A Filesystem Abstraction for Multiple Actors in a Distributed Software Defined Network

by

Matthew Lawrence Monaco

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science

2013

This thesis entitled:
A Filesystem Abstraction for Multiple Actors in a Distributed Software Defined Network
written by Matthew Lawrence Monaco
has been approved for the Department of Computer Science

_____

Eric Keller

_____

Prof. Dirk Grunwald

_____

Prof. John Black

Date _____

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Monaco, Matthew Lawrence (M.S., Computer Science)

A Filesystem Abstraction for Multiple Actors in a Distributed Software Defined Network

Thesis directed by Prof. Eric Keller

Traditional networks are plagued with problems ranging from complexity to proprietary lock-in. This has led to environment in which human error is a major problem and innovation crawls at a snail's pace. Lately, the concept of software defined network (SDN) has gained real traction in academia and industry. While SDN helps to solve problems with traditional networks, it brings some new problems of its own. Furthermore, the introduction of SDN has left an open landscape for researches to define the design and makeup of tomorrow's networks.

This thesis describes the *Yanc* software defined networking controller. Yet Another Network Controller is a filesystem interface to a software defined network. Its main contributions are providing a low barrier to entry for network applications, allowing multiple network control applications to operate concurrently, and facilitating a software defined network application ecosystem in which multiple vendors' products may be assembled into a unified system.

Yanc's main, enabling component is a filesystem. This allows Yanc and the applications built on top of it to benefit from many existing technoligies which are already designed to work with the Linux virtual filesystem (VFS) layer.

A prototype is implemented and described. The core component is implemented in C on type of Filesystems in Userspace (FUSE). A number of ancilliary components are also implemented from native languages like C++ to scripting languages like Python.

Contents

Figures

Figure

10

Chapter 1

Introduction

The introduction of software-defined networks has generated tremendous buzz in the past few years as it promises to ease many of the network management head-aches that have been plaguing network operators for years [16, 21]. Software-defined networking uses a logically centralized control plane to manage a collection of packet processing and forwarding nodes in the data plane. It has been proposed that this requires an operating system for networks [23] which provides an interface to program the entire network. Applications on top of the operating system perform the various management tasks by utilizing the operating system's interface. At a high level, an OS manages the interface with hardware (network devices) and allows applications (network management tasks) to run.

Despite a useful analogy, the practical realization is that while extensible, the current SDN controllers [23, 3, 8, 38, 10] are geared towards single, monolithic network applications where developers can write modules in the supported language using the API provided by the framework, compile the entire platform, and run as a single process. An alternate approach is to use new languages and compilers that allow programmers to specify the application with a domain specific language and run the compiled executable, still as a monolithic application [15, 32]. Among the downsides of a monolithic framework is that a bug in any part of the application (core logic, a module, etc.) can have dire consequences on the entire system.

Moreover, each of the existing controllers end up independently needing and developing a similar set of required features. This results in a fragmented effort implementing common features

where the main distinguishing aspect in each case is commonly the language in which applications are allowed to be written (**e.g.,** NOX–C++, Ryu–Python, Floodlight–Java, Nettle–Has-kell, etc). Further, these controllers are missing important features like the ability to run across multiple machines (a distributed controller) – limited to a hot standby (in the case of Floodlight) or a custom integration into a particular controller (in the case of Onix [27] on top of NOX). Even support for the latest protocol is lacking; many have yet to move past OpenFlow 1.0 for which newer versions have been released, the latest being 1.3.1 [16] — even Floodlight, a commercially available controller, only supports 1.0 [1].

This thesis explores the question: Is a network operating system fundamentally that different from an operating system such that it requires a completely new (network) operating system? I argue that instead of building custom SDN controllers, we should leverage existing operating system technology in building an SDN controller. Our goal is to extend an existing operating system (Linux) and its user space software ecosystem in order to serve as a practical network OS.

1.0.1 Yanc

The initial design of *yanc*[1] , which effectively makes Linux the network operating system and is rooted in UNIX philosophies (§5). The *yanc* architecture, illustrated in Figure 1.1, builds off of a central abstraction used in operating systems today – the file system. With *yanc*, the configuration and state of the network is exposed as file I/O (§6) – allowing running application software in a variety of forms (user space process, cron job, command line utility, etc) and developing in any language. Much like modern operating systems, system services interact with the real hardware through drivers, and supporting applications can provide features such as virtualization, or supporting libraries such as topology discovery (§8). By using Linux, we can leverage the ecosystem of software that has been developed for it (§5). One special example that is made possible by building **yanc** into an existing operating system is that distributed file systems can be layered on top of the *yanc* file system to realize a distributed controller (§9). Finally, while *yanc* is

[1] *yanc*, or yet another network controller.

mostly discussed in terms related to the OpenFlow protocol for ease of understanding, the design of *yanc*, extends into more recent research, going beyond OpenFlow (§10).

Once fully implemented, **yanc** will enable researchers to focus on value-added applications instead of yet another network controller.

Yanc enables an ecosystem similar to Linux and its many distributions. Linux itself is only an operating system kernel. The distributions collect software from many sources and provide a fully function operating system targeted at different audiences. For example there are distributions for developers, home users, corporations, security researches, embedded systems, etc. Networks are similar; *yanc* provides a common core, but it is not possible to provide a one-size-fits-all solution for home networks, college campuses, data centers, etc. Rather, *yanc* allows an administrator to assemble a network controller appropriate for the given environment.

