight controller. These flows are stored in an internal HashMap.

2.2 Conflict Detection

The flows that OFAnalyzer receives may conflict with flows existing in the internal HashMap. An $O(n^2)$ algorithm is used to identify the conflicts among the rules. n represents the number of flows in the flow table. This algorithm is similar to the one used in [11]. The conflict detection algorithm used here compares each flow rule as it arrives at the OFAnalyzer listener with the rules that came before it. As each rule is compared, any conflict detected is encoded into a field called *Conflict-Type*. This field is a semi-colon separated string holding the conflict information of each rule with all the other rules in the flow table.

2.3 Conflict Visualization

The OFAnalyzer front-end is a module under the DLUX UI [12]. Here the module performs a REST Request to get the flow rules along with the conflict information. The back-end responds to this REST Request by sending all the available flows as a list of JSON objects. With the JSON objects, JavaScript conversion routines are applied to prepare the flow rules for visualization.

Once these flows are received by the front-end, the flow rules are formatted using JavaScript routines and various visualization techniques are applied to this data. The information is then presented in a GUI inside the DLUX project, as a sub module. The information displayed can be viewed in a number of ways.

The subsequent chapter goes into detail about the tools and languages used for each task listed above.

Chapter 3

BACKGROUND

OFAnalyzer uses the OpenDaylight controller to listen for changes to the OpenFlow switch flow tables. Before the flow extraction process is examined in detail, it is necessary to have an overview of OpenDaylight and its associated tools and technologies.

3.1 Overview

OpenDaylight is an open-source project under The Linux Foundation [13]. The stable version *Lithium* of OpenDaylight is used in this thesis. Its architecture can be seen in 3.1. OpenDaylight has REST API's which are used by external applications. Applications that exist inside of OpenDaylight make use of the Service Abstraction Layer to communicate with different types of devices. They do so by using the various communication protocols such as OpenFlow, Netconf, CAPWAP etc. that are supported in Lithium.

There are a number of SDN controllers such as NOX, Floodlight and Beacon that support OpenFlow as its southbound plugin but OpenDaylight is regarded by many as the industry standard SDN controller. Apart from open-source flavors there are commercial alternatives such as Big Switch's Switch Light, Brocade's Vyatta Controller, Ericsson's SDN Controller and many more as listed in [14]. The OpenDaylight project is home to a number of sub-projects which when combined together provide a vast array of features under a single package. Some examples of such projects are *openflowplugin, controller, dlux, l2switch* and *yangtools*. These sub-projects are available in Gerrit mirrored in GitHub [15], a code review platform for Git.

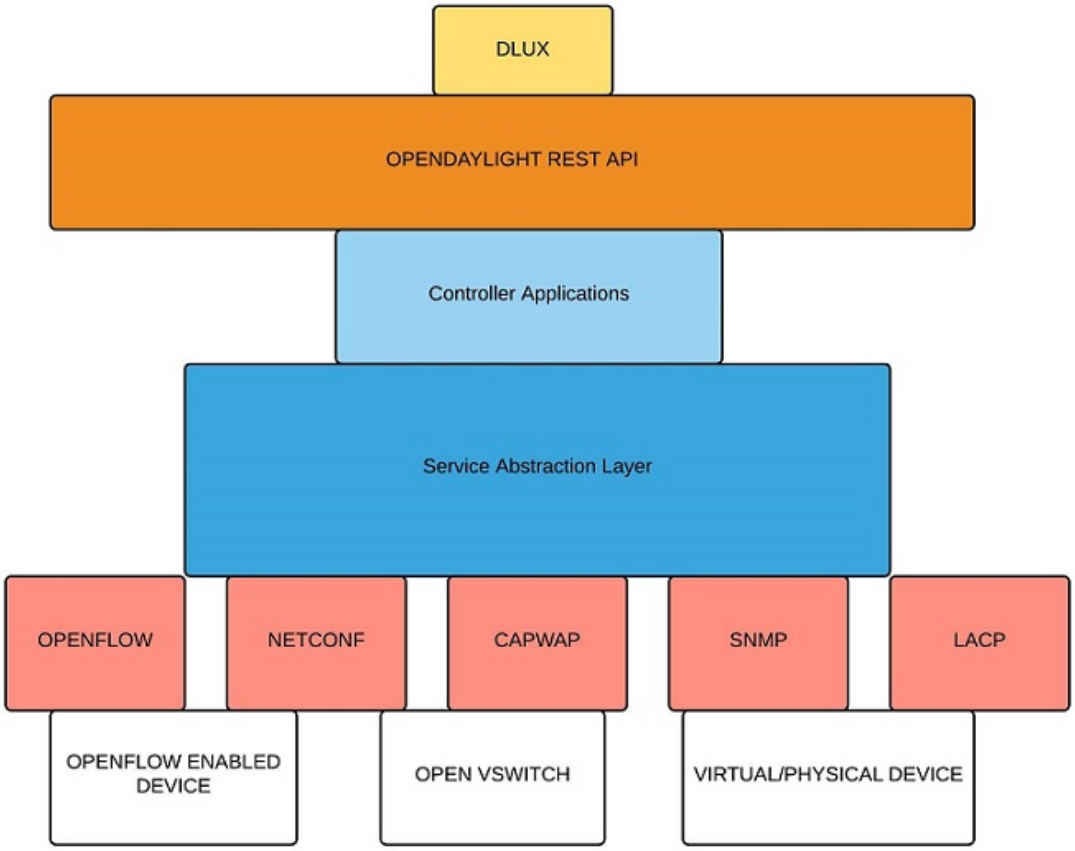


Figure 3.1: OpenDaylight Lithium Architecture (Reduced)

The division of functionality among different projects helps in simplifying the design of the controller. If, for example, the OpenFlow specification is updated to provide newer action types, the *openflowplugin* project alone can be updated to reflect this change. This separation of functionality is possible due to the use of the Open Services Gateway Initiative (OSGi) specification.

3.2 OSGi

The OSGi specification [16] and [17] is used predominantly for Java projects to provide modularity and reduction of code complexity by promoting code reuse. In

OSGi, the applications are run in a container that provides a basic framework for inter-application communication, runtime handling of modular needs and much more. To run projects in an OSGi container, the code is built as a combination of bundles with an accompanying manifest file that specifies all the requirements of the bundle. Bundles here refer to the application's logic which can either be built from scratch or from a combination of other available bundles.

The advantage of OSGi is that the bundles can be loaded or unloaded at runtime and be stopped, started and restarted without affecting the other running bundles. This a major feature when one considers the time it takes to deploy a solution, test it, unload it, make changes and then reload it again.

Applications are generally built either as consumers or producers of some functionality. When a producer is added to the OSGi container and started, it registers the service it provides with the central service registry. Consumers can then look-up this service and make use of it. This provides a way to build applications from existing technologies without rewriting code. When a service provider is stopped, the OSGi framework informs all the clients and from then on the clients cannot expect to use that particular service.

The glue here is the OSGi framework that provides the Bundle activator, service object initiator, graceful exit procedure for the consumers and producers, communication between applications and dynamic module handling.

3.3 Apache Karaf

There are a number of OSGi containers available such as Equinox, Felix, Eclipse Gemini and Concierge [18]. OpenDaylight uses Apache Karaf [19], a sub-project under Apache Felix, as its OSGi container. There are a number of advantages to using Karaf. The most important feature is that Karaf provides command completion

Figure 3.2: Apache Karaf

similar to that of Linux. An example karaf console is shown in Figure 3.2.

Applications and bundles can be installed and started from the command line presented by Karaf and hot deployment of bundles is also possible by placing the *.jar* files under the *deploy* directory. It is important to have a stable and well supported OSGi container to ensure seamless deployment of applications in the platform and Karaf provides this as it is part of the Apache ecosystem.

3.4 OpenDaylight Bundles

With an overview of the OSGi bundles and the Apache Karaf container, one can understand the way the OpenDaylight sub-projects work with each other. Each sub-project is usually a set of bundles that makes use of the bundles from other sub-projects. For example, the *l2switch* project [20] uses openflowplugin, openflowjava, yangtools and dlux to provide a complete implementation of a Layer 2 Switch. This l2switch application installs flows into any OpenFlow switch which connects to the OpenDaylight controller. The switch can either be a hardware switch or a virtualized

switch created as part of a *Mininet* [21] network. These models are defined in YANG and can be accessed using RESTCONF. The next few sections provide an overview of RESTCONF and YANG.

3.5 RESTCONF

According to the RESTCONF RFC draft [22], RESTCONF is an HTTP based protocol to access models defined in YANG by using the data-stores defined in NETCONF.

3.5.1 NETCONF

Configuring the network devices can be done by CLI (Command Line Interface). However, CLI configuration will become cumbersome as the changes to the network become far too frequent. Network configuration can therefore be automated by scripting the CLI commands. However, this too can become difficult to manage as the scripts are tightly coupled to the CLIs themselves. Any new CLI command or a change to an existing command will break the scripts. To mitigate this, a standard called *SNMP - Simple Network Management Protocol* can be used. SNMP [23] was created to make the configuration and management of network devices generic with the use of *MIB - Management Information Base* modules for each network element.

Although SNMP solves some problems, it is still plagued by a few core issues such as the difficulty in finding *MIB*s, committing configuration changes and use of insecure communication over UDP, as indicated in [24]. To handle the problems mentioned and much more, the *NETCONF* [25] protocol was designed. Some of the advantages of NETCONF over SNMP are:

- Distinction between the configuration and operational data.

- Retrieving data selectively. With SNMP, one has to walk through the whole data-set to get the required information.
- RPC and event notifications [24].

