FLOWOPS: OPEN ACCESS NETWORK MANAGEMENT AND OPERATION

by

Matt Strum

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Science

Computing

The University of Utah

December 2013

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

STATEMENT OF THESIS APPROVAL

The thesis of Matt Strum has been approved by the following supervisory committee members:

Robert Ricci	, Co-Chair	August 20 2013 Date Approved
Jacobus Van der Merwe	, Co-Chair	May 10, 2013 Date Approved
Sneha Kumar Kasera	, Member	May 10, 2013 Date Approved

and by Alan Davis , Chair/Dean of the Department/College/School of Computing

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

Arguably, the inherent complexity of network management makes it the top concern for network operators. While true for all networks, network management complexity is significantly exacerbated in open access networks where, unlike more monolithic "closed access networks," services are provided by different service providers on a shared network infrastructure that is operated by a separate network owner/operator. The intricate responsibilities of the role players in this environment, combined with the lack of automation in current network management and operation practices, conspire to prevent open access networks from reaching their true potential. In this thesis, we present our work on the *FlowOps* framework to address these concerns.

FlowOps is a network management and operations framework that provides structured, automated network management for heterogeneous open access network environments. In FlowOps, we are exploring the use of a production rules system to realize automated network management and operations. This rule-based approach enables us to accurately model dependencies and relationships of devices and role players in an open access network. FlowOps enables the automation of network configuration and fault management tasks in both traditional and software-defined networks. We present a prototype implementation of FlowOps and demonstrate its utility for network operators, service providers, and end users.

For my wife Jolin.

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

First off, I would like to thank Robert Ricci for working with me for so many years and providing me the opportunity to be a part of this project. Equally important, I would not be on this project without Jacobus (Kobus) Van der Merwe, who joined the Flux Research Group just in time to start this project and offer me the chance to build the foundation of what appears to be an important step in the right direction.

I cannot forget my wonderful parents, who guided me to learn self-discipline and the importance of always improving oneself. My wife and best friend, Jolin, stood by my side as I worked feverishly to complete my thesis and always motivated me to get work done.

Last but not least, I would like to thank EntryPoint LLC for funding the project as well as Jeff Christensen and Robert Peterson for providing feedback along the way.

CHAPTER 1

INTRODUCTION

Managing networks, especially shared networks, is challenging. Network operators have to worry about fault management, changing configurations without affecting existing services, adding new features, user setup and removal, etc. Many of these problems commonly require human intervention due to a lack of integrated and automated management tools that have knowledge in all aspects of the network. For example, the tools that detect errors may not have any knowledge of what services and users are affected. A problem may go unnoticed until a customer calls and reports a problem after which a technician must diagnose the issue. There are tools which can assist in these types of scenarios, but most lack the sophistication for truly automated network management.

Traditional network devices have contributed largely to this problem. Each vendor creates proprietary management configuration interfaces which use different terms and support different feature sets. This causes the need for domain experts to read through pages of explanation of how to configure and manage each network device they deal with.

Software-defined networking (SDN) is poised to solve many of the frustrations plaguing traditional networks. Instead of completely different control interfaces like those available in traditional networks, SDN solutions extract the control plane and place the logic in a centralized controller which communicates with network devices using a common protocol to configure the data path. While optimizing the control plane has been the focus so far, much work is still needed to provide integration and automation between tools to provide a richer framework allowing network operators to more efficiently manage all aspects of their network.

On top of choosing what control interfaces to use, network operators also must choose protocols for networking at the edges and within the backbone. These decisions can influence what network devices the network operator must buy or be influenced by what is available already since network devices often only support a subset of available protocols.

Protocols deployed in the infrastructure determine what services can be configured through the network, which makes this choice a highly important one.

Setting up and actively managing networks is extremely complicated and the abundance of control interfaces and protocols heavily contribute to the problem. If a network operator wants to allow external service providers to configure services within their own infrastructure, these complications quickly multiply due to multitenant environment challenges. Networks where service providers separate from the network operator are able to configure services through the network are called open access networks. Closed networks, like those deployed by cable companies, have services provided only by the network operator. Figure 1.1 visualizes the differences between closed and open networks. A management framework which solves the issues seen in closed networks is already extremely useful, so one which also provides the ability to run an open access network becomes even more valuable.

Open access networks are often part of government broadband initiatives and typically involve home owners buying a physical network connection to the operator network. This low latency, high capacity connection then effectively becomes part of the home's amenities and the home owner separately orders networking services from (potentially a number of different) service providers that operate on this infrastructure. Many municipalities and some private companies are moving to provide high-speed network infrastructure to users while encouraging innovation by allowing any company to provide services at any supported network layer [25].

We argue that the inherent complexity of roles and responsibilities in open access networks, combined with the lack of automated network management in these environments, prevents open access networks from being the enabler of innovation they were

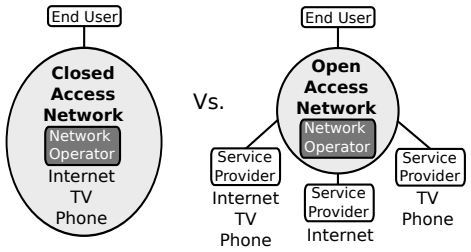

Figure 1.1. Closed vs. Open Networks

envisioned to be. Without automation, open access networks degenerate to more complex versions of their "closed access network" counterparts. In contrast, with automated network management and operations, we envision open access networks to enable the realization of new services and applications that can truly exploit the capabilities of low latency and high capacity access networks.

1.1 Thesis Statement

Our thesis is: an automated network management and operations framework built on a production rule system can capture the dependencies and relationships of both the network infrastructure and the role players in open access network environments.

FlowOps is the framework we designed to test this thesis. It enables network operators to: (i) operate an open access network in an automated, sustainable manner; (ii) reason about various levels of abstraction of the network; and (iii) provide a streamlined, unified, value-added experience to users. FlowOps relies on a layered abstraction model of the open access network environment that allows it to reason and react to events at different levels of abstraction and to propagate relevant information into other layers as needed. At a high level, the underlying hardware should not matter so long as the network can support the abstractions service providers desire to provide services. Therefore, the infrastructure could consist of traditional switches, SDN-enabled switches, or a combination of the two.

Towards this goal, we present in this thesis our work on the *FlowOps* framework. At its core, FlowOps is a network management and operations framework that provides structured, automated network management for heterogeneous open access network environments. However, FlowOps also forms the basis for our vision of a "truly open, dynamic open access network," in which becoming a service provider becomes a simple on-demand event, thus lowering the barrier to entry. In particular, FlowOps models the dependencies and relationships of both the network infrastructure and role players in an open access network, thus allowing for the automation of network management functions within the broader business context of open access networks.

Our approach to realizing the FlowOps framework is informed by the observation that network management and operations tasks involve applying domain-specific logic to realize network management objectives based on information derived from the current (dynamic) state of the network [10]. This definition cleanly maps to a classic description of

production rules systems in which *production rules* are used to reason about and represent knowledge of the domain, and to apply those rules to a *working memory* of assertions (or facts) representing the volatile state of the system [9]. As such, in FlowOps, we use a production rule system as the basis for our approach.

1.2 Contributions

In this work, we make the following contributions: (i) we designed FlowOps as a network management and operations framework targeted at open access networks; (ii) we explored the use of a production rule system as the base technology for the FlowOps framework; and (iii) we developed a prototype of the FlowOps framework and demonstrate its utility in an example open access network environment with a network operator, service provider, and end users along with various scenarios to demonstrate the utility of our system.

We have written and submitted a currently unpublished manuscript [32] to a conference from which this thesis borrows content.

1.3 Organization

Chapter 2 gives background and explores work that has been done in the network management and operations field.

Chapter 3 explains the motivation behind the framework, including a high-level overview of how actors interact with FlowOps.

Chapter 4 describes the architecture designed to provide the management and operations support in our framework.

Chapter 5 provides information and insight into the prototype we developed based on the FlowOps architecture.

Chapter 6 presents both a qualitative and quantitative evaluation of our prototype to see if FlowOps makes network management easier and to explore performance of the current code base.

Finally, Chapter 7 summarizes our contributions and provides some insights into future work.

CHAPTER 2

BACKGROUND

2.1 Production Rule Systems

Production rule systems (or rule-based systems) provide a practical way to capture domain knowledge in a set of production rules and to have those rules operate on the state of the system in question, represented as assertions (or "facts") [9]. The "working memory" in which the facts are stored acts like a database, except that it is more volatile as it can change during the operation of the system.

Our key observation is that this abstraction neatly maps onto the generic network management and operations problem: (Human) network operators apply domain knowledge to perform network management and operations tasks within a network, using information about both the desired and the observed state of the network. This state can dynamically change as conditions in the network change or as a result of actions performed by the operators. Based on this observation, we decided to use a production rule system as the base technology for the FlowOps automated network management framework. We provide a brief overview of the open source production rule system we used in our implementation in the next section.

2.2 Drools Rules Engine

Drools [3] is a Java-based production rule system that manages a working memory of facts and rules that fire based on existing (or nonexisting) facts and their properties. The Java host initializes Drools by loading rules and creating either a stateful or stateless knowledge session. Stateful knowledge sessions allow facts to be inserted after which rules can be run once to receive results. State is not persistent so the host can always add facts and run the rules without worrying about what state anything is in. This is useful for doing tasks like simple calculations or activities where a result is needed without any dependence on previous results. Stateful knowledge sessions, on the other hand, keep state across calls to fire rules.