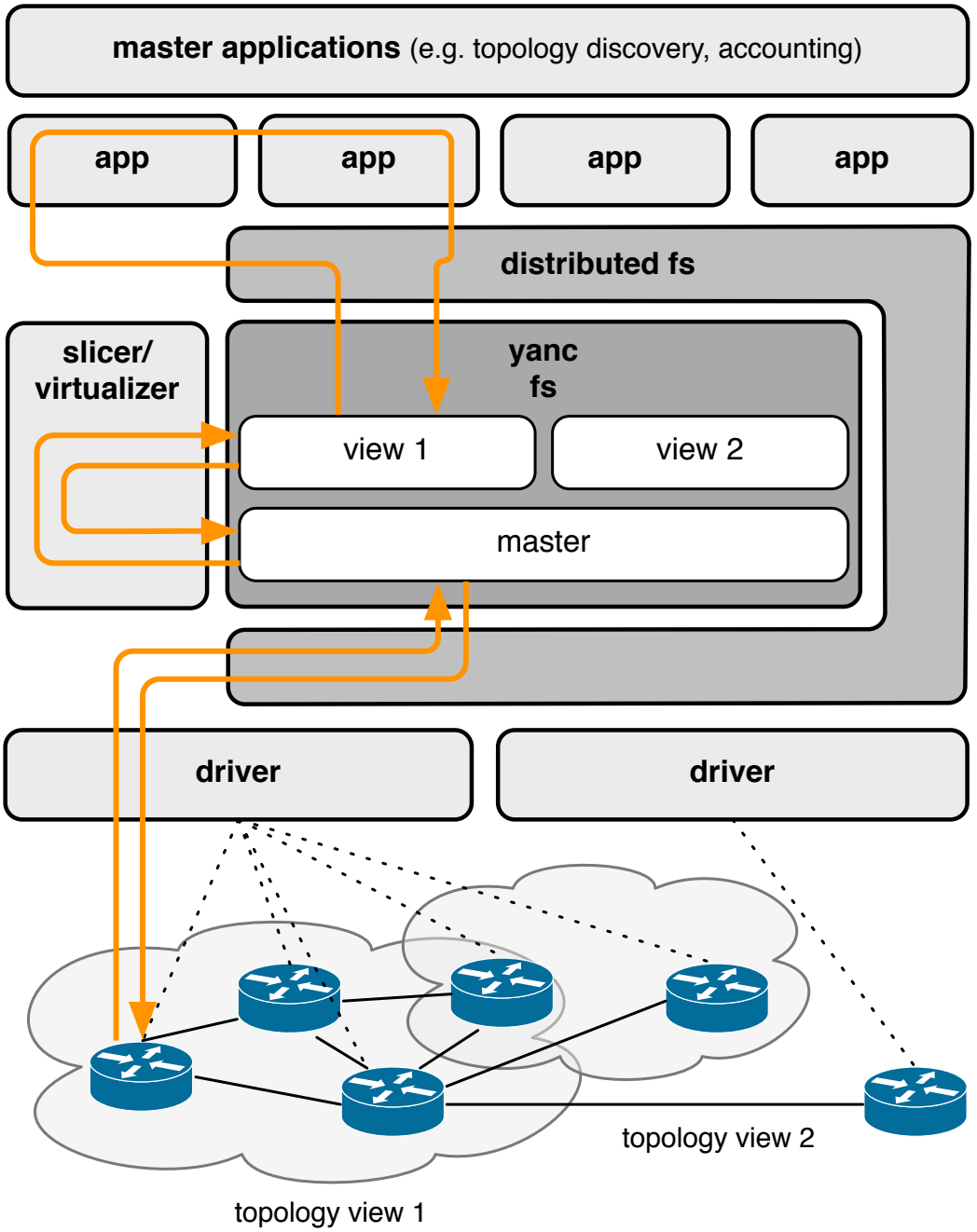


Figure 1.1: System Architecture

Chapter 2

Background

Traditional networking has been around for decades. It is used in a wide range of settings such as homes, small offices, enterprises, college campuses, and datacenter. The most commonly used networks are built up of Ethernet and IP from many different types of hardware devices and topological organizations.

The complexity of traditional networking causes many problems in design, administration, efficiency, and scalability. Therefore, a new paradigm has been growing in popularity: software defined networking (SDN). In the SDN paradigm, the network data plane and control plane are separated. The control plane (the sore spot of traditional networks) is moved to a logically centralized software controller. This allows centralized applications to manage the network based on a global view of network state.

2.1 Traditional Networking

Traditional networks are plagued with many problems. For example, Figure 2.1 illustrates a highly complex corporate network. In it, there are many different types of devices such as switches, routers, firewalls, and a range of middle boxes. The topology is also highly irregular because there are different locations of various sizes and different requirements and uses at each location.

2.1.1 OSI Seven Layer Model

To help reason about traditional networks, the OSI seven layer model (Figure 2.2) is used to divide a network up in layers, each with their own use and set of protocols.

Layer 1: Te physical layer consists of physical connections such as Ethernet (IEEE 802.3) wiring or Wifi (IEEE 802.11) wireless connections. Fiberoptics is also a popular physical layer medium.

Layer 2: The data link layer is responsible for moving frames of data around the networking and performing media access control (MAC) so that many end-hosts can operate on a shared medium. This layer is most typically Ethernet[1] , however other standards such as Infiniband do exist. A typical Ethernet header is shown in Figure 2.3. Other than Ethernet itself, a commonly used layer 2 protocol is the Address Resolution Protocol (ARP) which is used to translate between layer 2 and layer 3 addresses.

Layer 3: The network layer creates a network of networks. It uses the Internet Protocol (IP) to create one large, global network of individual layer 2 networks. Figure 2.4 shows a standard IPv4 header. Unlike the source and destination address shown in Figure 2.3, IP addresses were meant to be globally unique. However, due to its widespread use, a range of private addresses is also supported.

Layer 4: The transport layer is responsible for determining how data is transported between two endpoints. One simple method, the User Datagram Protocol (UDP) simply sends packets of data from a source to destination without and acknowledgements or delivery guarantees. Because UDP is very simple, its header doesn't contain much information (Figure 2.5).

On the other hand, the Transmission Control Protocol (TCP). Is far more complex. It establishes a stream of data (rather than packets as with UDP) and uses a three-way handshake so each endpoint can be sure data has been transmitted successfully. Furthermore, TCP employs congestion control for automatically throttling transmission speed in the event of network bottlenecks.

Layer 5: The session layer is sometimes used to establish session-level state between

[1] Ethernet (IEEE 802.3) specifies both the physical and data link layers

endpoints. For example, SSL/TLS operates on the session layer and is responsible for encrypting all traffic between two endpoints.

Layer 7: The application layer is where meaningful data between programs is transmitted. There are many application layer protocols such as the Secure Shell protocol (SSH), Network Time Protocol (NTP), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), Simple Network Management Protocol (SNMP), Domain Name System (DNS), and many more. The Hypertext Transfer Protocol (HTTP) is one of the most popular application layer protocols. HTTP was originally intended for web pages, but has also begun to be used for exposing RESTful APIs to **e.g.,** mobile applications.

2.1.2 Network Devices

There are a number of distinct network devices used in traditional networks. Each one tends to focus on a single layer, or even single protocol, in the network stack.

Switches are typically Ethernet devices which connect to hosts and other switches and form a single layer-2 network. They forward packets based on learning the location of hosts by examining source MAC addresses (Figure 2.3). Switches also speak the Spanning Tree Protocol (STP) to build a single, loop-free topology of all connected switches.

Rotuers are IP (layer-3) devices which connect Ethernet (or other layer-2) networks. They forward packets based on routing rules which are configured by an administrator or automatically discovered with distributed protocols such as the Border Gateway Protocol (BGP) or Interior Gateway Protocol (IGP).

Firewalls are used to block (and sometimes translate) traffic based typically on layer-2 through layer-4 headers.

Middleboxes are hardware appliances which do pretty much everything else on the network. They include HTTP proxies, intrusions detection systems (IDS), virus scanners, etc.

2.1.3 Challenges

Traditional networks are comprised of many layers, devices, (distributed, non-deterministic) protocols, and arbitrary topologies. This makes configuring and managing them an extremely complex and delicate procedure.

2.1.3.1 Efficiency

Even with a highly regular topology such as one would find in a datacenter, it is very difficult to obtain an efficient utilization of purchased hardware because the distributed spanning tree protocol will render many links unused. This is because traffic from one source to destination will always go over a single path which is calculated by STP, even when multiple paths may exist.

2.1.3.2 Complexity

For irregular topologies such as on a college campus, hardware may be purchased to more or less match the latency and throughput requirements of different buildings and departments, but end-to-end communication can become very difficult to manage because of the scattering of independent hardware around many buildings.