The main aim of NETCONF is to provide a means to configure the network elements in a vendor independent manner. Models of each network device, defined in a modeling language called YANG, can be loaded and changes can be made to them. These changes will then be propagated to the devices themselves.

3.5.2 YANG

Given the advantages of NETCONF it is important to define the models accessed using an expressive language. Although there are a number of contenders such as XML, UML and SMI (used by SNMP), these languages were not inherently created for the sole purpose of configuration management. This is the purpose of the YANG modelling language [26]. YANG was to be used specifically by NETCONF and was developed from the ground up with that aim in view. Some of the advantages of YANG are:

- Support for versions.
- Support for augmentation. This is an important feature as it allows other developers to augment models to suit their needs.
- Specification of Remote Procedure Calls (RPC).
- Grouping common data elements under structure which promotes code reuse.

It is to be noted that YANG, while used to define models for network devices, can also be used to define models of applications that will eventually run on the

OpenDaylight platform. The applications in OpenDaylight do not just use statically generated APIs but rather use APIs generated from the models. This is the foundation of MD-SAL programming, which will be touched upon in the next section. Since the applications defined can either be consumers or producers, access to the services offered or consumed is provided using RESTCONF [22]. The advantage of RESTCONF is that it greatly simplifies the various operations available in NETCONF. It also provides a RESTful API to perform CRUD (Create, Retrieve, Update and Delete) operations. RESTCONF thus hides the details of NETCONF while providing the same functionality but with a REST interface.

3.6 Model Driven-Service Abstraction Layer

With a basic understanding of OpenDaylight, its internal representation of data and the means to access data, we take a closer look at the programming paradigms used in OpenDaylight.

3.6.1 Service Abstraction Layer

Applications in OpenDaylight interact with network elements by employing a communications protocol. In its initial releases OpenFlow was the only protocol that was supported. In order to simplify the use of the southbound protocols a layer called the *Service Abstraction Layer (SAL)* was created. This layer was used to provide a simplified API to the protocol plugins. Any application that wants to talk with a device use SAL API's to invoke requests to the network devices.

This was simple when only OpenFlow was supported. As more protocols like OVSDB, NETCONF, BGP, SNMP, etc., became supported, the APIs started to become too complex. Hence the SAL was redesigned to use models instead of APIs to communicate with the devices. Here the models are tasked with communicating

to the devices and the application only deals with the models which are defined for each protocol plugin that is supported.

3.6.2 Model Centric Applications

Models are not restricted to the southbound plugins. As mentioned before, applications can be producers or consumers. If one application wishes to use the services offered by another application, a contract between the two in the form of an interface used for communication is required. This approach is too unwieldy as the interfaces are not consistent across different applications. To prevent a host of individual APIs, the YANG modeling language is used to generate the models for the applications themselves which would be accessible by using RESTCONF. To support this, the SAL was redesigned and called *Model-driven Service Abstraction Layer (MD-SAL)* [27].

The process of creating an application in OpenDaylight has many steps. To simplify this process there are a number of automated tools available.

3.7 MD-SAL Application

Each MD-SAL application has a number of components that it needs in order to work with Karaf. Instead of starting from scratch, the OpenDaylight developers have created a template which can serve as the starting point. To use this template one must use the *maven archetype* created specifically for this. *maven archetype:generate* is used to get the template from the *Nexus* [28] repositories which holds all the artifacts of the OpenDaylight project. With the template in hand the developers can then begin working on their code.

It can be seen from Figure 3.3 that the primary work of the application developer is to define their application's model using YANG. After that the *yangtools* project takes

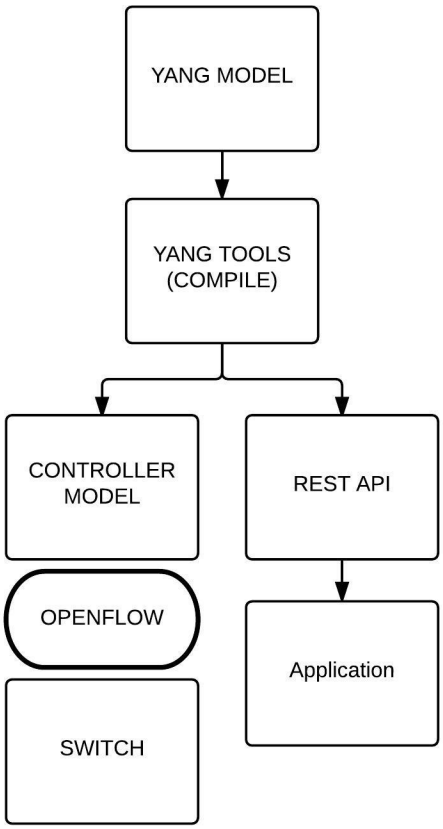


Figure 3.3: MD-SAL Application Generation

the model and generates the RESTCONF API and the models that the OpenDaylight platform needs. The REST APIs that get generated as part of the compilation process are used by programs running outside of OpenDaylight to use the service provided by the application. Additionally, the models that get loaded by OpenDaylight can be used by other applications running inside the platform. This allows the developers to create code that performs a specific task and reuse them as needed later on. For example, to access the nodes in a *Mininet* network, the *flow-node-inventory* YANG model [29] inside the *openflowplugin* project can used.

Once the basic code skeleton is generated by using the YANG tools, one can go in

and code their implementation. There is a file called the *ProjectName*Provider.java which serves as the entry point for all applications. If any RPC's are defined in the YANG model, then the application must register itself as the RPC provider in the central service registry.

An important point to note is the way the data is arranged inside the OpenDaylight platform. The arrangement is explained in the next section.

3.8 Data Trees

OpenDaylight maintains data stores similar to that of NETCONF. The data stores are classified into two based on the type of data maintained in them. They are:

- Configuration Data Store
- Operational Data Store

Configuration Data Store

When configuring the network elements, data that describe the changes to be made on the devices are referred to as configuration data. These are sent to the devices using one of the many communication protocols and applied. In Figure 3.4 the configuration tree structure can be seen. The *nodes* represents all the types of devices that are supported by OpenDaylight. The node may represent a device that uses one of the many available communication plugins. In 3.4 *of:1, of:2 and of:n* represent OpenFlow devices. Each OpenFlow device can have many *tables* and each table can have many *flows*. The requests sent by all applications reside in this tree before they are sent to the devices. If the request is a flow, then the details are placed under the appropriate table in the tree.

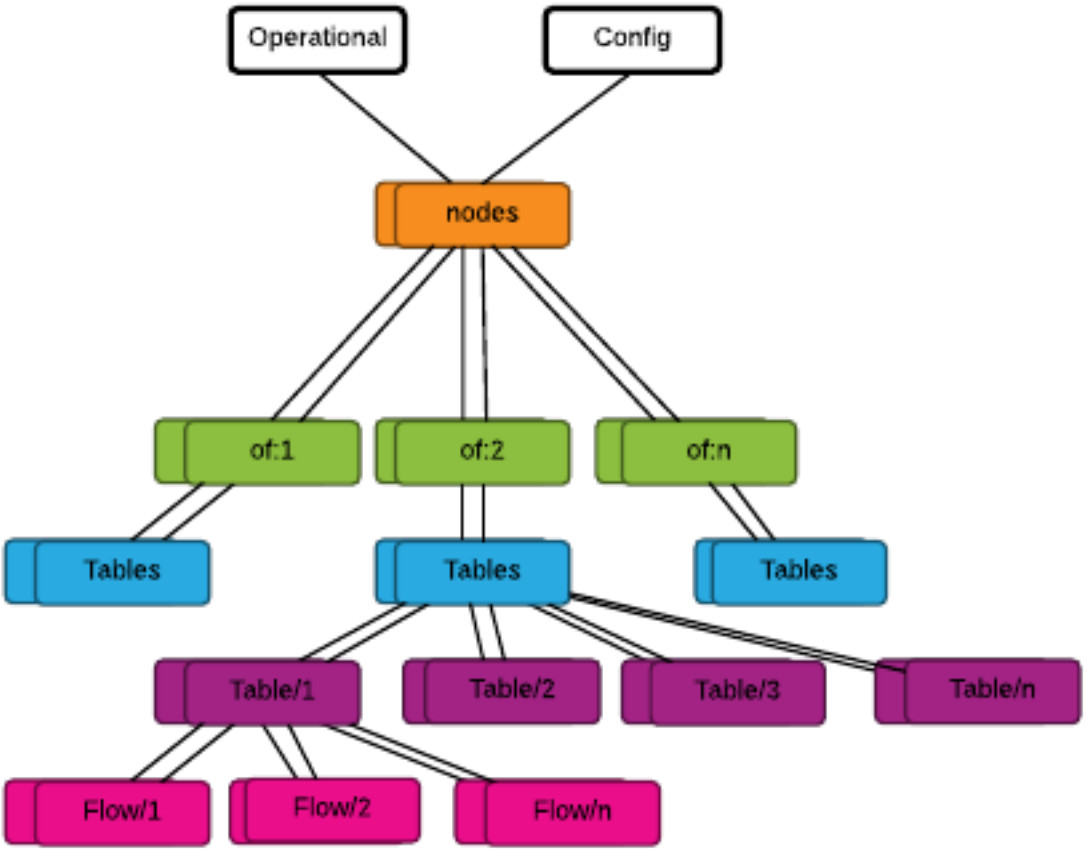


Figure 3.4: OpenDaylight Trees

Operational Data Store

Similarly, the operational data tree can also be seen from the Figure 3.4. It matches the configuration tree in structure and holds the same information. The difference between the two trees is that the Operational tree holds the information that it discovers about the network. The OpenDaylight controller periodically queries all the network elements to get their statistics, flow tables etc., and populates the operational data store with this information. This is done so as to maintain an accurate view of the network.