In evaluating how to go about creating a network management framework, the Drools engine appeared to be the foundation upon which FlowOps should be built. The ability to write many different co-existing rules which are triggered based on properties found in facts provides an interesting opportunity to explore what this paradigm offers and if a management framework could benefit. Here is an example rule:

```
1 rule ``Enable a port''
2 when
3   enableCmd : EnablePort()
4   port : Port(id == enableCmd.portId, !enabled)
5 then
6   port.enable();
7   update(port);
8   retract(enableCmd);
9 end
```

The "when" section (lines 3–4), also called the left-hand side (LHS), is analogous to the WHERE statement in an SQL query. When the "when" section becomes true, the "then" section (lines 6–8), also called the right-hand side (RHS), will be run. In our example, the port is enabled, the Drools engine is notified that the port object has changed, and the EnablePort command is removed.

Rules may be much more complex, with large "when" sections that test over groups of facts with certain properties and nested conditions. In the case that multiple rules are triggered at the same time, Drools consults a "salience" value assigned to each rule to determine the order in which they run, and the outcome of the earlier rules may change facts in such a way that the lower-salience rules no longer need to execute. Rules very often "chain"—when a rule runs, its side effects can include addition, deletion, or updates to other facts, which can, in turn, cause other rules to fire. For example, consider the following rule:

```
1 rule ``Check carrier on port''
2 when
3   port : Port(enabled)
4   not( CarrierCheck(portId == port.id) )
5 then
6   insert(port.carriercheck());
7 end
```

This rule provides a way to check for carrier on a port once it has been enabled: the rule

fires only if information about carrier on the port is not already available. The carriercheck() function could, in turn, cause other rules to be fired if carrier is not detected, such as running additional diagnostics on the port or altering the appropriate parties. Other rules could also invalidate the CarrierCheck() object, which would cause the carriercheck() function to be run again.

A naive approach to determining what rules need to be fired would be to loop through all of the rules and check each rule against all of the facts. RETE [20] has served as the best algorithm for determining what rules should should run in a time-efficient manner and is what Drools employs to trigger rules. The main idea behind RETE is that a network of nodes allows for tests against objects to only be done once and only when the object is added, removed, or updated. The network consists of three main types of nodes. The top-most nodes are object types. If an object of a certain type is changed, a flow will pass through the node. Under the type nodes are conditional nodes which allow flow through them based on if the condition passes. For the "Enable a port" rule above, a type node of EnablePort would need to be triggered. Next a conditional node testing the "enabled" property of Port exists. If other rules test for the same property, they would share the flow instead of adding more nodes. At the bottom exist the rule nodes themselves. All of the condition nodes from the LHS feed into the rule node which is triggered if all condition nodes are activated. All of the rule nodes which are activated are added into an agenda where they can be sorted and run in order. Speed is increased greatly at the expense of memory which becomes a problem in extremely large cases.

Information can flow into and out of Drools in several ways. Rules can insert, update, and remove facts, as seen in the examples above. The host running Drools can also insert, update, and remove facts.

There are more specialized methods for the host to insert facts into special "entry-points" which are logically separate from the main store, allowing only rules explicitly listening to those entry-points to be fired. This allows FlowOps to behave differently when, for example, a port goes offline because an administrator explicitly disabled it (an intentional action) as opposed to being reported by the switch itself (indicating a possible failure in the network). Here is an example of the LHS of two rules which are triggered based on what entry-point is used to insert the fact which triggers the rule:

```
rule ``Port was reported down by the switch''
when
  portDown : PortDown(port_id : id) from entry-point
      ``switch-status''
  port : Port(id == port_id)
then
  ...
end

rule ``Port was reported down by an operator''
when
  portDown : PortDown(port_id : id) from entry-point
      ``manual-status''
  port : Port(id == port_id)
then
  ...
end
```

Either scenario may cause a different chain of rules to fire to handle the same fact change differently. Similar to entry-points being used to insert facts, channels are used to send information from the RHS of a rule to the host. The host program can specify channel handlers which listen for objects being inserted into the channel. Sending an object through a channel is done using the following semantics:

```
...
then
  channel["email-handler"].send(new EmailMessage(email, "Port " +
     port_id + " is down!"));
...
```

Here the host program would need to have a channel listener set for the “email-handler” channel.

2.3 Related Work

Our work is inspired by the conceptual model presented in KnowOps [10]. KnowOps presents a framework which embeds a knowledge base to support network management and operations. KnowOps unifies PACMAN [13], which enables network management workflow tasks, and COOLAID [11], which uses a declarative language to capture knowledge from domain experts and documents. DECOR [12] presents a database-oriented automated network management system. FlowOps extends these ideas with a layered architecture

suitable for abstracting services for network operators and users along with a framework geared towards open access networks.

PRESTO [17] presents a model to update network device configurations by transforming templates from a higher-level language into device-specific configurations through configlets. These templates do not benefit from having a shared knowledge-base and rule engine which exists in FlowOps to reason from desired characteristics to configurations in the network for broader problems.

In FlowOps, we make use of a production rule system as the foundation for our framework. Rule-based approaches have been applied to specific problems in network and systems management. For example, the Eucalyptus cloud platform [29] mentions the use of a production rule system as part of their cloud control architecture, although details are not provided. A production rule system has been proposed as part of an intrusion detection approach in networked systems [26]. A policy description language has been proposed [33] to perform configuration changes of a specific network element (a software voice switch), based on a set of rules defined in their language. A rule-based approach has been applied to realize high-level process automation in a network operations context [23]. To the best of our knowledge, however, FlowOps is unique in applying a rule-based approach to both "read and write" network management tasks (i.e., network configuration and fault management) in a single framework.

Conceptually related to our work, a Knowledge Plane [14] has been proposed as an approach in which AI and cognitive system methods are used to build high-level models to provide services in other parts of the network. The principles described in this work are related to FlowOps. The 4D [21] architecture uses a decision plane which maintains a network-wide view and controls network elements and is similarly conceptually related to FlowOps. In our work, however, we take a more pragmatic approach and focus on open access networks as a particular problem domain of interest.

Network federation [22] has been suggested which focuses on the interfaces between network operators instead of the framework used to manage each network. FlowOps focuses on supporting allocations at multiple network operators with federation done manually through tools; however, it is feasible that similar federation methods like those described could be employed to give FlowOps a more robust solution for services spanning

multiple networks.

Network troubleshooting and analysis is a rich area of ongoing research under the general network management umbrella and a thorough coverage of related work is beyond the scope of this paper. For example, a Generic Root Cause Analysis platform (G-RCA) [34] was designed for service quality management in large IP networks by supporting customized rules which can be used to analyze network events. Another example is the NICE (Network-wide Information Correlation and Exploration) system [27], which enables troubleshooting of chronic network conditions by detecting and analyzing statistical correlations. We note, however, that these works are complementary to FlowOps which will simplify the creation of such network management tools that can benefit from the FlowOps network-wide visibility and abstractions.

Finally, the open access network abstractions that we explore in FlowOps is related to the ChoiceNet project [31]. This project describes a network architecture which would enable a network operator to expose different layers where users can configure services depending on their needs. The concepts described mirror our choice in FlowOps of exposing separate service layers so that users are free to innovate at different layers of the architecture. The framework described focuses heavily on the mechanisms used by users to provision services while the details of how that happens within the infrastructure and how management tasks like fault management work are touched on but not described in detail.

CHAPTER 3

MOTIVATION

As we described earlier, open access networks are often built on fiber-to-the-home technology, thus providing a network with the inherently attractive features of high capacity and low latency. The services offered by service providers on current open access networks often degenerate to a choice between a small number of "regular" internet service providers (ISPs). Although a high capacity, low latency connection is in and of itself attractive with such service offerings, there appears to be a lack of services and applications that fully exploit these capabilities. Such services could be interactive IPTV, home security and automation, smart grid utilities, medical monitoring, virtual private networks, and emergency services. It is our contention that this lack of innovative services and applications is a direct consequence of the lack of automation in the network management and operations in open access networks. In short, the burden of entry to becoming a service provider is too high. Below we describe our vision for a more open or *dynamic open access network*, in which there is a low barrier to entry in becoming a service provider.

3.1 FlowOps for Dynamic Open Access Networks

We illustrate how FlowOps enables the concept of a dynamic open access network with the aid of a scenario where an end user orders a service that a service provider then sets up. For our scenario, we assume that the network operator has already built out the network infrastructure. This involves: (i) deploying a fiber infrastructure to form the backbone of the network; (ii) deploying network equipment in this backbone to handle transporting packets between users; and (iii) running data lines out to users including homes, data centers, etc. The network operator must keep inventory of network devices and access methods to bootstrap the environment. Once the infrastructure is built out and FlowOps has been configured with the appropriate access methods, users would be allowed to start allocating resources for services over the open access network. The basic workflow for

an end user to request services within such a dynamic open access network is depicted in Figure 3.1 and described below.

In step 1, end users browse services available on their network that are offered by service providers. In a dynamic open access network, we envision there to be many service providers, offering services at various granularities and time scales. Some service will undoubtedly resemble current ISP style "static" and "heavyweight" services, with customer/provider relationships existing over an extended period of time. However, we also expect services (or networked applications) being offered at relatively small granularities and over shorter time scales. For example, the end user to end user "LAN party" depicted in Figure 3.2 might be created on the fly for the duration of a gaming event. We expect the FlowOps architecture to facilitate the creation of innovative services that can exploit the inherent capabilities of a low latency, high capacity dynamic open access network.