2.1.3.3 Vendor Lock-In

Furthermore, these independent devices from different vendors such as Cisco, Juniper, HP, Dell, and many others have proprietary interfaces mixing hardware from multiple vendors adds a degree of complexity to managing multi-vendor network.

2.1.3.4 Non-determinism and Independence

Because most network devices are independent and speak distributed, non-deterministic protocols, configuring and troubleshooting a network can be very difficult. Many network failures are simply caused missing configuration in one of many devices along a particular path.

Non-deterministic protocols such as STP and BGP also make it very difficult to predict how a network will behave, reproduce errors, and investigate problems.

2.1.3.5 Security

Independent configuration of many distributed devices also leads to security holes. Managing a single firewall is not difficult, but managing many around a campus network with network address translation (NAT) devices which could rewrite packets and thus unknowingly subvert the effects of important firewalls is a delicate task.

2.1.3.6 Human Error

Human error is the leading cause of network issues. There is a lot of room to make mistakes because there are many configuration items for even a single device and making changes that affect the entire network will often require a human to manually make many changes.

Furthermore, because of the complexity of networks, different layers of a large network are often managed by distinct teams. Making changes which affect the jurisdiction of multiple teams requires a lot of coordination so that they can be applied correctly and consistently.

2.2 Software Defined Networking

Software defined networking (SDN) is a new networking paradigm which addresses many of the challenges with traditional network technology. Figure 2.6 shows a high level overview of the software defined networking concept. The key idea is that the network data plane and control plane are separated. The control plane is moved to a logically centralized commodity server (or servers) and the dataplane is exposed through an open API.

An SDN switch is basically a traditional switch, router, firewall, etc combined into a simple piece of hardware without a proprietary software stack and is implemented using Tricontent Addressable Memory (TCAM). SDN switches perform (wildcard) matches against incoming packets, keep some statistics, and apply actions based on matches.

An SDN controller programs the match-actions on a switch using an open API. Controllers are logically centralized, meaning they have a global view of the network but can be distributed among many machines for resilience, performance, and administration. Controllers also export a *northbound* API so that applications can encapsulate logic such as firewalling, routing, congestion control, etc and make changes to the network.

2.2.1 History

Software defined networking has roots all the way back to the 1990's with Active Networks [37]. However in its modern form SDN begin with *A Clean Slate Approach to Network Management* [21] published at SIGCOMM 2005. *Ethan: Taking Control of the Enterprise* [11] was then published at SIGCOMM 2007. Finally *OpenFlow: Enabling Innovation in Campus Networks* [31] was published in SIGCOMM CCR in 2008.

SDN is also an extremely active area of research and development with many papers being published at NSDI, SIGCOMM, OSDI, HotNets, HotSDN, and others. The main topics include low-level controller design, northbound API, novel techniques for network control using SDN applications, controller distribution, and application composition.

2.2.2 OpenFlow

Following the SIGCOMM CCR OpenFlow paper, an official version, v1.0.0, was published in December of 2009 [17]. The latest version, v1.4.0, was released in October of 2013 [18].

A basic OpenFlow flow entry is show in Figure 2.7. A flow entry, or rule, consists of a match, action, and statistics. The match can be a wildcard or exact match across network layers 1 through 4. Match fields are based on the physical input port of a packet, Ethernet headers, IP headers, and TCP/UDP headers.

An OpenFlow flow entry action can be one or more of dropping, sending out a particular port, flooding out of all ports, sending to the controller to process further, and rewriting fields such as addresses and ports.

Flow entries also contain some per-entry statistics such as packets in and out and bytes in and out. They also contain timestamps for removing flow entries after a fixed period of time or a period of inactivity.

The protocol itself consists of a number of message types which are used to inspect and configure switch state. There are a set of messages which a controller can use to get and set switch features and configuration settings, a PACKET_IN and PACKET_OUT message for switches to send packets to the controller and the controller to send packets to a switch, and most importantly, a FLOW_MOD message used by the controller to add, modify, and delete flow entries on a switch.

2.2.3 SDN Effects

Software defined networking, and OpenFlow in particular, have simplified networking hardware by effectively flattening OSI layers 1 (physical) through 4 (transport). By exposing the fundamental networking hardware through an open API, SDN has greatly reduced proprietary vendor lock-in and allowed smaller companies to produce hardware at low cost and with high raw performance.

SDN has also enabled more powerful network control by doing away with distributed and non-deterministic protocols. Datacenters now have an opportunity to fully utilize their networking infrastructure by using multiple paths and making per-flow decisions based on a global network view.

2.2.4 SDN Challenges

While software defined networking solves more problems than it creates, it still brings some new challenges. By virtue of being new researchers, developers, and administrators are still figuring out how to best make use of SDN. There are quite a few SDN controllers that have been developed which offer different APIs for programming the network. However, the *northbound* API — the API which controllers expose to applications — is still being refined. For example, do applications need to know about every element of the network, or should they only be concerned with the edge.

Likewise, should applications know about individual flow entries, or end-to-end flows?

Composition is a major challenge in software defined network. How should multiple, logically distinct applications and configurations be composed into a single network? For example, how can routing rules and load balancing rules be written separately and not interfere with one another? Or, how can firewall rules and NAT rules be written such that there aren't security holes?

Another major challenge is that of distribution. When we talk about an SDN controller we stress that it is *logically* centralized. In fact, most production controllers are completely centralized as a single process. However in theory, we can leverage multiple physical servers in a control plane to increase fault tolerance and efficiency but the locking and consistency models for distributed controllers are still being researched.