It may be argued that the configuration tree should suffice and its contents could be considered the configuration that actually exists in the network but there are times when two separate trees help. Consider the scenario where the configuration request arrives at the controller and is forwarded to the network elements. Along the way due to some unforeseen reason a few packets, which are flow add requests, get dropped. With these dropped, the configuration tree will become incorrect. This is the reasoning behind maintaining two different trees.

Data Access

The data in the trees can be accessed in two ways. One way, called *Binding-Aware* format, is where the data and function calls are accessed based on the Java APIs generated during compile time. The other way, called *Binding-independent* format, is where the data and functions are based on a neutral Document Object Model (DOM) notation. When one intends to access any data on one of the trees an *InstanceIdentifier* is required. This is the object that is used to specify the exact node in the tree to access, in a Binding-aware or Binding-independent format.

With an overview of OpenDaylight, its associated tools and technologies, the flow extraction process can be explained in detail.

Chapter 4

FLOW EXTRACTION

The OpenDaylight controller provides RESTful API's to add, update and delete flows. The API's can be accessed through DLUX [12] , a User Interface (UI) subproject under OpenDaylight, or by using any external RESTful client like POSTMAN [30]. The flows that are pushed to the controller using one of the two mentioned methods are then forwarded to the specified switch by using the OpenFlow protocol.

4.1 Flow Listener

OFAnalyzer listens for changes in the flows present in the operational data tree. The operational tree is selected as it represents the view of the network with the configuration applied and not the view of the configuration tree where the configuration has not been applied to the network. To listen for changes, OFAnalyzer first creates an InstanceIdentifier object pointing to the Flow.class under the *opendaylight-flow-types* YANG model [31] which represents OpenFlow flows in the data trees. With this InstanceIdentifier object, OFAnalyzer registers itself as a listener using the *registerDataChangeListener* function found in the *DataBroker*. As the name suggests, DataBroker is responsible for brokering the data between applications and their listeners. An application can register itself as a listener only if it implements the *DataChangeListener* interface and defines the *onDataChanged* function. When registering as a listener, OFAnalyzer specifies the type of change as subtree - a way to be informed of any addition or modification of Flows. There are a number of options available to customize the time when the application is informed of changes. They are:

- one - Listen to only changes to one node and its children.
- base - Listen to changes to one particular node.
- subtree - Listen to changes to any node and its subtree.

```
grouping flow {
    container match {
      uses match:match;
    }
    container instructions {
      uses instruction-list;
    }
    ....
}
grouping match {
    container "ethernet-match" {
      uses "ethernet-match-fields";
    }
    container "ip-match" {
      uses "ip-match-fields";
    }
    choice layer-3-match {
      case "ipv4-match" {
        uses "ipv4-match-fields";
      }
    }
    choice layer-4-match {
      case "udp-match" {
        uses "udp-match-fields";
      }
      case "tcp-match" {
        uses "tcp-match-fields";
      }
    }
    ....
}
```

Figure 4.1: Flow Structure

4.2 Flow Structure

The flows that have been mentioned are structured as shown in Figure 4.1. The structure of the flow is based on the OpenFlow specification [32]. The flow structure is a grouping of the match and list groupings respectively. The match grouping itself is a grouping of structures defined for each layer. This is the way the flow is arranged in the configuration and operational tree. The flow is defined in the YANG file *opendaylight-flow-types* under the *openflowplugin* project. If another project were to use this YANG file, YANG's augment option be used to add functionality without defining the whole structure again. The YANG file when passed to *yangtools* will create the necessary java interfaces under the *api/generated-classes* directory.

4.3 Flow Extraction Algorithm

The application is informed of any change to the subtree by having the onDataChanged function invoked and a Java *Data Transfer Object* (DTO) passed to it. The DTO passed to OFAnalyzer has all the new flows that are added to the flow table to any OpenFlow switch in the network. With the map extracted from the DTO, OFAnalyzer extracts the flows from the entry-set value and stores it in a local HashMap of its own. The algorithm to extract the flow elements is shown in Algorithm 1.

The basic check in each invocation of the onDataChanged call is to see if the Map contains elements of the Flow.class object. This is the class that corresponds to a flow rule. If the element is indeed an *instanceof* flow, the algorithm then steps through each layer, i.e., Layer 2, Layer 3 and Layer 4 to get the respective flow elements. Since the Layer 4 can be either *TCP, UDP* or *ICMP*, the type of Layer 4 protocol is extracted first and then used to further obtain the Layer 4 Source and Destination details. Finally, the action field is extracted, if available, or set to drop since Open

Algorithm 1: Flow Extraction

```
   Data: Java DTO containing flows
 1 Get Map entry − set in DTO as flow;
 2 if flow instanceof Flow.class then
 3 |   Get Priority ;
 4 |   Get L2, L3 and L4 Match objects;
 5 |   if L2Match ≠ NULL then
 6 |   |   Get ETH-SRC and ETH-DST;
 7 |   end
 8 |   if L3Match ≠ NULL then
 9 |   |   Get IP-SRC and IP-DST;
10 |   end
11 |   if L4Match neq NULL then
12 |   |   Get L4Match Type;
13 |   |   if L4Match = TCP then
14 |   |   |   Get TCP-SRC and TCP-DST;
15 |   |   else if L4Match = UDP then
16 |   |   |   Get UDP-SRC and UDP-DST;
17 |   |   end
18 |   end
19 |   Get Instruction object;
20 |   if Instruction ≠ NULL then
21 |   |   Set Action;
22 |   else
23 |   |   Set Drop;
24 |   end
25 end
```

vSwitch does not set an action if the flow action is set to drop during flow installation.

Once the flows have been extracted, each flow is given a specific identifier, making it easier to track the flow when analyzing for conflicts and during visualization transformations in the front-end module. As each flow is stored in the HashMap it is also checked with the previously existing flows in the HashMap for conflicts. The conflict information for each flow is stored as a separate element in the HashMap with each flow rule.

Chapter 5

CONFLICT DETECTION

With the flow rules collected and with its own unique identifier, OFAnalyzer passes it on to the conflict detection engine. This engine is responsible for detecting any conflicts which may exist between each rule as it comes in with all the rules that came in before it.

5.1 Conflict Detection Algorithm

The algorithm to detect conflicts among the rules is shown in Algorithm 2. The process begins with adding the incoming rule to an ArrayList which holds all the rules that came before it. The addition of a rule involves changing the IP addresses, both source and destination, from its dotted notation to an integral value. This conversion helps in setting the range of IP addresses that each particular rule targets. With this range set, it will become easier to check if the IP range of future rules fall within it.

With the conversion performed and each rule added to the ArrayList, the engine can begin locating conflicts. The process begins with a rule selected from the ArrayList. Let this rule be r. r is then compared with all the rules present in ArrayList except r. The comparison is done by taking into account the IP Addresses, protocol, ports, action and the priority.

Let the comparison be between rules r and r' where $r' \epsilon$ ArrayList - r. If r and r' have differing protocols, then there exists no conflicts between the two rules and the loop continues to the next iteration. If the protocols are the same and Priority(r) >Priority(r') then the engine uses the other parameters in its check. The conflict identified is one of 5 types:

Algorithm 2: Conflict Detection

```
   Input: FlowTable f
   Output: Conflict-identified FlowTable f'
 1 for Rule r in f do
 2 |   for Rule r' in f-r do
 3 |   |   if Protocol(r) == Protocol(r') ∧ Priority(r) > Priority(r') then
 4 |   |   |   if Addr(r) ⊂ Addr(r') then
 5 |   |   |   |   if Action(r) == Action(r') then
 6 |   |   |   |   |   Conflict = Redundancy;
 7 |   |   |   |   else
 8 |   |   |   |   |   Conflict = Generalization;
 9 |   |   |   |   end
10 |   |   |   end
11 |   |   |   if Addr(r') ⊂ Addr(r) then
12 |   |   |   |   if Action(r) == Action(r') then
13 |   |   |   |   |   Conflict = Redundancy;
14 |   |   |   |   else
15 |   |   |   |   |   Conflict = Shadowing;
16 |   |   |   |   end
17 |   |   |   end
18 |   |   |   if Addr(r) ∩ Addr(r') ≠ ∅ then
19 |   |   |   |   if Action(r) == Action(r') then
20 |   |   |   |   |   Conflict = Overlap;
21 |   |   |   |   else
22 |   |   |   |   |   Conflict = Correlation;
23 |   |   |   |   end
24 |   |   |   end
25 |   |   end
26 |   end
27 end
```

- Redundancy
- Generalization
- Shadowing
- Correlation
- Overlap

The next sections describe the algorithm followed to identify each type of conflict.

Redundancy

If Priority(r) <Priority(r'), AddressSpace(r) $\subset$ AddressSpace(r') and Action(r) = Action(r') then the type of conflict is detected as *Redundancy*. This is true even if AddressSpace(r') $\subset$ AddressSpace(r).

Generalization

If Priority(r) <Priority(r'), Action(r) $\neq$ Action(r') but AddressSpace(r') $\subset$ AddressSpace(r) then r has a *Generalization* conflict with r'. This is due to the fact that r' represents the packet space of r even though its address space is a subset of r's address space. Any packet that falls in the address space of r will also fall in r' and will use the action of r'. If the priorities are reversed the actions change as well.

Shadowing

If Priority(r) <Priority(r'), AddressSpace(r') $\subseteq$ AddressSpace(r) but Action(r) $\neq$ Action(r') then r is said to be *shadowed* by r'. A rule is completely shadowed since the priority of the rule is less and the address space is a subset of another rule.