Once the end user has selected a service, the order is sent to the service provider in step 2 in Figure 3.1 to be fulfilled. The service provider must then determine what resources are needed to realize the requested service. End points included in the order are determined

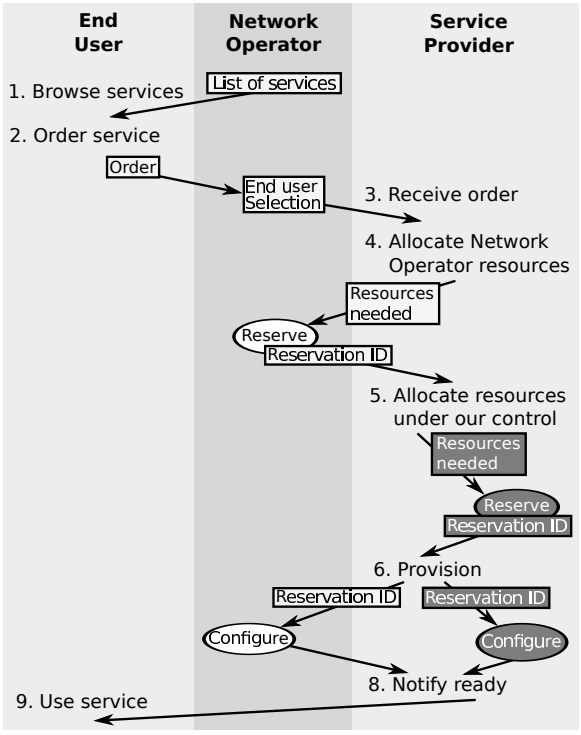

Figure 3.1. FlowOps workflow for Dynamic Open Access Networks

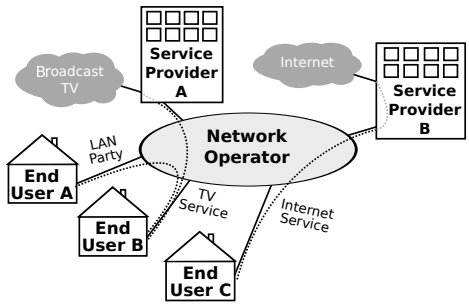

Figure 3.2. Actors

by the end user and the service provider. End users are free to choose what endpoints a service uses or allow any appropriate free endpoints be used. Service providers are also free to configure services through generic or specific end points where they are connected into the network. For example, they could provide different services from separate ports or load balance services between ports. The step to reserve end points includes mapping generic points to available points since the reservation requires knowledge of exactly where services need to be configured.

Simple services only require configuration in the network operator's network. More complex services may require configuration of resources under the control of the service provider as well. Figure 3.1 depicts a service that requires resources to be configured in both the network operator and service provider networks. Resources must first be allocated in all networks which means they are promised but not yet given to the requestor so that the service provider knows the service is fully realizable end-to-end. Step 4 starts the allocation process for resources in the network operator's network. If and only if resources are successfully allocated there, the resources will also be allocated in the service providers network in step 5, which results in a full allocation. If all allocations work, then the service provider can provision the resources in step 6 to realize the service. Once the service has been successfully provisioned, the end user is notified that the service is usable and the service provider can monitor the service using automated fault management and other available service support capabilities provided by the network operator.

A key point of the presented scenario is that interactions between end users, service providers and the network operator are completely automated and programatic. In this manner, adding and removing service providers, and adding and removing services pro-

vided by such providers become automated actions resulting in a dynamic open access network. In the remainder of this paper, we describe details of the FlowOps architecture and implementation as the basis for such a dynamic open access network environment. Having a unified method to specify users and edge points and handle allocation and fault management simplifies matters for all users but especially service providers who may deal with multiple network operators to cover large areas. Automated tools could handle most of the steps in the workflow. Having such an ability is a great boon to all users involved.

3.2 Open Access Networks

Open Access Networks (OANs) are most commonly realized as high capacity, low latency Fiber-to-the-Home (FTTH) networks. There are many examples of publicly owned OANs [25], including Utah's Utopia [7], Stockholm's Stokab [19], and Amsterdam's CityNet [5]. Some private organizations have also built out OANs, including Reggefiber [15], Quadracom [1], and MBC's network [2].

Figure 3.2 depicts the role players in an open access network. The *network operator* owns and operates the OAN and facilitates slicing (or sharing) of the network between different users. *Users* are all other actors who wish to provide or receive services. Users can be categorized as *end users* who primarily receive services provided by *service providers*. End users and service providers are both users having equal ability to provide or receive services from other users; however, we will use these terms to denote a user's primary purpose. For example, in Figure 3.2, Service Provider A provides broadcast TV service to End User B and Service Provider B facilitates Internet connectivity to End User C. End User A and End User B have set up a "LAN party" without the intervention of a service provider, demonstrating that end users and service providers have the same abilities to configure services with other users. We note that open access networks in general do not provide the latter type of service abstraction. This is a simple example of the "richer" service abstractions that we wish to enable with the FlowOps approach.

Roles in an OAN are relative to each other since any single actor may have multiple roles. A service provider who offers services in a network operator's network may need to configure their own network and could be running FlowOps as their own network operator.

3.2.1 Network Operator

A network operator is primarily interested in running their network in such a way to minimize time and effort needed to manage the network while providing the most value to users. A layered model allows the network operator to reason at various levels on how management works in various scenarios. A standard API gives users a common interface to allocate and provision services to connect with other users. The vertical integration from user down to device configuration provide value-added management opportunities for better fault handling and auditing. Network operators and users use the same management software with the only difference being what they are allowed to see and do.

3.2.2 Users

Users are all actors other than the network operator. They can configure services through points where they are connected through other points on the network where other users are connected. The network manager's primary job is to enable users to be able to provide and consume services from each other. Most users would be categorized as either a consumer (end user) or a producer (service provider) even though consumers could also act as producers and vice-versa. OANs are a great opportunity to explore what happens when every user can receive or provide services just like any other user.

3.2.2.1 End Users

End users, in general, are only interested in ordering services and having them work. In an OAN, they would most likely order services through a portal where they can discover what is available. Allowing users to find all services easily should help drive innovation and quality where service providers must compete with each other through features, quality, and support instead of relying on being able to spend more money than each other for visibility or access.

Equipment needed on end user premises depends on the network design of the network operator and potential secondary equipment needed by service providers. A common practice for network operators is to install a Network Interface Device (NID) at each site, which acts as a demarcation point between the responsibility of the network operator and end user. In a FTTH environment, fiber enters the NID where the signal is converted and different wiring exits into the premise.

3.2.2.2 Service Providers

Service providers have two main tasks: develop a sustainable service which other users want and support getting that service out to as many users as possible. Modern service providers are either low-level providers of Internet connectivity or high-level providers of over-the-top services available to any user connected to the Internet.

In an OAN, service providers can straddle the line between being a low-level or high-level provider due to having control over the network being used.

Service providers can provide and optimize services any layer available for allocating. They are responsible for describing the network needed between them and their customers. A service provider may have connections at a data center into the OAN where all of their services are configured to pass through. When they provision a service in the OAN, they may also need to configure their side for everything to work. Because the tasks in both networks are very similar, it may be advantageous for service providers to run an instance of FlowOps for their own network as well. If both actors use FlowOps, tools could be used to stitch services requiring configuration in both networks. When service providers deal with multiple network operators or collaborate with other service providers, common management techniques like those available in FlowOps provides great value.

Service providers might be interested in getting priority connections to ensure behavior like high bandwidth, high priority, or low latency or lower priority connections for lower prices. Priority would enable services to be able to kick off or throttle back other lower priority services. An example would be an emergency 911 call getting priority over IPTV.

CHAPTER 4

FLOWOPS ARCHITECTURE

An overview of the FlowOps architecture is depicted in Figure 4.1. All actors interact with FlowOps through a common API that provides all of the functionality needed to configure and manage services. The Knowledge Store is where all state is kept and updated as services get configured, faults occur, etc. Abstraction layers keep the state manageable, allowing complex high-level definitions to be mapped to low-level details. Inside the Knowledge Store, the Rules Engine enables model dynamics by detecting changes and reacting to them. Underneath the Knowledge Store is a Driver Engine that translates messages to and from devices using their control interfaces. Through the various layers in this model, complex high-level tasks such as setting up a new service can be translated into raw configurations and low-level errors can propagate up to be detected and dealt with.

4.1 FlowOps Components

4.1.1 Knowledge Store

Information needed to drive the model needs to be accessible in a structured manner. All entities from our model, like actors, services, infrastructure, etc. are added into the Knowledge Store as facts. The Knowledge Store is the "brains" of FlowOps, containing the facts and rules needed to make decisions. Our model lives and reacts within the Knowledge Store in the layers we have defined.

4.1.2 Abstraction Layers

Figure 4.2 shows the layered model employed in FlowOps. Abstraction layers provide simplified management of the entire system at different levels, including a business layer for actors and other high-level entities interacting with the system, a service layer exposing reservable resources, a network operator layer that maps the services into the backbone network defined by the network operator, and an infrastructure layer representing the physical devices deployed by the network operator. A layered model provides the following

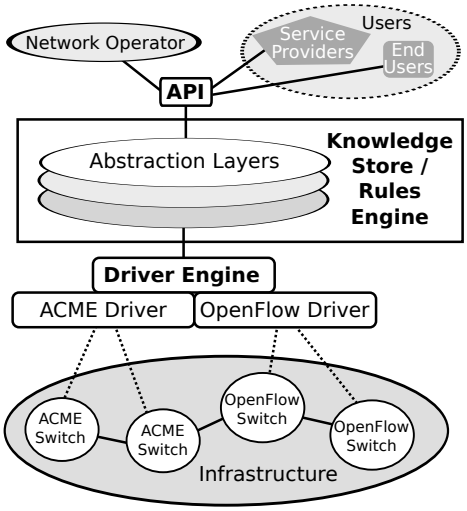

Figure 4.1. FlowOps Overview

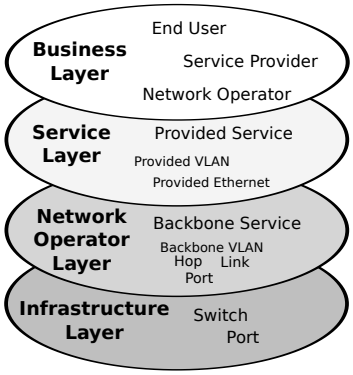

Figure 4.2. Abstraction Layers

benefits: (i) separation between desired resources vs. how those resources actually get implemented; (ii) supporting allocating, provisioning, and monitoring received resources; and (iii) offering relevant views and alerts to each actor.