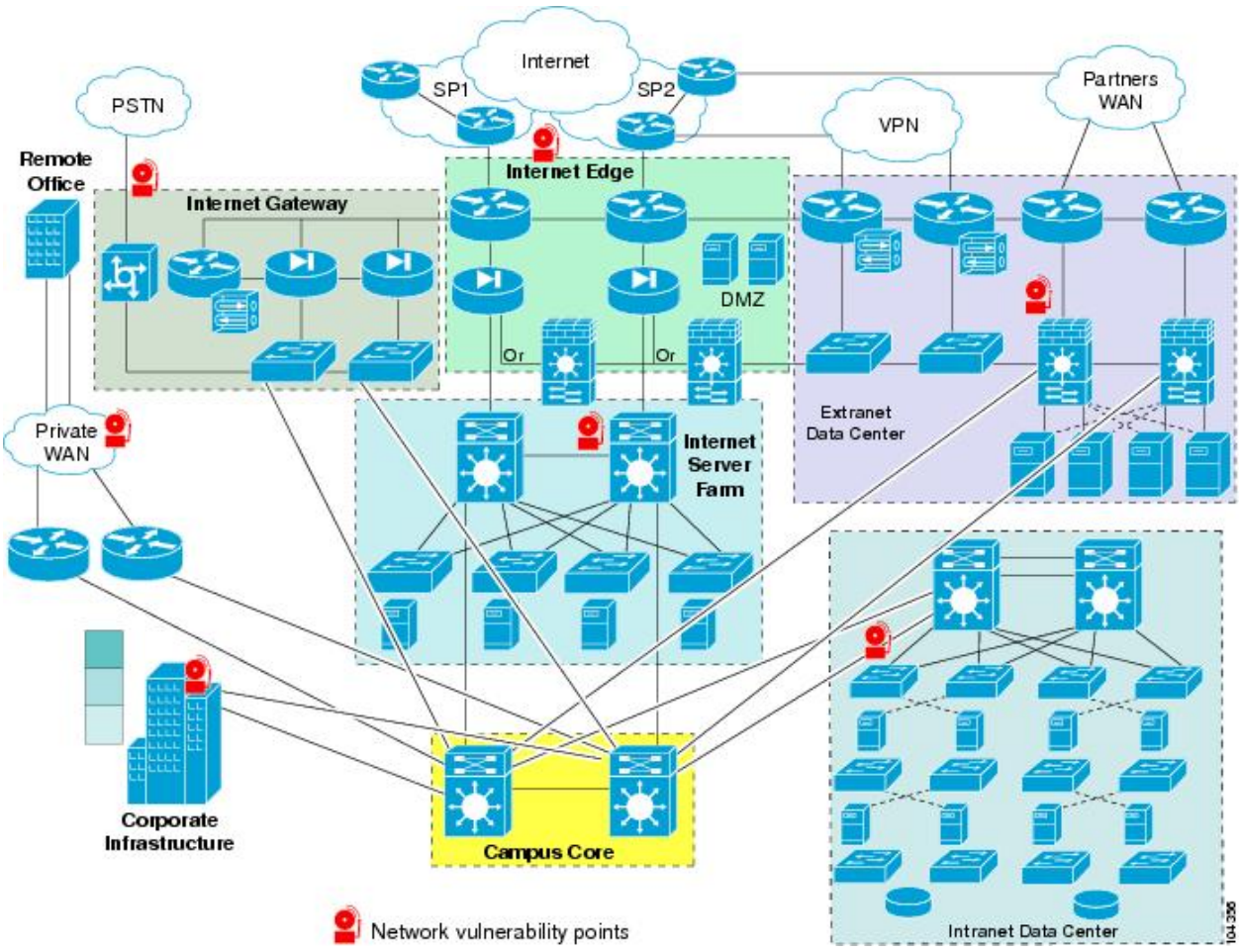


Figure 2.1: A traditional and complex network.

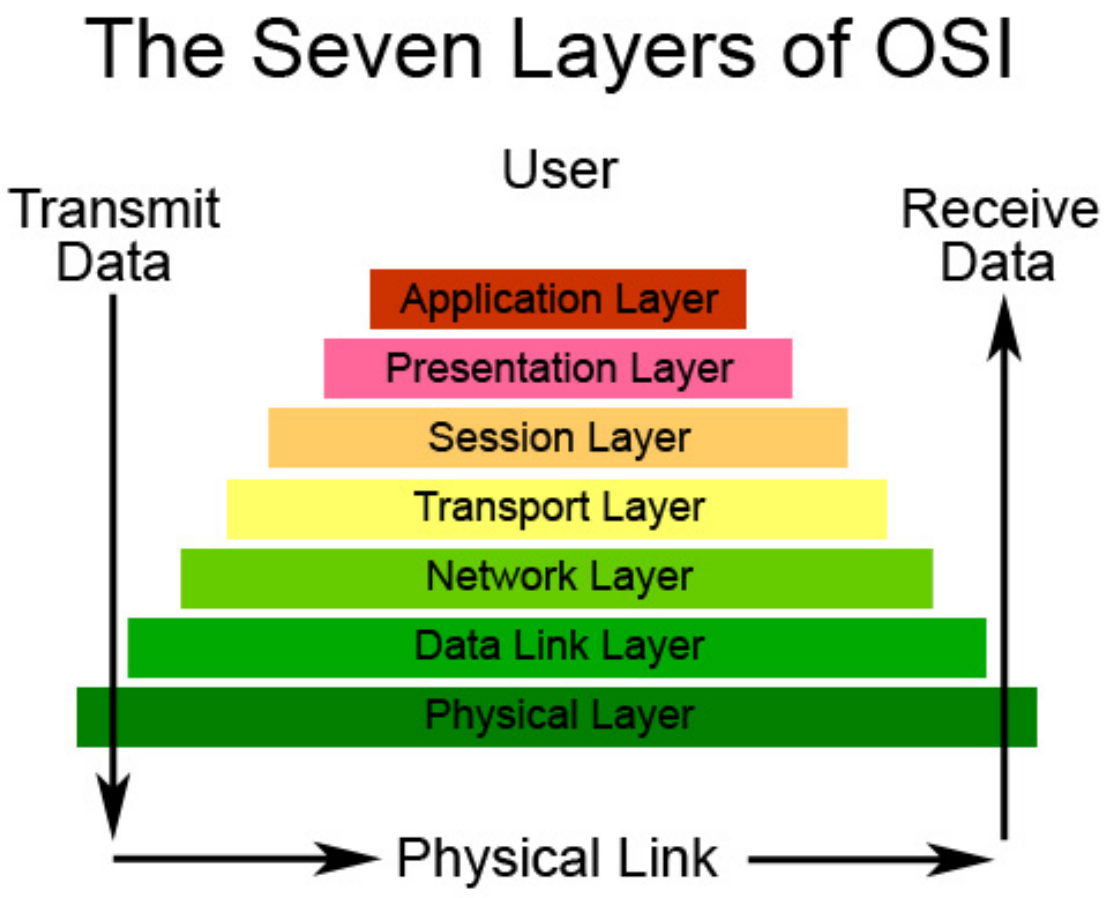


Figure 2.2: The OSI seven layer model. http://www.washington.edu/lst/help/computing_fundamentals/networking/osi

Word Offset	Byte 0	Byte 1	Byte 2	Byte 3
0x0000	Destination MAC Address			
0x0010			Source MAC Address	
0x0020				
0x0030	Type (Level 2)			

Figure 2.3: An Ethernet (IEEE 802.3) header. http://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/f2011/mis47_ayg6/mis47_ayg6/