Correlation

If Priority(r) <Priority(r'), Action(r) $\neq$ Action(r') but AddressSpace(r) $\cap$ AddressSpace(r') $\not\in \emptyset$ and the AddressSpaces of r and r aren't subsets of each other then the rule r has a *Correlation* conflict with rule r'.

Overlap

If Priority(r) <Priority(r'), Action(r) = Action(r') but AddressSpace(r) $\cap$ AddressSpace(r') $\notin \emptyset$ and the AddressSpaces of r and r are not subsets of each other

then the rule r has an *Overlap* conflict with rule *r'*. The packets that fall in the intersection of the address space of both r and *r'* will have the action of the higher priority.

5.2 Transferring Rules

Once the conflicts have been identified, OFAnalyzer sends this information with each rule to the visualization module. If, for example, rule r has 5 shadowing conflicts, 3 generalization conflicts and 7 overlapping conflicts then the total number of rules that r conflicts with is 15. There are two ways to transfer this information to the visualization module:

- Send all the conflicts with each rule
- Use an encoding scheme

Table 5.1: Rule Conflicts

Rule	Conflicts-with
Rule 1	Rule2, Rule3, Rule4
Rule 2	Rule5, Rule4, Rule3, Rule1
Rule 3	–
Rule 4	Rule1, Rule5, Rule3
Rule 5	Rule2, Rule3, Rule1

5.3 Send All Conflicts

Sending all the conflicts with each rule has some diadvantages. Each rule can have all the rules that it conflicts with appended to its JSON object and transmitted. This

JSON object can then be used by the visualization module to display the relationship between a rule and its conflicts. This scheme works when the rules and its conflicts are small in number. As the number of rules increase, the probability of conflicts among them will also increase. This may exponentially expand the number of rules that has to be transferred with each rule. With this growth, the time and bandwidth required to transfer the rules to the visualization module increases. From Table 5.1, if all the rules are to be transmitted, then each rule is duplicated at least once when sent.

Rule	Conflicts-with
Rule 1	Shadowing:Rule2, Redundancy:Rule3, Shadowing:Rule4
Rule 2	Generalization:Rule5, Overlap:Rule4, Shadowing:Rule3, Shadowing:Rule1
Rule 3	–
Rule 4	Correlation:Rule1, Shadowing:Rule5, Shadowing:Rule3
Rule 5	Overlap:Rule2, Correlation:Rule3, Overlap:Rule1

Table 5.2: Types of Rule Conflicts

5.4 Use Encoding Scheme

Instead of sending all the conflicting rules with a particular rule, an encoding scheme can be used. This takes advantage of the fact that all the rules in the internal flow table are sent to the visualization module.

The unique identifier that is assigned to each rule during the flow extraction process is used. It is enough to just encode the rule identifier in a field called *conflict-type* in each rule's JSON object. This field is semi-colon separated list of unique identifier's which specifies the conflicting rule list.

From Table 5.2 it can be seen that some rules can have multiple rules that have

Type of Conflict	Assigned number
Generalization	1
Shadowing	2
Redundancy	3
Correlation	4
Overlap	5

Table 5.3: Conflict Type Assignment

the same type of conflict. For example, *rule 1* has *shadowing* conflicts with *rule 2* and *rule 4*. To be able to extract this information in the front-end without any confusion, the conflict type is encoded as well.

The encoding scheme works as follows:

- Use the assignment from Table 5.3.
- Specify the type of conflict and the rule, using an $x.y$ notation where x is the conflict type and y is the unique identifier of the rule.
  - For example, shadowing conflict with rule x which has unique identifier 3 is encoded as 2.3
- Encode using the $x.y$ scheme for each rule in conflict list.
- Use a semi-colon to separate each unit.
  - This separates units during processing in the visualization module.

Given this encoding scheme the conflict information in Table 5.2 is encoded as in Table 5.4.

Rule	Conflicts-with
Rule 1	2.2;3.3;2.4
Rule 2	1.5;5.4;2.3;2.1
Rule 3	–
Rule 4	4.1;2.5;2.3
Rule 5	5.2;4.3;5.1

Table 5.4: Encoded Table

Once each rule has its own conflict information encoded in the *conflict-type* field, the resulting rule is stored in a HashMap. After this step, the conflict detection engine passes the map back to the main application.

5.5 Compilation Process

The flow extraction module and the conflict detection engine together make up the OFAnalyzer back-end. The application with all its logic has to be compiled and loaded onto the OpenDaylight platform to execute. OpenDaylight uses *maven* [33], an Apache build manager for Java projects. Using *maven* to compile and create the executable JAR requires some changes to the basic project skeleton that is generated using the *maven:archetype* command [34].

Since OFAnalyzer uses the code from *opendaylight-model-base* to listen to flows, this module must be referenced in the *features.xml* file. The purpose of this file is to let OpenDaylight know of the requirements of an application. Before OFAnalyzer is loaded onto the platform, the modules listed in the *features.xml* file are loaded first. This ensures that no consumer runs until all the producers that it depends on are loaded. This step can be skipped provided the module dependencies are taken care of before loading OFAnalyzer.

Once the code is compiled, the jar files are located under the maven *m2* directory. The application can be imported into the OpenDaylight platform by using the *mvn:repo-add* command. This is used to bring the jar, from the m2 directory and install all the dependencies in the features.xml file. After this process, the application can be installed using the *mvn:install* command.

The work done by the back-end to identify the conflicts is used by the visualization module to display the relationship between the rules. The next chapter takes a detailed look at the visualization module.

Chapter 6

CONFLICT VISUALIZATION

The visualization module takes the rules with their conflict-type details and displays them in a manner that is both intuitive and concise. A number of works have focused on identifying the conflicts present in the flow table but have not looked at displaying this information in a suitable format.

OFAnalyzer has a front-end user interface which is a stand alone application under the DLUX project. This project is home to a number of UI modules that provide information to the user. The developers of the DLUX project have created an *archetype* that allows developers to add modules to the project. Essentially they have created a platform that can be extended by developers. OFAnalyzer takes advantage of this fact and uses the libraries and support provided by DLUX to visualize the data collected by the back-end. This has the added advantage of being compliant with the OpenDaylight platform while adhering to the best practices followed in UI design. Hence, the need to create a separate front-end outside of the OpenDaylight environment is not necessary. This saves a lot of work that is done when setting up a server to host the webpage and ensuring that the front-end runs only when the back-end is available.

The visualization in OFAnalyzer involves four steps. They are:

- Fetch rules from the back-end.
- Process retrieved rules.
- Apply transformations.
- Display rules using visualization techniques.

– Tabular form with rule detail

– Reingold-Tilford tree with hierarchical edge bundling for rule relationship

6.1 Fetch Rules

The first task that OFAnalyzer performs on initialization is to try and fetch the rules present in the back-end HashMap. To perform this task the OFAnalyzer uses the REST API that has been created specifically for this application. The REST API on return provides a list of rules with all the information gathered during the *flow extraction* and *flow conflict detection* phases. The rules are also fetched after initialization using a WebSocket listener.

6.2 WebSocket Listener

OpenDaylight allows applications to subscribe to data change event notifications. Notifications are generated when there is a change on either one of the two trees in OpenDaylight. This feature is taken advantage of by OFAnalyzer to get the flow details in real-time i.e. as and when they arrive at the back-end. However, it is not feasible to listen to the flow nodes on the tree directly since these nodes get updated frequently with statistics and other miscellaneous data.

Hence, the location that the front-end listens to is a node in the configuration tree that will be updated by the back-end. This node is the synchronization mechanism that is used between the front-end and the back-end. The node is updated only after all extraction and conflict detection phases run to completion for the new flows.

Use of the OpenDaylight change event notification feature involves the following steps:

- Create a stream by invoking the stream creator

  - Use the REST API */restconf/operations/sal-remote:create-data-change-event-subscription*
  - Output of the REST call will be the created stream named *stream-name*
- Subscribe to the created stream using REST call */restconf/streams/stream/*
  - The output of the REST call will be the *WebSocket location* that can be parsed from the header of the response
- Use the AngularJS [35] websocket listener to listen to the WebSocket location

With this setup, the new flows arrive at the front-end automatically and get passed to the AngularJS controller for further processing.

6.3 REST API

The REST APIs, apart from the ones used to create the WebSocket listener, are generated by the yangtools as it compiles the back-end application. The basic structure of the REST RPC is specified using YANG as follows:

- Input
  - Specify the name of API call
  - Append the type of conflict
- Output
  - List of rules with conflict details

This REST RPC is not restricted to the OpenDaylight platform. Using *curl* [36], *POSTMAN* and other REST clients the call can be made and the output can be processed as required.