- *Business Layer:* Relationships between the different role players are captured in the top layer of our model. Alerts for actors are kept here as well.
- *Service Layer:* Provided services are described in the service layer. A desired service could be a VLAN from point A to point B or a LAN with multiple endpoints. This layer simply defines what services can be provided in the network and leaves the implementation details to the network operator layer.
- *Network Operator Layer:* Provided services like the VLAN or LAN examples above

need to be mapped onto the backbone infrastructure. This mapping is dependent on how the network operator runs the backbone network. One network operator might provide VLAN and LAN service abstractions through configuring a actual VLAN through the backbone network while another may create a VPN over an IP network. Users need not be concerned with how the network operator provides the service so long as their desired networks act correctly. The network operator layer should be chosen based on available infrastructure and technologies needed to realize the services available in the service layer.

- *Infrastructure Layer:* The infrastructure layer consists of the hardware components under the network operator's control, like switches, wires, gateways, routers, etc.

4.1.3 Driver Engine

If the Knowledge Store is the brain for the model, the Driver Engine is the "brawn." After configurations for infrastructure devices are generated, the Driver Engine takes responsibility to apply network configuration changes to network devices. Drivers configure devices through appropriate control interfaces like CLI, NETCONF [18], OpenFlow [28], etc. In addition, the Driver Engine also updates the facts in the Knowledge Store when states and settings change or errors are generated in the network. It is crucial that the Driver Engine keep the Knowledge Base consistent so that rules are not applied to facts which no longer represent the state of the network. We note that abstracting interaction with the network devices in the Driver Engine implies that it is the only part of our architecture that needs to "know" about the actual underlying hardware in the network. Specifically, the different layers of the model represented in the Knowledge Store can deal with network equipment in an abstract manner, leaving it up to the Driver Engine to realize that abstraction on the actual hardware.

Up until now, the abstractions have been generic enough that we havent cared how the Driver Engine actually propagates infrastructure settings to the hardware. Is the Driver Engine a SDN controller? Is it just software that manages traditional switches? The answer is the Driver Engine is whatever the network operator needs it to be. The model is abstract because at a high level, it really does not matter which is used. Hardware choice may have implications on what is possible to provide when configuring the infrastructure, but at a high level, you should only have to specify what is needed in the infrastructure without the

need to know how it gets done.

A SDN-based driver would allow the abstraction model to be very closely tied to the flows which are allowed through switches. For example, instead of writing out state to switches proactively, the driver could just react when packets are sent through the network. When a packet is seen without any flows installed for the switch, the driver could query the Knowledge Store to determine what to do. Of course, the driver could also proactively install such commands on switches as services are provisioned so it is just a choice of implementation. In either case, the driver would need to actively edit or remove flows from switches as services are deleted or modified.

A driver which hosts traditional switches would need to proactively program the switches in the network according to the abstraction state. Most current network operators would need to use this style of driver since SDN-based networks are not yet widely deployed.

4.1.4 Rules Engine

Every entity in our model can be thought of as a fact that is stored and accessible to the rules engine. Rules enforce constraints defined in the system and can be defined to be fired when new facts are entered into the system (e.g., when there is a new user), when facts are removed (e.g., when a user leaves), or when facts change (e.g., when the status of a port changes). Rules act as glue between the layers in our model. Many events and facts need to be communicated between layers.

4.2 FlowOps Operation

4.2.1 Allocation, Provisioning, and Deletion

The basic user actions needed in the model are allocate, provision, and delete. Allocation is the process in which an actor reserves a resource, e.g., as described in the example scenario in Section 3.1 and depicted in Figure 3.1. If an allocation is successful, the network operator has promised that it will, for the valid length of the allocation promise, honor the request if provisioned. After allocation, no configuration has actually made it to the infrastructure. Provisioning an allocated resource realizes the reservation and instantiates it. Deleting an allocated or provisioned resource frees the resource. In addition to these basic actions, update is available to change resources on-the-fly without the need to delete and then re-allocate resources.

Figure 4.3 displays how an allocation travels through each abstraction layer in our model. When an allocation is made by an actor in the business layer, a provided service fact is added to the Knowledge Store. Rules detect when a desired service is requested for allocation and maps it to a supported backbone service in the network operator layer or fails if no mapping is possible. If the network operator uses VLANs in the backbone to offer services, then rules map desired services to a backbone VLAN. Rules in the network operator layer configure a path through the network and ensure no existing configuration conflicts with the service. Once the configuration is prepared for each network device along the path, the Driver Engine translates the configuration for each specific device when the service is provisioned.

4.2.2 Views and Alerts

An important use of FlowOps is enabling actors to diagnose problems and audit their services to ensure they are getting what they require. While the network operator has complete access, users are given limited views based on their services. These views can include measurement results, path information, etc. It is left up to the network operator to decide what should be included in a view. Along with views, users are able to receive alerts about important changes in their views like connectivity issues and planned maintenance.

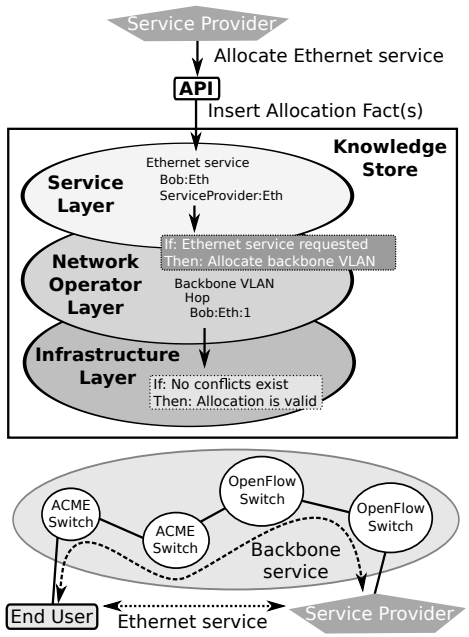

Figure 4.3. Allocation

Having clear views into the network is important so that problems can be tracked by all affected entities.

Figure 4.4 displays how an alert propagates from a switch to interested actors through the model. Errors reported by a device are translated and added into the Knowledge Store by the Driver Engine, which knows how to deal with each device. The error is mapped to the specific port, switch, etc. that is able to connect the error to dependent backbone services. Provided services that depend on the affected backbone services receive alerts about the error. Actors can be contacted based on the severity of the problem, or the system could try to fix the problem automatically by rerouting services. Similar to fault management, planned maintenance can be dealt with in a similar manner. If a switch needs to be taken down for a brief period, for example, an alert could be sent notifying all services depending on that switch far before the change occurs giving time to reroute services.

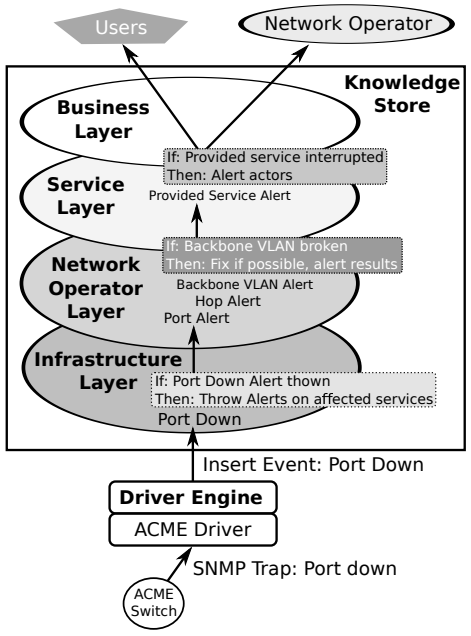

Figure 4.4. Alerts

CHAPTER 5

IMPLEMENTATION

We implemented the Knowledge Store, Rules Engine, Driver Engine, and API of the FlowOps architecture. We used the open source production rule system Drools [3] as the basis for our implementation. Everything is written in Java except for the rules, which are a hybrid of the Drools rules language and Java. FlowOps initializes the Knowledge Store using configuration files with basic properties of the expected network, i.e., the switches and other equipment FlowOps is expected to communicate with. The Knowledge Store is initialized as Drools stateful knowledge session. Rules are loaded separately from the Knowledge Store and are currently compiled into FlowOps, although they could be loaded externally to support a more pluggable model in which network operators could create, edit or load custom rules. An internal HTTP server is started which supports the user commands available in the API.

5.1 Driver Engine

The Driver Engine is initialized by assigning hardware types to their respective drivers. We implemented a driver that supports configuring OpenFlow-enabled switches. We opted to use Floodlight [4], an open source OpenFlow controller written in Java, which allows installing static flows on OpenFlow-enabled switches through a RESTful API. Our driver translates Driver Engine commands to install or remove configurations into RESTful calls to Floodlight. Floodlight is started separately from FlowOps and the only interaction with FlowOps is through the static flow pusher API. Because we use the static flow pusher, our driver creates network configurations that do not utilize any learning based on network addresses. This results in a hub-like network where packets are sent to all possible destination paths.

5.2 Supported Services

We chose a number of basic network abstractions as example services to demonstrate the utility of FlowOps. Our implementation uses VLANs, the most common slicable method available in switches, in the backbone for services. Supported provided services include LANs and VLANs. Provided LANs are mapped to backbone VLANs where the edge points remove the VLAN before exiting the OAN. Provided VLANs are mapped directly to backbone VLANs and allow the user to define whether a tag should exit at any end point or not. Supported topologies for both LANs and VLANs include E-LINE (point-to-point), E-TREE (root-to-leaf), and E-LAN (many-to-many). These topologies are defined by the Metro Ethernet Forum [8] and are used to allow network operators to provide a common environment.