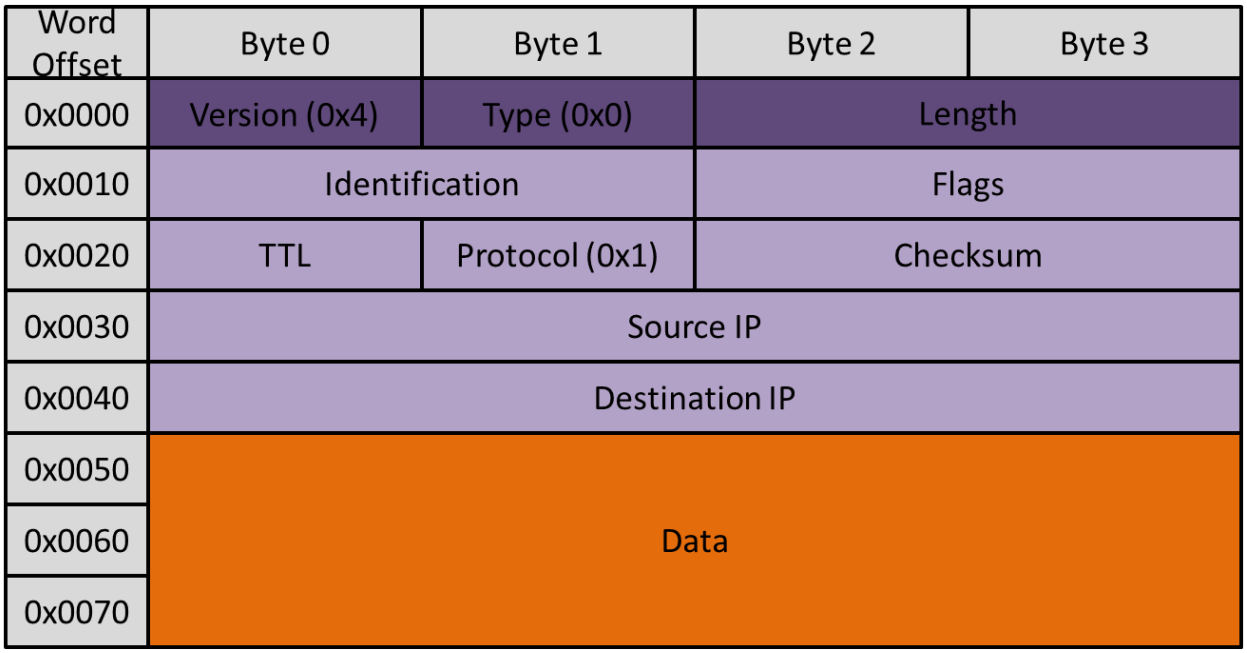


Figure 2.4: An Internet Protocol (IP) header. http://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/f2011/mis47_ayg6/mis47_ayg6/

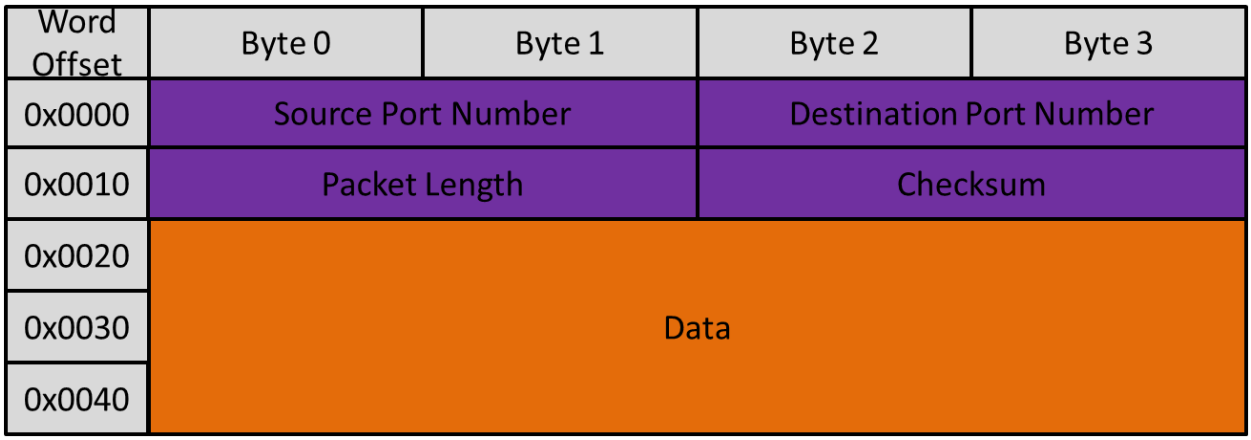


Figure 2.5: A User Datagram Protocol (UDP) header. http://people.ece.cornell.edu/land/courses/ece5760/FinalProjects/f2011/mis47_ayg6/mis47_ayg6/

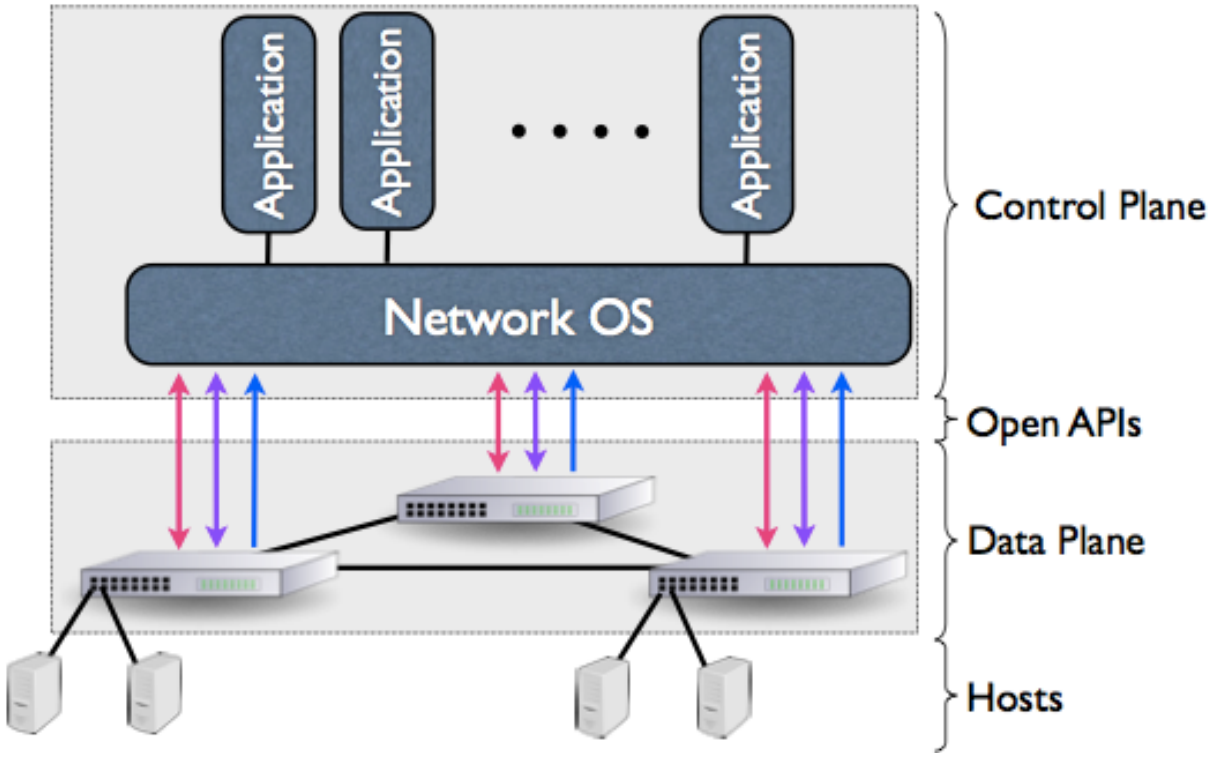


Figure 2.6: A high-level view of a software defined network (SDN). http://blog.sflow.com/2012/05/software-defined-networking.html