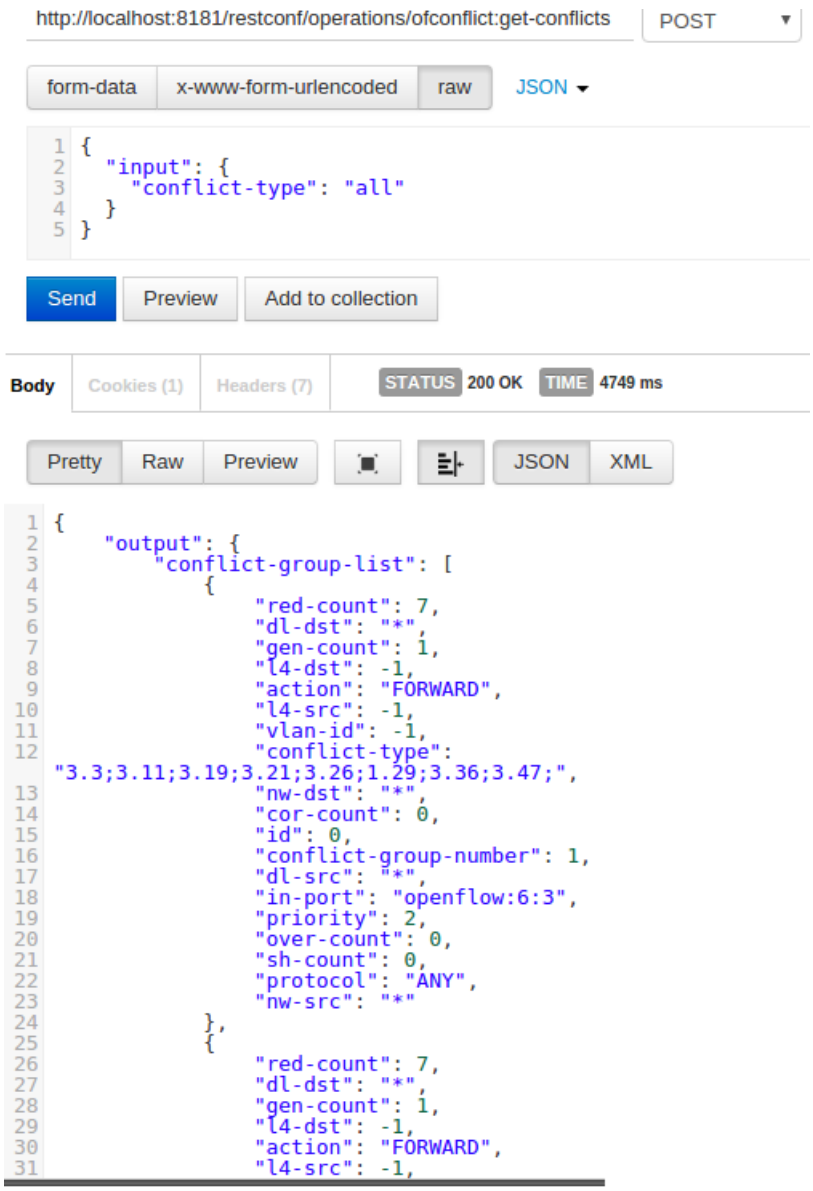


Figure 6.1: POSTMAN REST Invocation

With the API defined and type of conflict information required appended to the API call, the REST call can be made to the OpenDaylight platform. The value returned by the call is a *JSON object* that contains a list of all the flow rules. An example of the retrieved flow rules using *POSTMAN* is shown in Figure 6.1. The name of the API call can be seen at the top of Figure 6.1. OFAnalyzer has only one API call - *get-conflicts*. The type of conflict required is specified in the raw JSON field as *input*. It can be seen from the bottom part of Figure 6.1 that the *conflict-type* field consists of the semi-colon separated encoded values of the conflicts detected by the conflict detection engine. This field is integral in displaying the relationship between the rules. However, they cannot be used in this format.

6.4 Process Retrieved Rules

With the encoded list of conflicting rules available at the front-end, the *decoding* phase can begin. The decoding phase is where the rules are converted from an $x.y$ notation into an array of rules. This process involves the use of a string splitter to first identify the type of conflict from the $x.y$ notation and appending that data to a list which will hold all the details of that particular rule. This list will then be held in a multidimensional array for future use. For example, the values under rule 4 from Table 5.4 in page 35 will be converted as per the following:

- Create array in the multidimensional array with index = rule.id
- Start processing $x.y$ elements from *conflict-type* field for rule 4
- For each $x.y$, retrieve conflict type = Map(x)
- Map converts the integer into a String indicating the type of conflict
- Iterate through the whole *JSON* list received to get the rule where

  - y is the rule's identifier

- Append identified rule to the created array with the conflict information listed as the one identified

At the end of this process, each rule will have its own array containing all its conflicts. Additionally, the appropriate conflicting information is also appended. Once this array has been filled, the tabular form of visualization can be displayed. To perform rule relation visualization the rules present in the output from the REST call and the multidimensional array holding the conflicting rules will have to be converted into a different format.

6.5 Transforming the Rules

The visualization of rule relationships require the use of *D3.js*, a JavaScript library from *Bostock* [37].

6.5.1 D3 - JavaScript Library

D3 is JavaScript library that allows binding data to the elements of a document. The data can be bound to the elements of a DOM and styled using CSS, SVG, etc. The advantage of D3 is the ease of modifying the elements of a document using its built-in API. The APIs are extremely useful in creating DOM elements based on the data. An example is creating svg elements in the body element based on the dataset. If the dataset is [1,2,3,4,5] then 5 svg elements can be added using a loop over the dataset.

This advantage of using datasets and binding them to the elements make D3 a valuable library to have when attempting to visualize the flow rules and the relationship among them. To leverage the inherent power of D3, the data fed to its

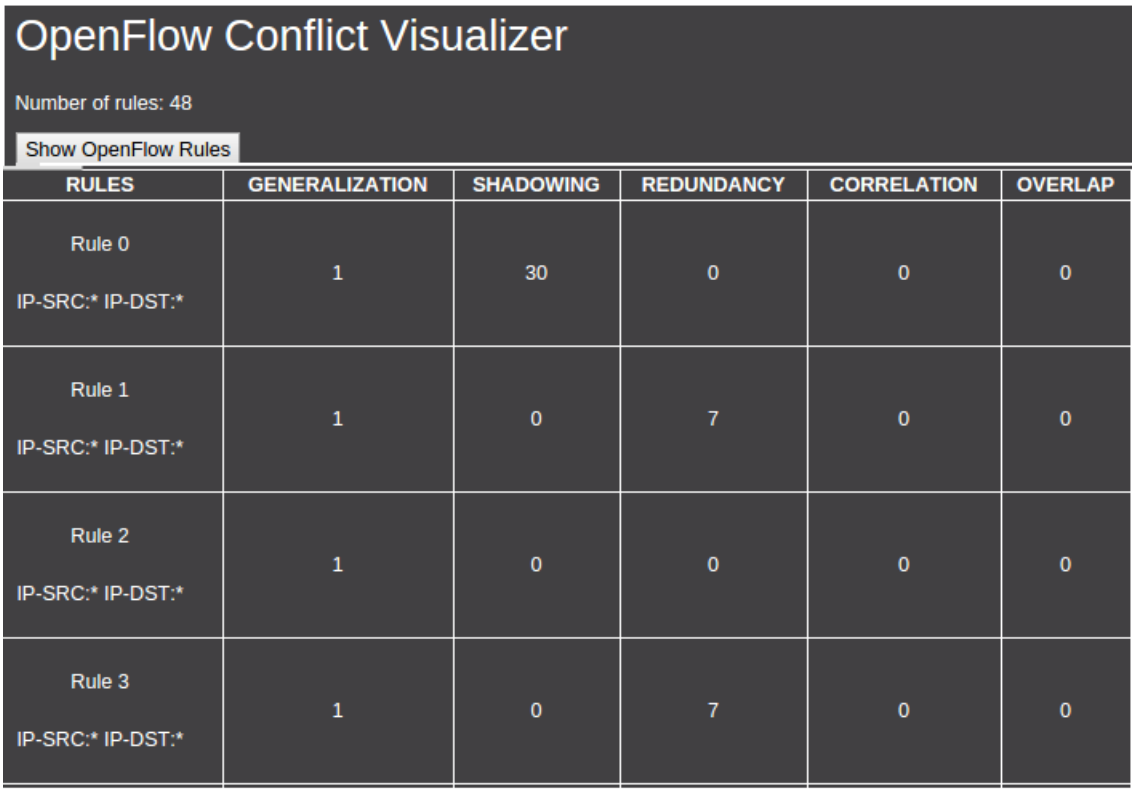


Figure 6.2: Landing Page

API must be formatted in a particular way. For example, the Reingold-Tilford tree API will take in a list of JSON objects in a parent-child format and append (x,y) co-ordinates that satisfy the Reingold-Tilford algorithm [38]. The Reingold-Tilford algorithm helps in creating tidy trees that can be used to display relationships. The relationship depends on the relative position of the nodes in the tree.

The flow table rules, however, are not ordered in a parent-child relationship. OF-Analyzer transforms the JSON flow rule list procured from the back-end along with the multidimensional array into a format suitable to be used with D3's API. Once the transformation step is finished, the data can be fed into the routines that perform the visualization.

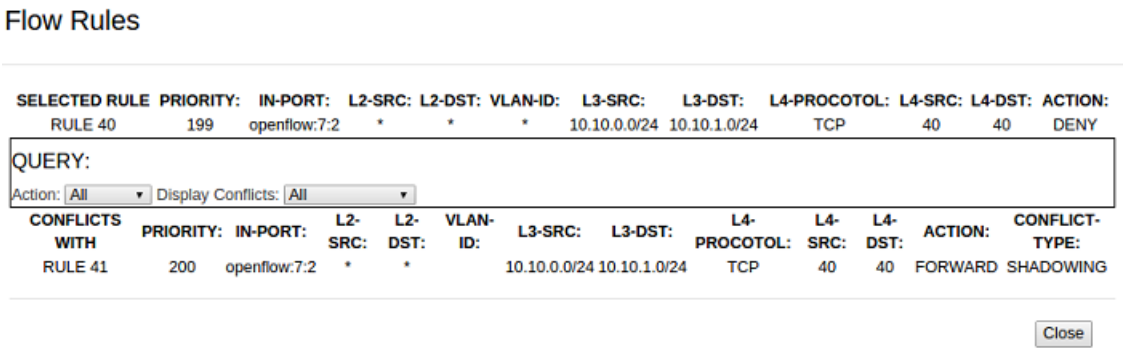


Figure 6.3: Tabular View

6.6 Visualization of OpenFlow Rule Conflicts

The visualization of OpenFlow conflicts in OFAnalyzer is done in two ways. The first provides a detailed view of the flows displayed in a tabular format with controls to filter the table. The network administrator would, at times, prefer to have a closer look at the flow entries. This tabular view is useful in such scenarios. The second is to highlight the overall relationship between all the flow entries in the table. This is achieved by making use of a hierarchical edge bundle [39] and a Reingold-Tilford tree. Hierarchical edge bundle is a generic method of bundling adjacent edges together to reduce clutter. This is useful in a list where there are multiple edges between flow rules. A partial view of the OFAnalyzer landing page is shown in Figure 6.2. Here, the number of conflicts of each type for each rule is listed in a tabular format. The tabular view is obtained by clicking on any of the rules and the relationship view is navigated to by clicking on the appropriate tab.