5.3 API

A common API is provided as a RESTful service integrated in FlowOps so that all actors can develop useful, shareable tools which interact with the resources. Actors who run FlowOps, like the network operator, allow themselves and other actors to perform management and operations functions. Through the API, each actor may have a separate view based on what they should or should not be able to access. Alerts that bubble up, like a port-down event, eventually make it to alerts for provided services which relevant actors are able to see and react to.

The API as exposed in our implementation allows for unauthorized calls to view general information in the Knowledge Store. Most calls require a token to be set in the HTTP header that designates the user making the call. This provides access to user views and accountability when dealing with resources. Available API calls to work with resources include *allocate* which requires a resource specification document (RSPEC) [6], *provision* and *delete* with the reservation ID from the allocation step, and *update* with an updated RSPEC. State can be queried using the API functions for *all* to view all facts, *provided* to see provided services, *backbone* for services in the network operator layer, *slivers* for all configured resources, *hostedon* with an ID of an infrastructure device to view services configured through it, and *connectables* to view connectivity points, actors, and alerts.

We wrote a python library which makes it easy to specify variables like the caller ID, RSPEC, reservation id, etc.

5.4 Rules Techniques

We utilized many features available in the Drools rules engine to implement our model more efficiently. For example, initially we found ourselves writing many rules that were more or less duplicates of each other with very minor differences. Salience values allowed us to consolidate our rules. For example, one rule is written which is the default behavior at any layer of the model, while rules for specific conditions that require modified behavior from this default could be created with a higher salience value.

For example, an allocation is valid if all dependent lower allocations are successful. That is a very simple statement which can be applied at any level of our model. There might be certain allocations which need more policies applied to determine if the allocation worked or not. In that case, a higher salience value could be set on a rule with more specific triggers like allocation on a certain hardware type.

We needed to get information in and out of the model. Initially, simply inserting or updating facts and calling a function to run all rules and then checking the facts we were interested in was enough. However, as our model developed, this simple interaction quickly became cumbersome. We made use of channels and entry points as a means to enter information into and extract information from the Knowledge Store. FlowOps uses channels to handle asynchronous workflows where something like a provision command can be inserted into the knowledge store and the framework can wait until a message is sent back saying whether the provision worked or not.

CHAPTER 6

EVALUATION

6.1 Environment

We designed an example environment consisting of a network operator, service provider, and several end users, as shown in Figure 6.1, where we could evaluate FlowOps. The network operator has e.g., switches enabling connectivity through e.g., points where users connect into the network along with backbone switches connecting these e.g., points together. For simplicity, any reference to the network operator's network will simply be referred to as the Open Access Network (OAN). e.g., switches could be a large commercial switch in a data center where service providers connect or Network Interface Devices (NIDs) on the houses of end users. The service provider has its own network with a switch connected to one port on one of the network operator's e.g., switches. Attached to this switch are hosts that offer services if networks are provisioned from them into end points in the OAN. Three end users (Alice, Bob, and Carl) are connected each to a NID found on the e.g., of the OAN. NIDs have a fixed connection into the OAN and many open end points where the end user can connect to and configure services through. In our simple model, these open end points where the user can connect to are Ethernet ports. Both the network operator and service provider have a configuration file which specifies all of the information needed to initialize their own instances of FlowOps, including information like lists of equipment and actors.

6.2 Emulation

We performed an initial evaluation by running FlowOps on a host machine while connected to an emulated network environment using Mininet [24] inside a virtual machine. Mininet supports creating virtual instances of OpenFlow-enabled switches and connecting them together to emulate a real network environment. In our emulated environment, each switch is a separate instance of Open vSwitch [30], a software switch that supports the

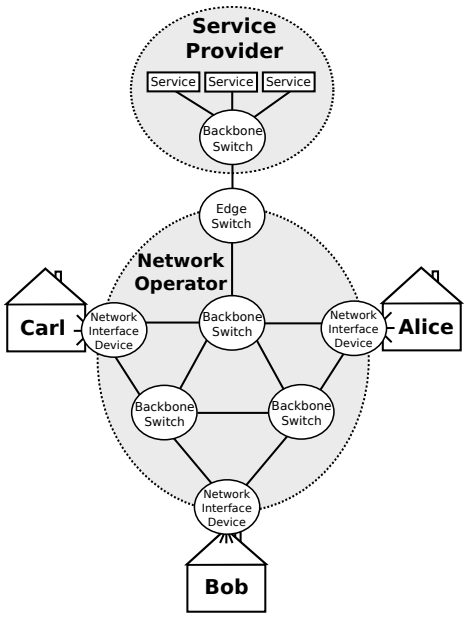

Figure 6.1. Environment

OpenFlow protocol. These switches are controlled via a Floodlight OpenFlow controller, which in turn is driven by FlowOps Driver Engine. (We leave extensions to the Driver Engine to support legacy switches as future work.)

The switches are told a certain IP address where they should expect an OpenFlow controller which we run on the host machine. We give the script the configuration files for the network operator and service provider since we need to simulate the entire network. Next, we run Floodlight, an OpenFlow controller, on the host machine after which the switches connect and listen for instructions. Lastly, two copies of FlowOps are executed on the host: one with the configuration file for the service provider and one for the network operator. Any individual actor who runs FlowOps will only have control over equipment owned by them, which means a new instance of FlowOps is required for each actor hosting their own infrastructure. Once the FlowOps instances has started after the network has been instantiated, we are ready to run experiments.

6.3 Configuration

Our first experiments demonstrate that FlowOps supports creating usable networks using the architecture described and implemented for the environment we prepared. These examples demonstrate that FlowOps is able to operate an OAN in an automated manner

based on external users asking for slices of the network resulting in reasoning in the various abstraction layers of our model. Services successfully configured include: (i) Ethernet E-LAN between Alice, Bob, and Carl. None of the users need special equipment that understands VLANs. A LAN party is one example of what the users could do in this scenario. (ii) VLAN E-LINE between the service provider and Bob. Because the service provider only has access on one port into the OAN, it must use VLANs to differentiate services as it sends packets. Bob's end point is configured to remove the VLAN so that his equipment has no need to understand VLANs. The service provider could use this to provide Internet, VoIP, etc. to individual users. (iii) VLAN E-TREE between the service provider as the root to all the users as leaves. Similar to the VLAN E-LINE, a VLAN is needed for the service provider to specify which packets belong to what service after which the tag can be removed before it reaches the end user equipment.

6.4 Fault Management

We created a bottom-to-top test demonstrating that higher-level entities are able to deal with low-level problems like ports going down. For simplicity, we created a simulated driver which does not perform any configuration commands (acting as a no-op) and added a simple API we could call which would simulate port up/down events which would get added into the Knowledge Store. These examples demonstrate some of the value-added features available through FlowOps by putting hooks into our model where fault management can be automated by attempting to fix problems and notifying respective parties of affected services. The following scenarios were tested: (i) Port down in a path which can be rerouted. In this case, the alert propagates from the port up to the backbone VLAN. At that point, a new route is generated and installed and the alert propagates up, informing provided services depending on the backbone VLAN that a problem was detected and a new route was installed. (ii) Port down in a path where no other possible path exists. Since no other route exists, the port down event propagates all the way up where the user and/or the network operator can decide what to do next.

6.5 Performance

6.5.1 Methodology

To gauge behavior, we ran various experiments to observe the system over time. We were interested in evaluating system characteristics related to our main actions: (i) allocate; (ii) provision; and (iii) delete. Each has a different set of rules which fires using different facts and patterns. Since the behavior of each call might be related to what calls were previously made, we used various call sequences including:

- Calling allocate then delete multiple times (e.g., ADADAD).
- Calling allocate then provision multiple times (e.g., APAPAP).
- Calling allocate, provision, then delete multiple times (e.g., APDAPD).
- Calling allocate multiple times, then provision multiple times, then delete multiple times (e.g., AAPPDD).
- Calling allocate multiple times, then delete multiple times, then repeating (e.g., AADDAADD).

See Table 6.1 for the list of call orders we were interested in within the call sequences. For simplicity, figures will refer to the shorthand used for the orders where A, P, and D refer to allocate, provision, and delete, respectively, and a capital case letter is used for the specific calls being evaluated. Since the underlying infrastructure could have a large impact in behavior, we tested using three very different topologies: (i) One short with 100 users and 4 service providers connected to 1 switch with a total of 104 network devices; (ii) One massive with a backbone ring of switches connected to e.g., switches which each connect to multiple users or service providers for a total of 122 network devices; and (iii) One long with many backbone switches in a ring each connected to only one user with 204 total network devices. Evaluating calls on the LONG topology were very long so fewer data points were taken for it. We observed the following information for each call made: (i) how long it took to complete; (ii) how many total facts were in the Knowle.g., Store; and (iii) how many rules fired. In addition to recording data related to each individual call, we profiled the entire application to give us a good idea where most of the time was taken. A combination of all of this information gave us insight into how and why FlowOps performs the way it does.

Table 6.1. Call sequence orders.

Call	Order	Description
Allocate	AA	back-to-back
	aaddAA	after a complete cycle of allocating and deleting
	adAdA	after deleting
	apApA	after provisioning
	apdApdA	after provisioning and deleting
Provision	aaPP	after a complete cycle of allocating
	aPaP	after allocating
	adPadP	after deleting and allocating
Delete	aDaD	after allocating
	apDapD	after allocating and provisioning
	aaDD	after a complete cycle of allocating
	aappDD	after a complete cycle of allocating and provisioning
	aaddaaDD	after a complete cycle of allocating, deleting, and allocating again

6.5.2 Drools Rules Fired and Total Facts

Initially, we speculated that the number of total facts or rules fired would follow closely with how long each API call took. To test our speculation, we recorded how many rules fired during each API call and how many facts were in the system when the call was made. As expected, the topology heavily impacted the number of rules fired and total facts.