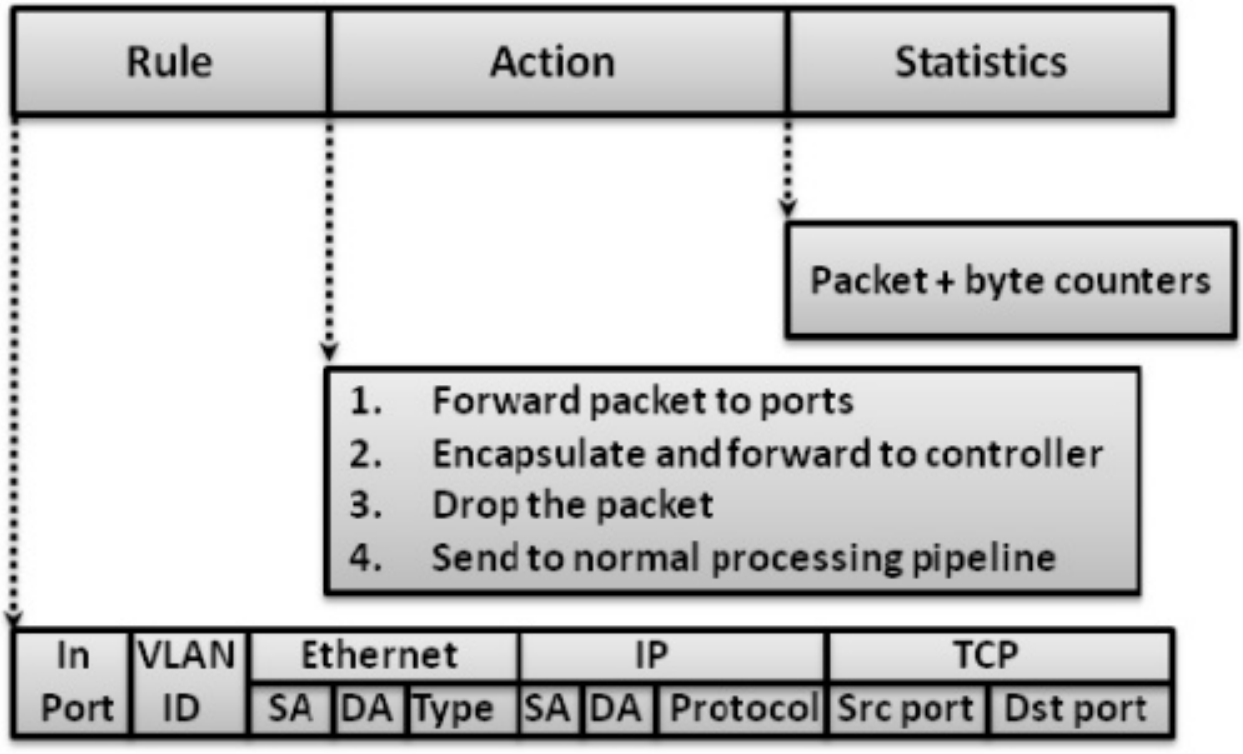


Figure 2.7: An OpenFlow flow entry. http://www.opticsinfobase.org/oe/fulltext.cfm?uri=oe-19-26-B421&id=224728

Chapter 3

Related Work

As a software defined networking controller, there is quite a bit of related work to *yanc* in three main categories. First, SDN research itself has been ongoing since the 1990's [37] and becoming very active recently [31, 11, 12, 20, 21]. Second, SDN controller research and development itself [3, 8, 10, 6, 23, 30]. Third, are specific SDN applications [36, 33, 25, 39].

3.1 SDN Research

The concept of *active networks* has been around for some time [37]. In an active network, packets carry executable source code to be executed on network elements as the packet moves around the network. This design was criticized for security issues as well as suffered from low performance.

In 2005, *A Clean Slate Approach to Network Control and Management* [21] was published at SIGCOMM Computer Communications Review. This paper isolated the data plane from three components of a control plane: decision, dissemination, and discovery. The modern SDN model groups these latter three *dimensions* into a single control plane.

SANE: A Protection Architecture for Enterprise Networks [12] divides networks into a data plane and comprehensive control plane, but only for certain network operations. In particular, it focused on access control and routing.

Following from *SANE* the same group published, in 2007, *Ethane: Taking Control of the Enterprise* [11]. This design builds on those from *SANE* and offers a full software defined network

control plane for all aspects of the network.

A simple software defined network, *VL2* [20] by Microsoft at SIGCOMM in 2009. This paper describes a network with a software control plane which effectively creates a single *one-big-switch topology* and uses intelligent flow control for taking advantage of multiple paths available in the network fabric.

3.2 SDN Controllers

There are a number of software defined networking controllers that have been designed and released. The most well-known is Floodlight [3] which is a Java-based monolith controller. Applications built on top of Floodlight are implemented as modules directly in the Floodlight source tree. Similar to Floodlight is Open Daylight [6], which is a fork of Floodlight and related to the Linux Foundation.

Another set of related controllers is NOX [23] and POX [30]. NOX is a C++ controller. Administrators extend base classes provided by NOX in order to customize control of the network. POX is similar to NOX but implemented in Python.

Ryu [8] is another SDN controller which is implemented in Python. Developers use the Ryu library to register handlers for various types of events on the network.

What all of these controllers have in common is that they result in a single, monolithic (Figure 3.1) controller. The projects themselves simply provide a starter-kit for creating an SDN.

3.3 SDN Applications

Since the publication of the OpenFlow protocol, there have been a number of important SDN applications. A controller must support the logic and algorithms of these applications in order to be viable.

Flowvisor [36] is a proxy which sits between OpenFlow switches and one or more OpenFlow controllers. It allows the control plane to be sliced and managed by independent parties. Because Flowvisor operates by intercepting and rewriting OpenFlow messages, a controller does not need

to support it directly. However it would be reasonable for a controller to implement the Flowvisor algorithm natively.

FortNox [33] is an application which prevents flow entries from undermining one another. For example, a high priority flow entry might deny traffic from the Internet to port 22 on a particular host. Another flow entry might NAT some public traffic to a local address and allow undesirable traffic on port 22. ForNox would cause the second flow entry to result in an error.

Header Space Analysis [25] is similar to FortNox. It provides static flow entry analysis, whereas FotNox provides real-time checking

Finally, OFRewind [39] is a tool for systematically logging and replaying traffic on an OpenFlow network. It allows for detailed analysis of traffic after **e.g.,** an attack or mysterious behavior.