6.6.1 Tabular Visualization

The tabular view presented to the network administrator is shown in Figure 6.3. This view provides all the details with regards to a particular flow. The rule that is

selected from the landing page is placed at the top. Below this are the query controls that allows a user to filter the conflict list that follows. The query controls include filtering the type of conflict or action to view. Overall, a detailed look of a rule and its associated conflict list is shown in this tabular view.

6.6.2 Hierarchical Edge Bundle

The relationship between the rules are shown using a *hierarchical edge bundle*, where links are drawn between the nodes arranged in a circle. This structure is ideal when attempting to display the relationship of each rule with the other rules present in the flow table. An example of the hierarchical edge bundling structure is shown in Figure 6.4. When the mouse hovers over a particular rule, all the rules that have a conflicting relationship with it are highlighted. The color of link depends on the relationship between the two endpoints.

Red Links

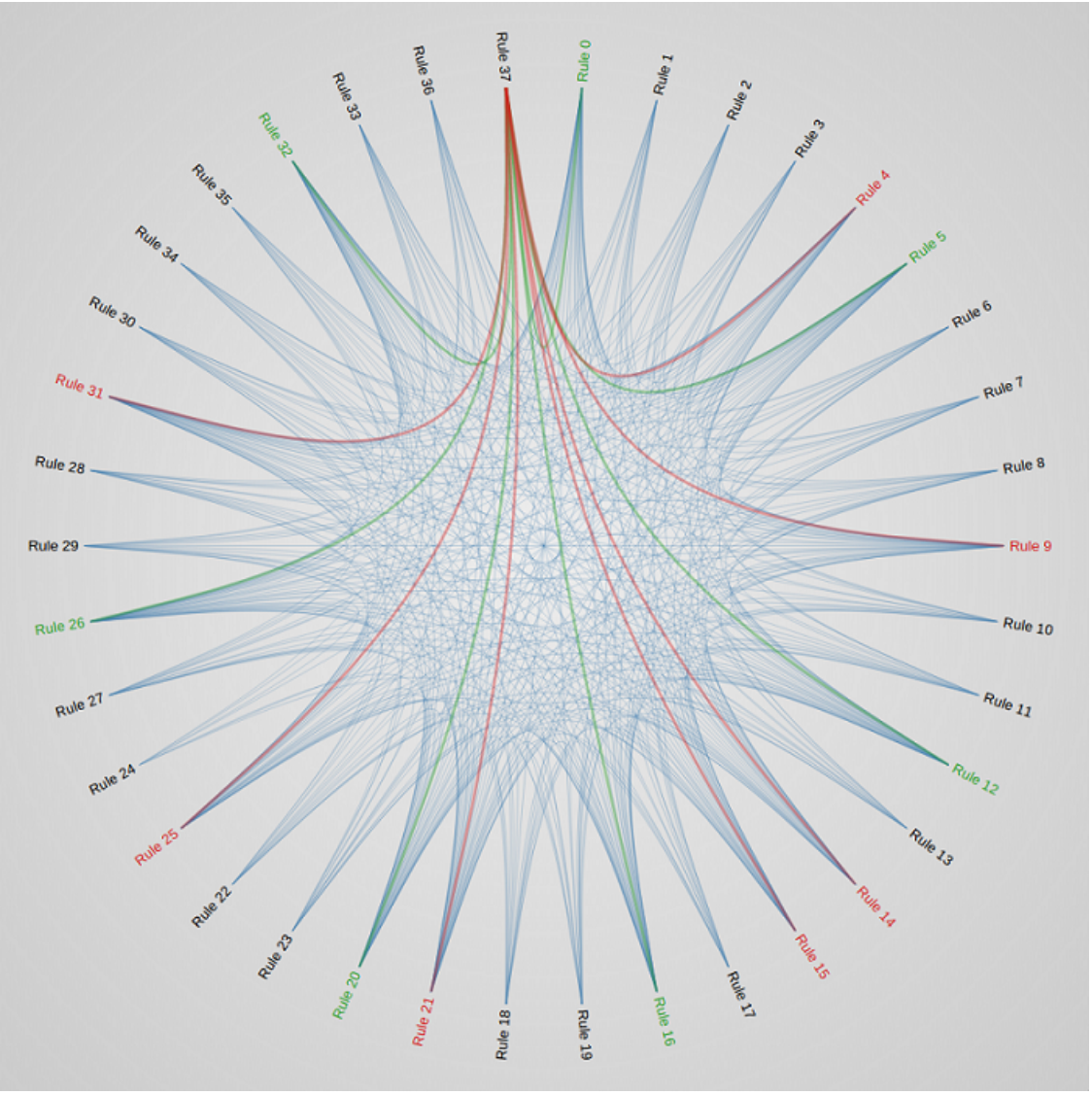
If rule x has a shadowing conflict with rule y, then rule y is included in rule x's conflict list. This relation is shown by having a red line from rule x to rule y. A red link indicates that the rule that is currently under the mouse has the other rule in its conflict list.

Green Links

Conversely, rule y does not have rule x in its conflict list. To show that rule y is included in rule x's conflict list a green line is used between them. Hence a green link indicates that a rule , though it may not necessarily have conflicts with others, is included in the conflict list of other rules.

Finally, this relationship is further explored by clicking on a rule on the circle.

Figure 6.4: Hierarchical Edge Bundling



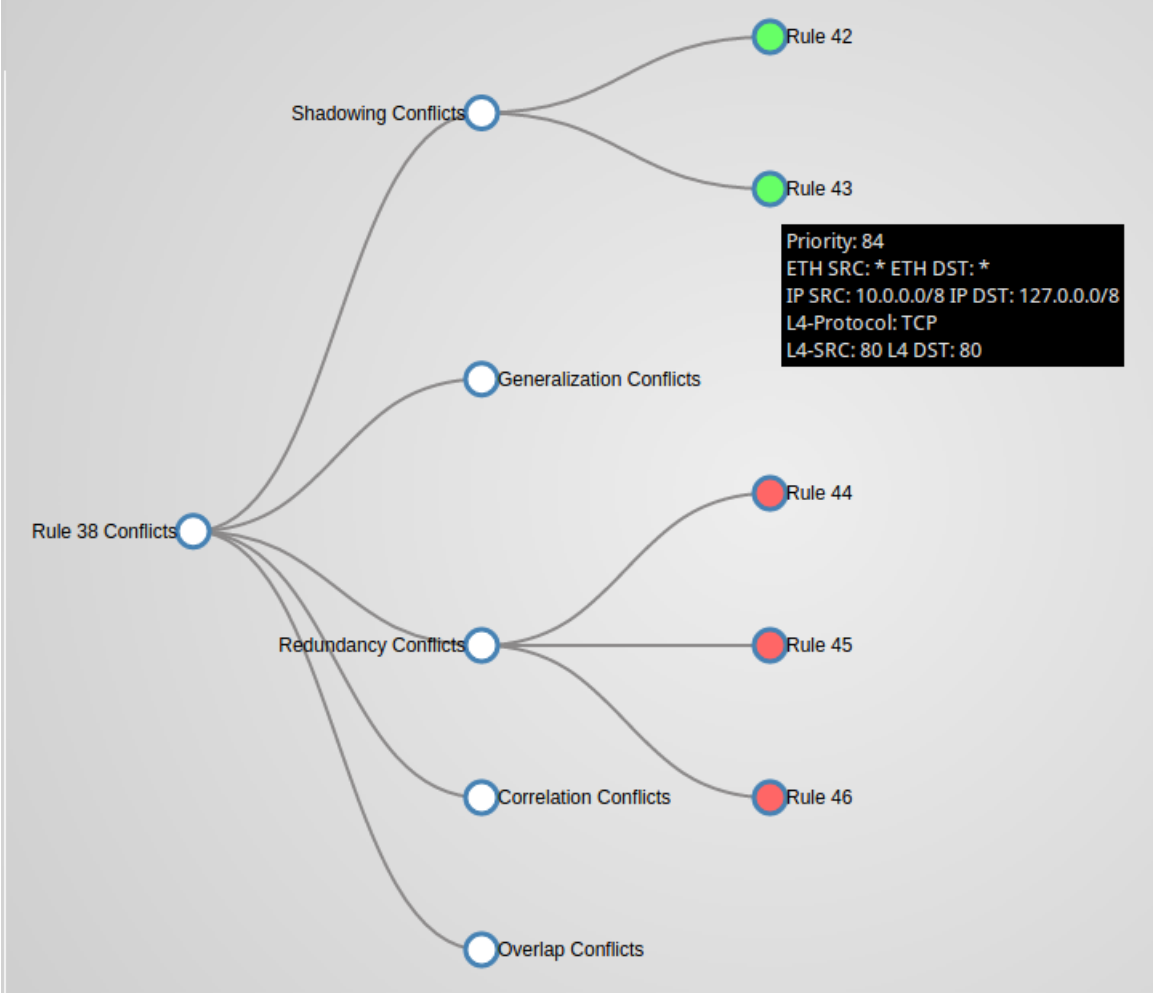
The click will display another page where the Reingold-Tilford tree is employed.