Figures below (6.2 for rules fired during and 6.3 for total global facts after) show data collected for each allocate call: (i) back-to-back (e.g., AA); (ii) after deleting (e.g., adAdA); (iii) after provisioning and deleting (e.g., apdApdA); (iv) after provisioning (e.g., apApA); and (v) after a complete cycle of allocating and deleting (e.g., aaddAA).

There are also figures (6.4 for rules fired and 6.5 for total facts) showing data for individual provision calls made: (i) after a complete cycle of allocating (e.g., aaPP); (ii) after allocating (e.g., aPaP); and (iii) after deleting and allocating (e.g., adPadP).

The remaining figures (6.6 for rules fired during and 6.7 for total global facts after) show data collected for each delete call: (i) after allocating (e.g., aDaD); (ii) after allocating and provisioning (e.g., apDapD); (iii) after a complete cycle of allocating (e.g., aaDD); (iv) after a complete cycle of allocating, deleting, and allocating again (e.g., aaddaaDD); and (v) after a complete cycle of allocating and provisioning (e.g., aappDD).

The number of rules fired tend to stay constant. It would be expected that the fired rules would stay constant until paths length change; however the multithreaded property causes

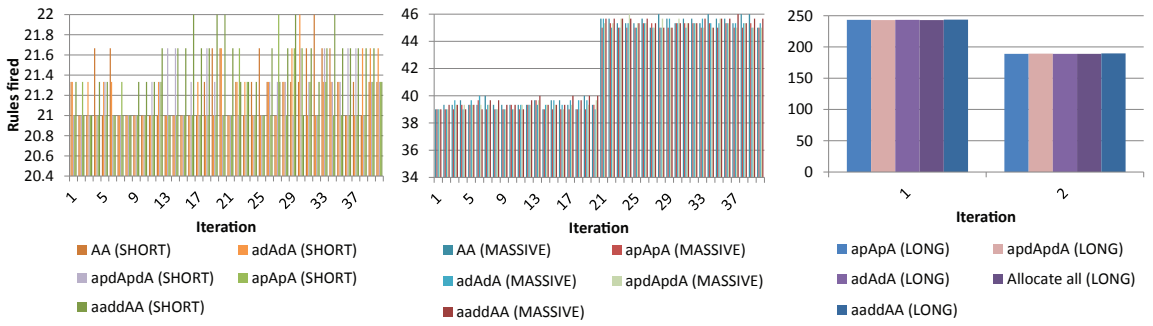

Figure 6.2. Fired rules on allocation.

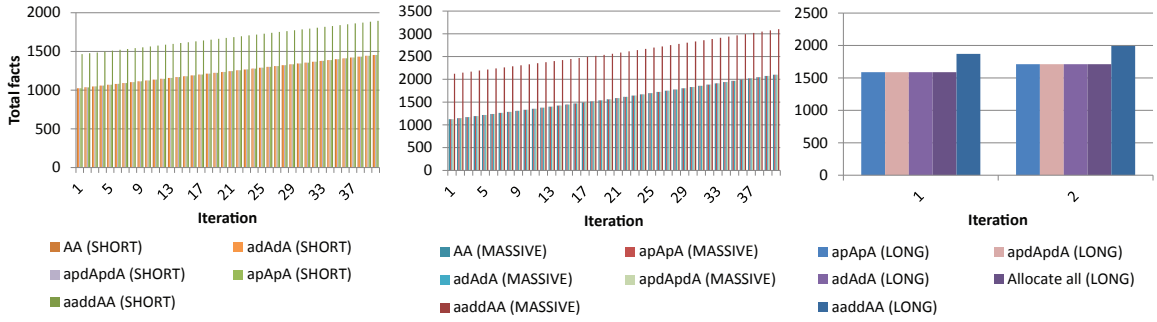

Figure 6.3. Total facts on allocation.

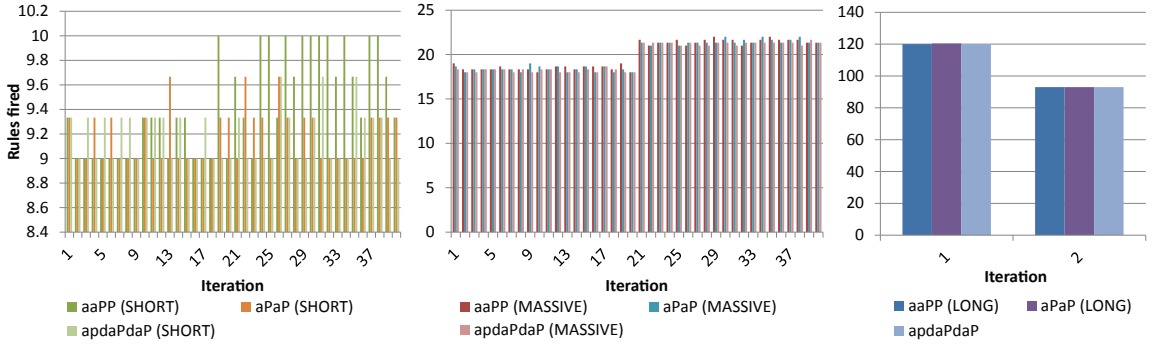

Figure 6.4. Fired rules on provision.

the fluctuations you see where the number of fired rules varies slightly. The large jump in the number of rules fired in the middle and right graph in Figure 6.2 is due to the change in path length. Rules are fired for each hop along the path in an allocation. Again, because there are more hops in the allocated path, more rules are fired in delete (Figure 6.6) and provision (Figure 6.4) as well.

The total number of facts followed trends according to the behavior of the API calls.

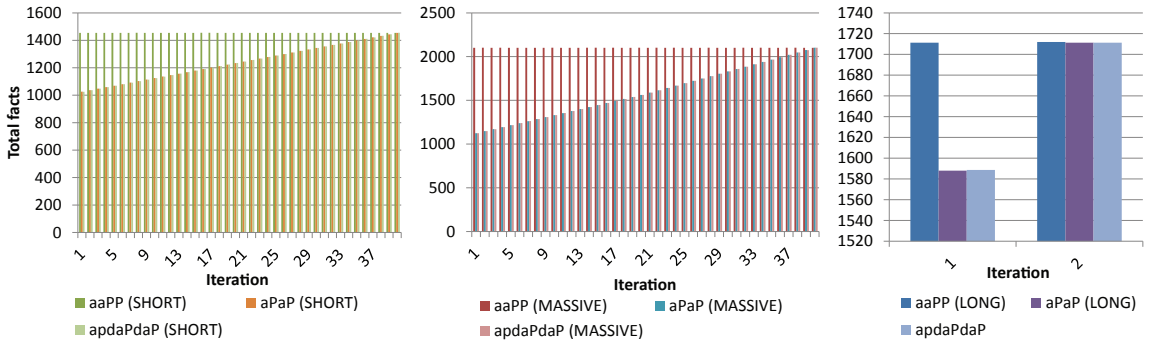

Figure 6.5. Total facts on provision.

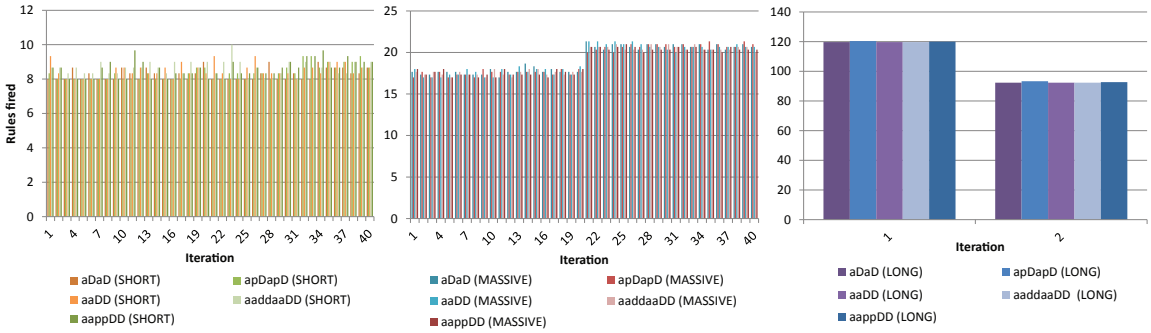

Figure 6.6. Fired rules on deletion.

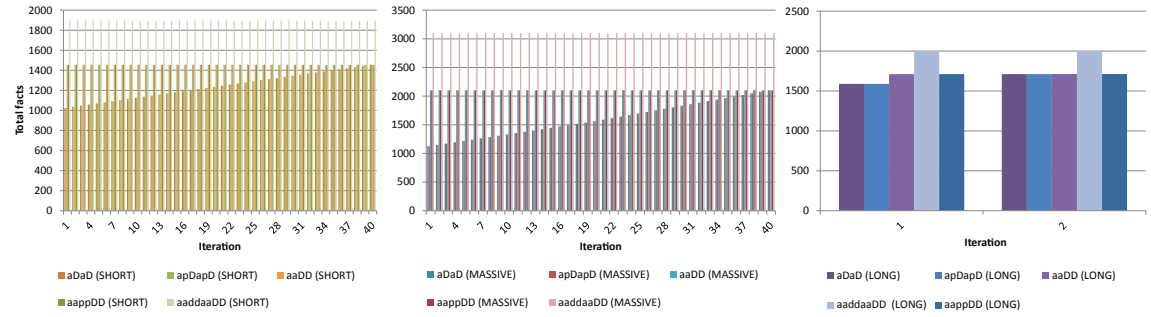

Figure 6.7. Total facts on deletion.