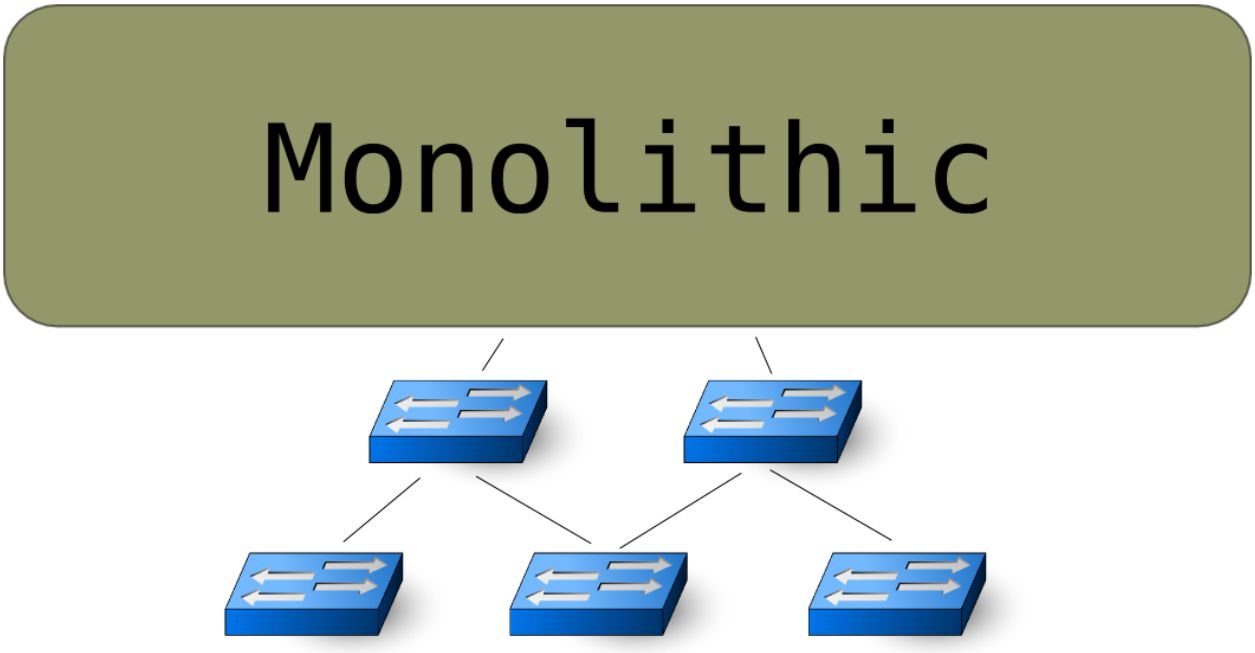


Figure 3.1: A monolithic SDN controller such as Floodlight, NOX, and Ryu.

Chapter 4

Yanc: Yet Another Network Controller

The basic principle of *yanc* is that everything is a file. This concept dates back to the original UNIX operating system [35] and is a powerful operating system primitive. While holes are inevitably poked in this abstraction for performance reasons, the abstraction is a solid base for designing an operating system (for networks). In *yanc*, for example, switches and flow entries are represented as directories and their sub-elements and fields are represented as sub-directories and files.

Yanc is in the same vain is procfs and sysfs in the Linux operating system [14]. Procfs allows users and programs to inspect and manipulate process and kernel state. Sysfs allows users and programs to inspect and manipulate device state. Likewise, *yanc* allows users and programs to inspect and manipulate network state.

Central to *yanc* is exposing network configuration and state as a file system. This decision stems from the fact that file systems are central to modern operating systems and enabling interaction with network configuration and state through file I/O enables a powerful environment for network administration. This follows from the file system abstraction providing a common interface to a wide range of hardware, system state, remote hosts, and applications. On Linux, ext4 controls access to block devices, procfs to kernel state and configuration, sysfs to hardware, and nfs to remote file systems.

A high-level overview of *yanc* is depicted in Figure 4.1. *Yanc* consists of many independent applications which coordinate via a filesystem implemented on top of the Linux kernel. Linux, the

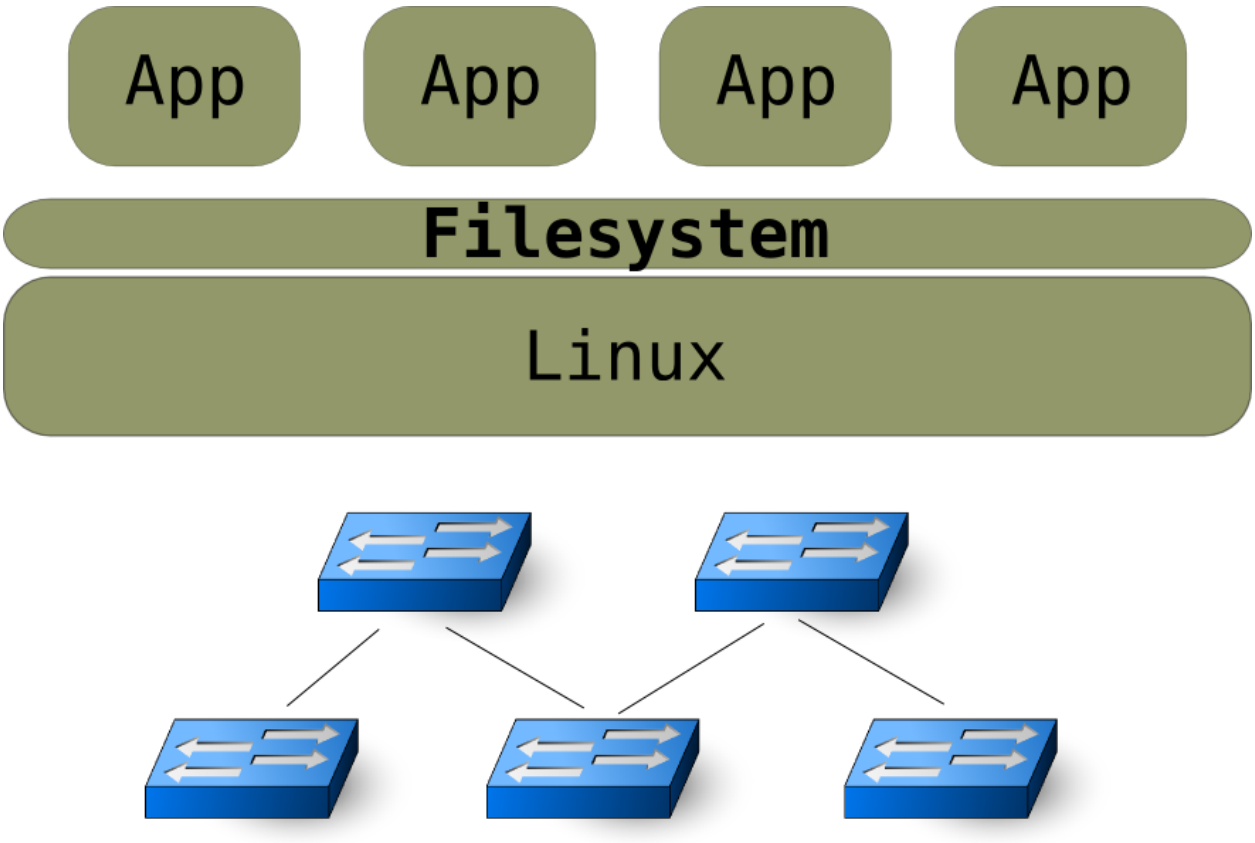


Figure 4.1: A high-level overview of the Yanc SDN controller.

yanc filesystem, and the collection of applications make up software defined network controller.