6.6.3 Reingold-Tilford Trees

Reingold and Tilford in [38] describe an algorithm that can be used to create an aesthetically pleasing and tidy drawing. D3's API can be used to generate these Reingold-Tilford trees by just specifying the data in an appropriate format. The API

also possesses additional functionality which allows the tree to become interactive. The appropriate format expected by the API can be constructed by having JavaScript routines work on the retrieved JSON flow list. An example of the Reingold-Tilford tree is shown in Figure 6.5.

Figure 6.5: Reingold-Tilford Tree



Since the tree is interactive, a click on the conflict types will hide the corresponding nodes. The users can view a particular type of conflict alone if they so desire. The advantage of this is scalability. If there are a large number of nodes under each conflict, the users can choose to hide all conflicts that are of no interest to them.

With just the tree, the details of each rule are not apparent. To enable a quick view of the details of a rule, a hover box is used. The content of a hover box is illustrated in Figure 6.5.

Chapter 7

EVALUATION

OFAnalyzer was evaluated for correctness by providing it with a number of rules that were known to have conflicts. The tool was able to identify the conflicts and display the relationship using the visualization techniques discussed in the preceding chapters.

The flows that were presented to OFAnalyzer are shown in Table 7.1. The topology used in the evaluation is seen in Figure 7.1. The relationship between the rules is captured in Figure 7.2. When Rule 40 (Rule 9 in Table 7.1) is clicked the output is

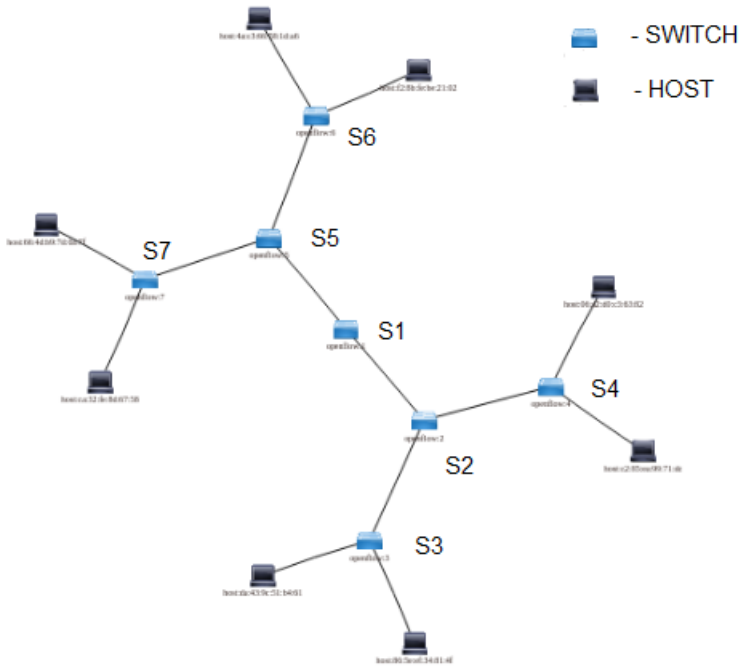


Figure 7.1: Test Topology

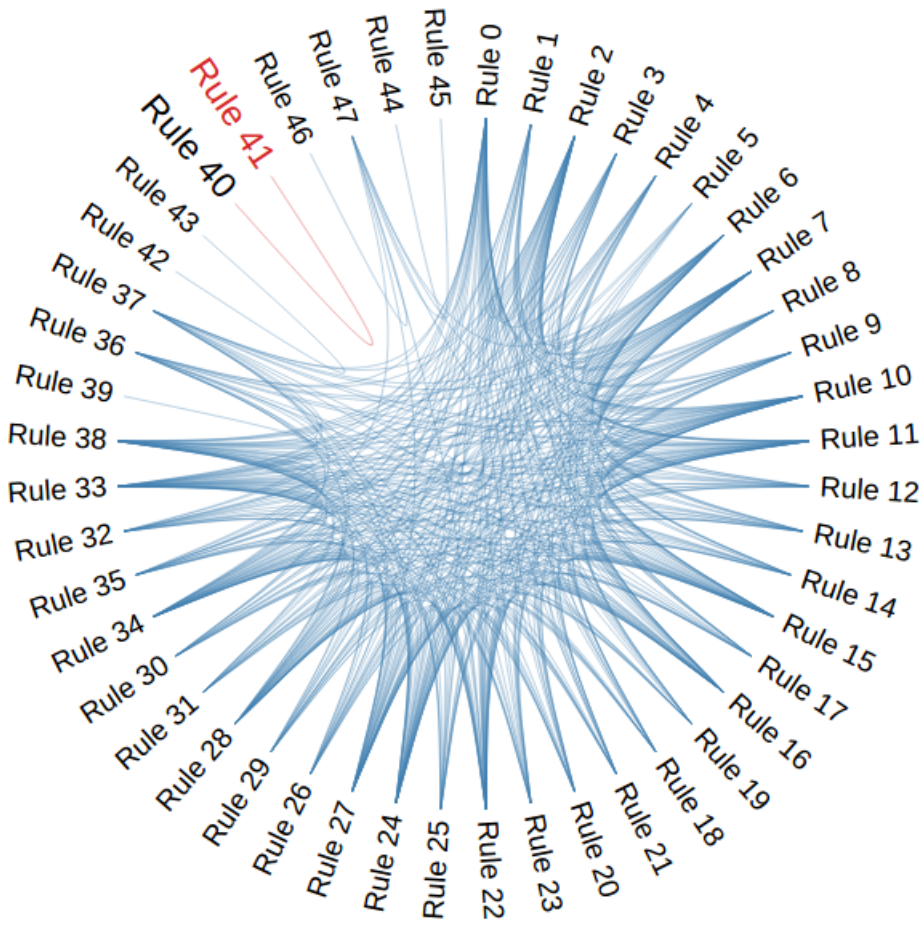


Figure 7.2: Test Flows Landing Page

shown in Figure 7.3.

The evaluation was carried out in a virtual machine having the following configuration:

- Intel Core i7 with 2 dedicated cores
- Hardware virtualization enabled (VT-x)
- 4GB memory
- Linux kernel 3.1