Since allocate calls always insert new facts, the total facts always grow, as shown in Figure 6.3. Delete and provision calls only change the state in existing facts inserted in allocate, which explains the flat lines visible in Figure 6.5 and 6.7. The cases where the total facts are increasing are due to the allocate calls being made between each call.

6.5.3 Timing

The graphs showing the time taken for each API call are split up into three groups, one for each API call (allocate, provision, and delete). Each group is split up into rows and columns. Columns show the same calls and sequences on the different topologies. The range of values differs greatly based on the topology so make sure to note the changes to the Y-axis range in each graph. Rows display the sequences of calls which have the same increasing or decreasing trend.

Figure 6.8 displays the number of seconds each allocate call took, based on its relation to other API calls. The first row shows timing results for allocation calls made: (i) back-to-back (e.g., AA) and (ii) after a complete cycle of allocating and deleting (aaddAA). The second row shows timing results for allocation calls made: (i) after deleting (e.g., adAdA); (ii) after provisioning (e.g., apApA); and (iii) and after provisioning and deleting (e.g., apdApdA).

An interesting trend in these numbers is that the allocation time increases based only

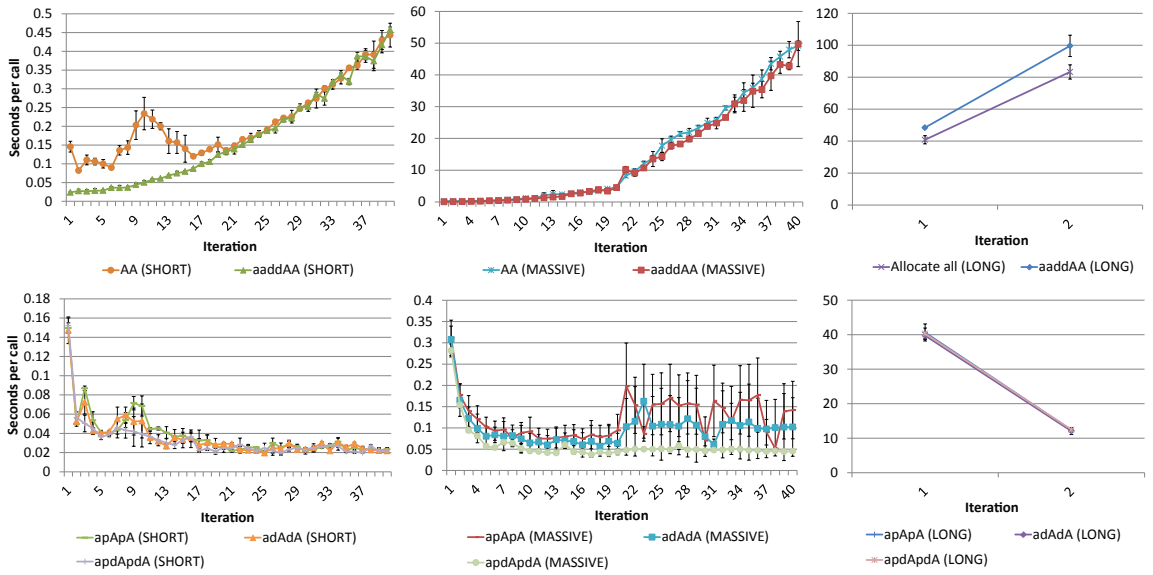

Figure 6.8. Allocation timing benchmarks.

on the number of active allocations. The top row shows consecutive allocations, which means that each allocation increases the total allocated services. The first data set shows 40 allocations while the second shows 40 allocations made after 40 allocations were already made and deleted. Since deleted services only have their state changed to delete, the number of facts grows even as allocations are deleted. This means that even though the two different allocation sequences from the first row have a different number of total facts in the system, there is no obvious increase in time with more total facts pointing to the observation that active allocations are the main variable which influence how long an allocation call takes. Row two, on the other hand, has provision and delete calls between the allocate calls which means the total number of active allocations stays at zero since all allocations are provisioned or deleted which moves them to a different state. Putting all of this information together, we speculate that the major cause of performance degradation seen is due to how Drools handles insertion or updates of facts given a set of rules which potentially have complicated conditions when facts change related to those conditions.

Figure 6.9 shows the number of seconds each provision call took in various sequences of API calls. The first row shows timing results for calls made: (i) after a complete cycle of allocating (e.g., aaPP) and (ii) after allocating (e.g., aPaP). The second row shows timing

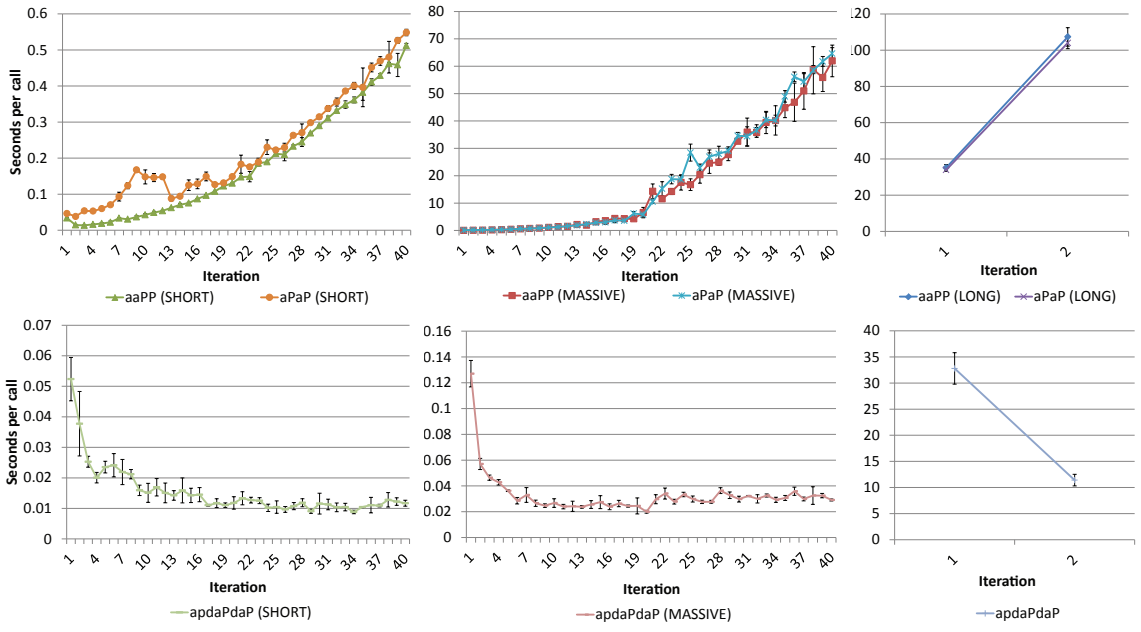

Figure 6.9. Provisioning timing benchmarks.

results for calls made: (i) after deleting and allocating (e.g., adPadP).

Provision calls follow the same timing trends as Allocate calls. The first row shows provision calls back-to-back demonstrating that time increases as more provisioned services exist. The second row includes delete calls between the provisions so the number of provisioned services stays at zero with no increases in time to provision new services.

Figure 6.10 displays the number of seconds each delete call took in a sequence of API calls. Delete calls were timed: (i) after allocating (e.g., aDaD); (ii) after allocating and provisioning (e.g., apDapD); (iii) after a complete cycle of allocating (e.g., aaDD); (iv) after a complete cycle of allocating and provisioning (e.g., aappDD); and (v) after a complete cycle of allocating, deleting, and allocating again (e.g., aaddaaDD).

A major difference in the timings for delete calls is that the time never decreases. The main cause of this is that facts related to deleted services never go away in the current implementation of FlowOps. Given that the number of facts for deleted services only increases, it makes sense that the trend for deletion calls is always increasing. This could be remedied by simply removing facts for deleted services instead of changing the state and keeping the facts around.

6.5.4 Hot Spots

Using JProfiler [16], we determined which functions were taking the most amount of time. Two tests were run during different API calls to get the maximum coverage. In one test which ran for two minutes, the hot spots were:

- 42% (56s): org.drools.reteoo.BaseLeftTuple.getSubTuple
- 18% (24s): org.drools.reteoo.NotNode.assertLeftTuple

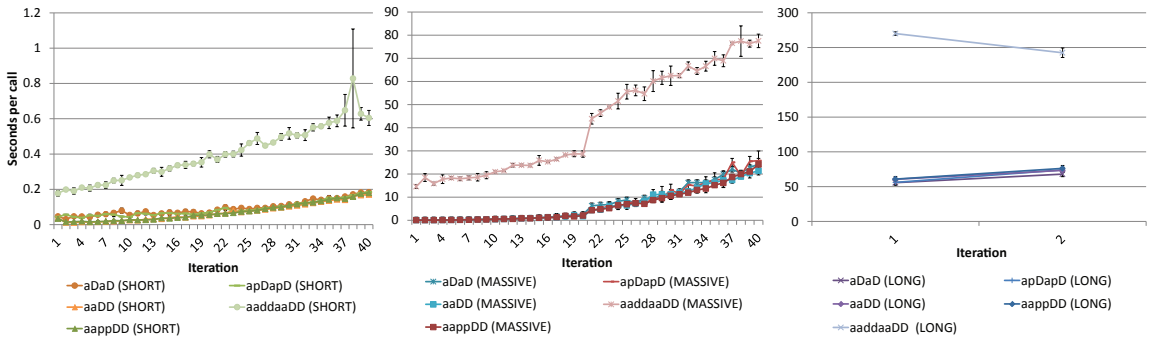

Figure 6.10. Deletion timing benchmarks.