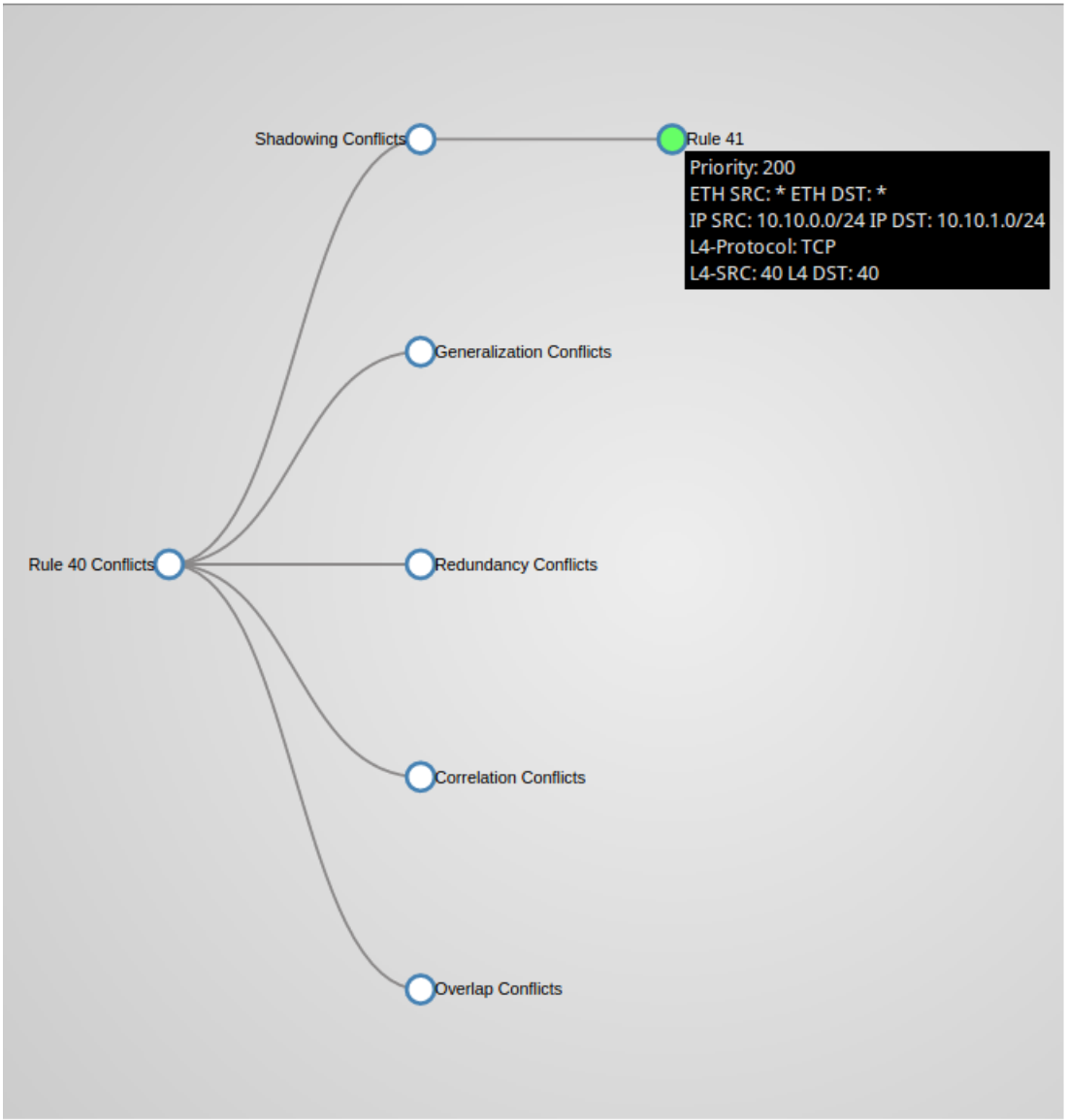


Figure 7.3: Reingold-Tilford Relationship

Rule	IP-Src	IP-Dst	L4-Src	L4-Dst	Action	Conflict Type
1	14.14.0.0/24	14.14.0.0/24	40	40	Accept	Overlap(2)
2	14.14.0.0/32	14.14.0.0/32	*	*	Accept	-
3	13.13.0.0/24	13.13.0.0/24	60	60	Accept	Correlation(4)
4	13.13.0.0/24	13.13.0.0/24	60	60	Deny	-
5	12.12.0.0/16	12.12.1.0/16	41	41	Deny	Generalization(6)
6	12.12.0.0/24	12.12.1.0/24	41	41	Accept	-
7	11.11.0.0/24	11.11.1.0/24	63	63	Accept	Redundancy(8)
8	11.11.0.0/24	11.11.1.0/24	63	63	Accept	-
9	10.10.0.0/24	10.10.1.0/24	40	40	Deny	Shadowing(10)
10	10.10.0.0/24	10.10.1.0/24	40	40	Accept	-

Table 7.1: Test Flows

- Xubuntu Operating System 64-bit

The network was generated using *Mininet* [21]. The generated network had 7 switches and equal number of hosts. OpenDaylight Lithium was used as the OpenFlow controller and the *L2Switch* project was employed to connect to the Mininet Open vSwitch(OVS) switches. OVS and OpenDaylight Lithium both support OpenFlow 1.0 and OpenFlow 1.3. The L2Switch project uses the *openflowplugin* to connect to the OVS using the OpenFlow protocol.

Chapter 8

CONCLUSION AND FUTURE WORK

8.1 Conclusion

As seen from the evaluation work carried out, OFAnalyzer is able to identify the conflicts and visualize them using the different views present in the front-end. The tabular view presents a detailed look at each flow and its related conflicts and the relationship between the flows is displayed using the hierarchical edge bundle and the Reingold-Tilford tree.

Although a large number of research works have focused on conflict detection, not much has been done in the visualization of the conflicts. This thesis provides a number of ways to display the conflicts using various visualization techniques. Its correctness is also tested to see if the conflicts that are identified are displayed properly.

As can be seen from the examples provided, OFAnalyzer is very useful when the relationship between the conflicting rules is to be understood.

8.2 Future Work

As mentioned in the flow extraction process, OFAnalyzer listens to the operational data tree of OpenDaylight. This tree will have the details of all the flows of the network under OpenDaylight. The tree is populated after the flows have been communicated to the switches.

In the future, the ability to stop the flows from being sent to the switches could be added. This will allow OFAnalyzer to look at the incoming flows, compare them with the existing flows to make an allow or reject decision on the new flow. To do this,

a sub-module could be added to the back-end which will listen to the configuration data tree and compare the flow to be added with the flows in both the trees.

Additionally, the front-end could also be upgraded to provide newer features to assist in scalability. A zoom-in zoom-out feature aiding in the visualization process and graphs depicting the statistical data gathered from the switches using the OpenFlow protocol could be added.

Moreover, the conflict detection engine could be updated to use an algorithm that is more efficient than the currently used implementation that has a time complexity of $O(n^2)$, where n is the number of flows in the flow table.

Finally, the work done in this thesis has been submitted as part of [40] and [41] for publication.

REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[2] A. Mayer, A. Wool, and E. Ziskind, "Fang: A firewall analysis engine," in *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*. IEEE, 2000, pp. 177–187.

[3] E. S. Al-Shaer and H. H. Hamed, "Firewall policy advisor for anomaly discovery and rule editing," in *Integrated Network Management, 2003. IFIP/IEEE Eighth International Symposium on*. IEEE, 2003, pp. 17–30.

[4] H. Hu, G.-J. Ahn, and K. Kulkarni, "Fame: a firewall anomaly management environment," in *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*. ACM, 2010, pp. 17–26.

[5] E. Al-Shaer and S. Al-Haj, "Flowchecker: Configuration analysis and verification of federated openflow infrastructures," in *Proceedings of the 3rd ACM workshop on Assurable and usable security configuration*. ACM, 2010, pp. 37–44.

[6] L. Yuan, H. Chen, J. Mai, C.-N. Chuah, Z. Su, and P. Mohapatra, "Fireman: A toolkit for firewall modeling and analysis," in *Security and Privacy, 2006 IEEE Symposium on*. IEEE, 2006, pp. 15–pp.

[7] S. Natarajan, X. Huang, and T. Wolf, "Efficient conflict detection in flow-based virtualized networks," in *Computing, Networking and Communications (ICNC), 2012 International Conference on*. IEEE, 2012, pp. 690–696.

[8] P. Gupta and N. McKeown, "Algorithms for packet classification," *Network, iEEE*, vol. 15, no. 2, pp. 24–32, 2001.

[9] T. Tran, E. S. Al-Shaer, and R. Boutaba, "Policyvis: Firewall security policy visualization and inspection." in *LISA*, vol. 7, 2007, pp. 1–16.

[10] E. Fang, "The power rank," http://thepowerrank.com/ncaa-tournament-predictions/, 2016.

[11] A. C. C.-J. C. Sandeep Pisharody, Abdullah Alshalan and D. Huang, "Brew: A conflict-free flow rule appliance for sdn environments," 2016, aCM.

[12] E. K. Harman Singh, Maxime Millette Coulombe and M. Venugopal, "Welcom to the opendaylight ux dlux project," https://wiki.opendaylight.org/view/OpenDaylight_dlux Main, September 2013.

[13] "The opendaylight project," https://www.opendaylight.org/, 2010.

[14] "Sdn controller comparison part 1: Sdn controller vendors," https://www.sdxcentral.com/resources/sdn/sdn-controllers/sdn-controllers-comprehensive-list/, 2012-2016.

[15] "Opendaylight project hosted in gerrit mirrored in github," https://github.com/opendaylight/, 2013.

[16] "Open services gateway initiative architecture," https://www.osgi.org/developer/architecture/, 2014.

[17] S. Patil, "Osgi for beginners," http://www.javaworld.com/article/2077837/java-se/hello–osgi–part-1–bundles-for-beginners.html, March 2008.

[18] "Choosing an osgi distribution: Equinox, felix, gemini or others," http://blog.webforefront.com/archives/2010/10/choosing_an_osg.html, October 2010.

[19] "Apache karaf," http://karaf.apache.org/, 2016.

[20] "L2switch project," https://github.com/opendaylight/l2switch, May 2014.

[21] B. Lantz, B. Heller, and N. 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*. ACM, 2010, p. 19.

[22] A. Bierman, M. Bjorklund, K. Watsen, and R. Fernando, "Restconf protocol," *IETF draft, work in progress*, 2014.

[23] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "Rfc 1157: Simple network management protocol (snmp)," 1990.

[24] "What is netconf," http://www.tail-f.com/education/what-is-netconf/, 2011, overview of NETCONF and SNMP.

[25] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman, "Rfc 6241, network configuration protocol (netconf)," 2011.

[26] M. Bjorklund, "Yang-a data modeling language for the network configuration protocol (netconf)(ietf rfc 6020)," 2010.

[27] "Model driven-service abstraction layer," https://github.com/opendaylight/mdsal, 2016.

[28] "Nexus opendaylight repository holding outdated and recent artifacts," http://nexus.opendaylight.org/content/repositories/opendaylight.snapshot/, 2016.

[29] "Flow capable node inventory yang model in openflowplugin," https://github.com/opendaylight/openflowplugin/blob/master/model/model-flow-service/src/main/yang/flow-node-inventory.yang, March 2013.

[30] "Postman supercharge your api workflow," https://www.getpostman.com/, 2013.

[31] "Openflowplugin model for flow service offered in opendaylight," https://github.com/opendaylight/openflowplugin/blob/master/model/model-flow-base/src/main/yang/opendaylight-flow-types.yang, October 2013.

[32] O. O. S. Specication, "Version 1.3. 0 (wire protocol 0x04)(june 25, 2012)."

[33] "Maven in 5 minutes," https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html, March 2016.

[34] "Introduction to archetypes," https://maven.apache.org/guides/introduction/introduction-to-archetypes.html, March 2016.

[35] "Angularjs by google," https://angularjs.org/, 2016.

[36] "curl.1 manpage," https://curl.haxx.se/docs/manpage.html, 2016.

[37] M. Bostock, "D3. js," *Data Driven Documents*, 2012.

[38] E. M. Reingold and J. S. Tilford, "Tidier drawings of trees," *Software Engineering, IEEE Transactions on*, no. 2, pp. 223–228, 1981.

[39] D. Holten, "Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 12, no. 5, pp. 741–748, 2006.

[40] S. Pisharody, J. Natarajan, and D. Huang, "Of security policy conflicts in large scale distributed software defined network cloud environments," March 2016, submitted in HotCloud 16.

[41] ——, "Security policy conflict resolution in sdn based cloud environments," March 2016, submitted in Special Issue: Cloud Security.