- 15% (20s): org.drools.reteoo.BaseLeftTuple.hashCode
- 8% (11s): org.drools.core.util.LinkLinked$LinkedListFastIterator.next

In another test which ran for four and a half minutes, the hot spots were:

- 13% (28s): java.lang.Object.hashCode
- 11% (23s): org.drools.reteoo.BaseLeftTuple.getSubTuple
- 8% (18s): org.drools.reteoo.BaseLeftTuple.hashCode
- 7% (16s): org.drools.common.TupleStartEqualsConstraint.isAllowedCachedLeft
- 6% (13s): org.drools.reteoo.BaseLeftTuple.equals

Going further down the list, Drools functions accounted for a large majority of the time taken during API calls. The functions appear to be related to those used to test condition nodes within the graph used by the RETE algorithm. Although a relatively small number of rules get fired, the conditions which lead to those rules being fired appear to be what takes the most amount of time as the number of facts and complexity of rules is large. As noted above, performance degrades the most when many facts exist (e.g., many provisioned services) which have complicated conditional statements in the rules

CHAPTER 7

CONCLUSIONS

FlowOps is not only an abstract concept to improve network management and operations but is implemented and demonstrated to work in a reasonable time for the size of networks we are targeting at this point. Having knowledge at all layers in the model allows the framework to reason and work to create a value-added experience through integration of abstract, high-level definitions all the way down to bare-metal and back up.

7.1 Summary of Contributions

To summarize the contributions made, FlowOps:

- Includes a Knowledge Store which implements our layered model.
- Uses a Rules Engine to support handling tasks which flow through the layers of the Knowledge Store.
- Configures and handles faults through a Driver Engine with drivers for simulating faults or configuring OpenFlow-based switches.
- Was shown to successfully support configuration and fault management in an Open Access Network environment consisting of many users with multiple types of services.
- Performs reasonably fast on networks with a small number of switches.

7.2 Performance

All production systems must peform in a reasonable amount of time to be useful. FlowOps performance noticably starts to degrade as the number of network devices increases. Our evaluation determined that most of the time taken for API calls is due to the production rules system evaluating the conditions which trigger rules. For use with larger networks, the rules will need to be rewritten so that the time taken to evaluate all of the conditionals is minimal.

7.3 Future Work

More work will be needed to demonstrate that FlowOps can support multiple traditional switches using different drivers. Our tests and simulations so far have dealt with OpenFlow-enabled switches. The implementation will most likely need to be expanded to support more functions needed to bootstrap and manage these devices.

Network management is an extremely complex task and involves dealing with a vast field of issues and mixed environments. The initial implementation of FlowOps deals with the most important tasks: configuration and fault management. More types of provided and backbone services need to be added. For example, FlowOps should be able to support backbone networks using more than just VLANs like Multiprotocol Layer Switching, Provider Backbone Bridging, etc. Supporting these extra network types would certainly make FlowOps more robust and prove the usefulness in more scenarios.

Creating innovative services which run on top of the Open Access Network environment could further demonstrate the usefulness of FlowOps so that service providers and end users have incentives to migrate to a network operator providing such an environment.

The API has demonstrated useful for configuring and managing services as well as viewing state in the Knowledge Store. However, a more user-friendly service would be needed when deployed. A portal could be designed which interacts with the network operator and service providers using the described API in the background after users point and click through a series of pages.

Performance is currently a mixed bag. Networks tend to only have provisioned or deleted services at any point. Allocated services are temporary and quickly provisioned, deleted, or timed out so production systems should not encounter performance issues related to having many simultaneous allocations. On the other hand, provisioned services will grow over time which would be the bottleneck in our current system. Future enhancements related to lower API call times should be focused on handling multiple existing provisioned services.

REFERENCES

[1] Building a nationwide open-access network. http://www.bbpmag.com/2011mags/marchapril11/BBP_MarApr_NationwideOpenAccess.pdf.

[2] Cooperative model for innovative open-access telecommunications network in rural virginia. http://www.usda.gov/oce/forum/2008_Speeches/PDFPPT/Hudgins.pdf.

[3] Drools, the business logic integration platform. http://www.jboss.org/drools/.

[4] Floodlight. http://www.projectfloodlight.org/floodlight/.

[5] How amsterdam was wired for open access fiber. http://arstechnica.com/tech-policy/2010/03/how-amsterdam-was-wired-for-open-access-fiber/.

[6] Resource Specification (RSpec). http://groups.geni.net/geni/wiki/GeniRspec.

[7] UTOPIA wikipedia page. http://en.wikipedia.org/wiki/Utah_Telecommunication_Open_Infrastructure_Agency.

[8] Mef technical specification mef 6.1 - ethernet services definitions, phase 2.

[9] BRACKMAN, R. J., AND LEVESQUE, H. J. *Knowledge Representation and Reasoning*. Morgan Kaufmann, 2004.

[10] CHEN, X., MAO, Y., MAO, Z., AND VAN DER MERWE, J. KnowOps: towards an embedded knowledge base for network management and operations. In *Proceedings of the 11th USENIX conference on Hot topics in management of internet, cloud, and enterprise networks and services (Hot-ICE11), Berkeley, CA, USA* (2007), pp. 7–7.

[11] CHEN, X., MAO, Y., MAO, Z., AND VAN DER MERWE, J. Declarative configuration management for complex and dynamic networks. In *Proceedings of the 6th International COnference* (2010), ACM, p. 6.

[12] CHEN, X., MAO, Y., MAO, Z., AND VAN DER MERWE, J. DECOR: DEClarative network management and OpeRation. *ACM SIGCOMM Computer Communication Review 40*, 1 (2010), 61–66.

[13] CHEN, X., MAO, Z., AND VAN DER MERWE, J. PACMAN: a Platform for Automated and Controlled network operations and configuration MANagement. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies* (2009), ACM, pp. 277–288.

[14] CLARK, D., PARTRIDGE, C., RAMMING, J., AND WROCLAWSKI, J. A knowledge plane for the internet. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications* (2003), ACM, pp. 3–10.

[15] DAVIDS, J. The reggefiber model. `http://www.slideshare.net/INCA_NextGen/reggefiber`.

[16] EJ TECHNOLOGIES. Jprofiler. `http://www.ej-technologies.com/products/jprofiler/overview.html`.

[17] ENCK, W., MOYER, T., MCDANIEL, P., SEN, S., SEBOS, P., SPOEREL, S., GREENBERG, A., SUNG, Y., RAO, S., AND AIELLO, W. Configuration management at massive scale: system design and experience. *Selected Areas in Communications, IEEE Journal on 27*, 3 (2009), 323–335.

[18] ENNS, R. Netconf configuration protocol.

[19] FELTEN, B. Stockholms stokab: A blueprint for ubiquitous fiber connectivity? *Available at SSRN 2114138* (2012).

[20] FORGY, C. L. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence 19*, 1 (1982), 17–37.

[21] GREENBERG, A., HJALMTYSSON, G., MALTZ, D., MYERS, A., REXFORD, J., XIE, G., YAN, H., ZHAN, J., AND ZHANG, H. A clean slate 4d approach to network control and management. *ACM SIGCOMM Computer Communication Review 35*, 5 (2005), 41–54.

[22] HAYASHI, M., MATSUMOTO, N., NISHIMURA, K., AND TANAKA, H. Design of network resource federation towards future open access networking. In *AICT 2011, The Seventh Advanced International Conference on Telecommunications* (2011), pp. 130–134.

[23] KALMANEK, C. R., MISRA, S., AND YANG, Y. R. *Guide to reliable internet services and applications*. Springer, 2010.

[24] LANTZ, B., HELLER, B., AND MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (2010), ACM, p. 19.

[25] LEHR, W., SIRBU, M., AND GILLETT, S. Broadband open access: Lessons from municipal network case studies. In *Proceeding of the TPRC conference* (2004).

[26] LINDQVIST, U., AND PORRAS, P. A. Detecting computer and network misuse through the production-based expert system toolset (p-best). In *Security and Privacy, 1999. Proceedings of the 1999 IEEE Symposium on* (1999), IEEE, pp. 146–161.

[27] MAHIMKAR, A., YATES, J., ZHANG, Y., SHAIKH, A., WANG, J., GE, Z., AND EE, C. Troubleshooting chronic conditions in large ip networks. In *Proceedings of the 2008 ACM CoNEXT Conference* (2008), ACM, p. 2.

[28] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review 38*, 2 (2008), 69–74.

[29] NURMI, D., WOLSKI, R., GRZEGORCZYK, C., OBERTELLI, G., SOMAN, S., YOUSEFF, L., AND ZAGORODNOV, D. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on* (2009), pp. 124–131.

[30] PFAFF, B., PETTIT, J., KOPONEN, T., AMIDON, K., CASADO, M., AND SHENKER, S. Extending networking into the virtualization layer. *Proc. HotNets (October 2009)* (2009).

[31] ROUSKAS, G. N., BALDINE, I., CALVERT, K., DUTTA, R., GRIFFIOEN, J., NAGURNEY, A., AND WOLF, T. Choicenet: Network innovation through choice.

[32] STRUM, M., RICCI, R., VAN DER MERWE, J., CHRISTENSEN, J., AND PETERSON, R. Flowops: Open access network management and operation. Unpublished manuscript.

[33] VIRMANI, A., LOBO, J., AND KOHLI, M. Netmon: network management for the saras softswitch. In *Network Operations and Management Symposium, 2000. NOMS 2000. 2000 IEEE/IFIP* (2000), pp. 803–816.

[34] YAN, H., BRESLAU, L., GE, Z., MASSEY, D., PEI, D., AND YATES, J. G-rca: a generic root cause analysis platform for service quality management in large ip networks. In *Proceedings of the 6th International COnference* (2010), ACM, p. 5.