Chapter 5

Why a Filesystem?

There are a number of technical reasons for choosing a filesystem abstraction for a software defined network beyond simply being a very robust API for systems development.

5.1 Logically Distinct Applications

A filesystem assumes simultaneous access by independent entities. This is an important benefit of *yanc*. Rather than creating a monolithic SDN controller as described in chapter 3, a *yanc* controller can be made up of independent applications. An example of this is show in Figure 5.1. In this figure, there is a firewall application, a learning switch application, an application for introspection, and an application to facilitate slicing.

5.2 Independent Development

Because applications can be run independently, they're easier to develop independently. As show in Figure 5.2, different SDN applications can be developed by different parties such as a university, Apache Software Foundation, and The Open Networking Foundation (ONF). However, they still operate on the same network control plane.

From this, administrators and developers are able to obtain *yanc* applications through various methods. For example some software can be installed via the system's package manager:

```
# apt-get install yanc-learning-switch
```

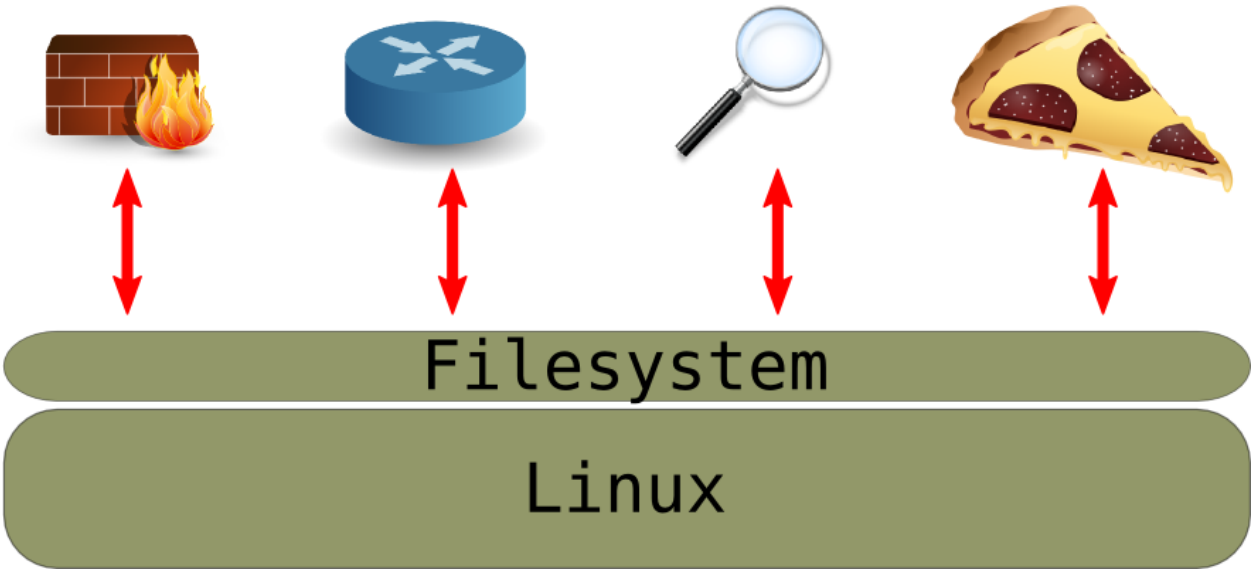


Figure 5.1: *Yanc*'s logically distinct applications. In this example, a firewall, learning switch, logger, and slicer.

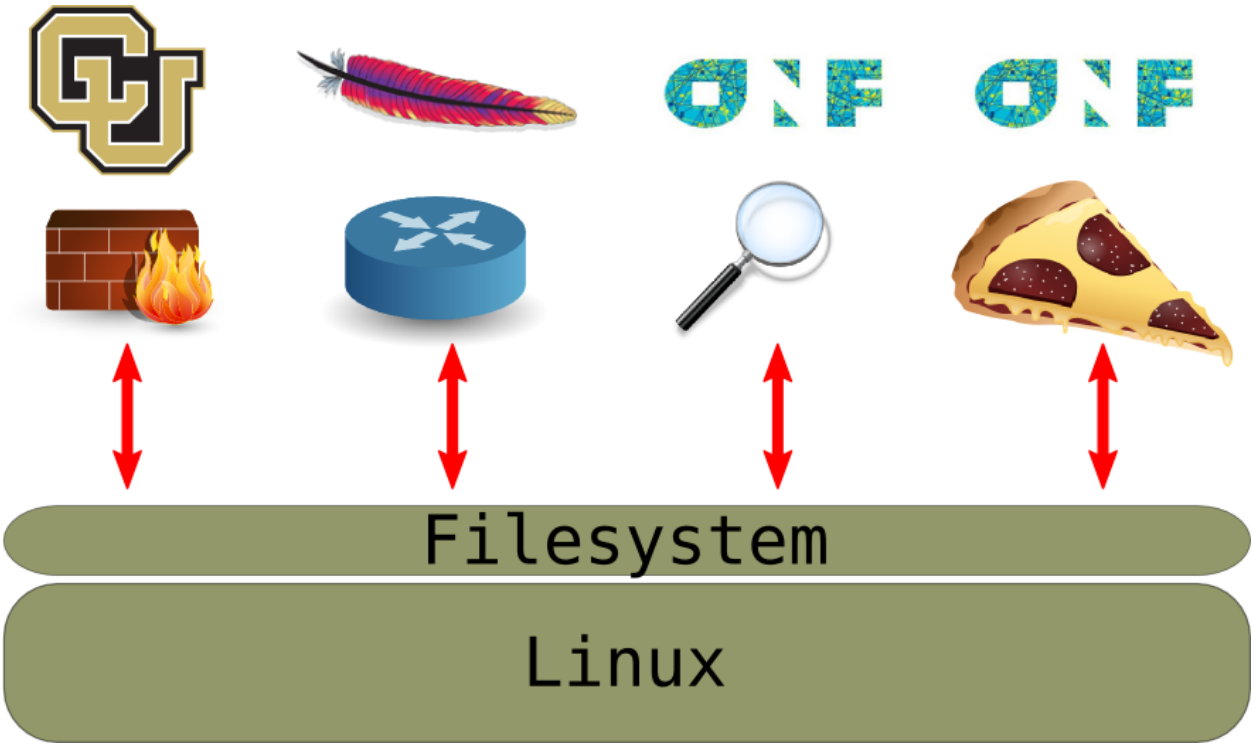


Figure 5.2: *Yanc*'s logically distinct applications, developed by independent sources such as a university, the Apache Software Foundation, and the Open Networking Foundation.

```
# apt-get install yanc-router
```

while others can be retrieved from source and compiled:

```
# git clone git@github.com/mmonaco/yanc-firewall
# cd yanc-firewall
# make
# make install
```

5.3 Language Flexibility

A filesystem is a universal interface. File I/O is supported by almost all programming languages. As shown in Figure 5.3, this includes native languages like C and C++, shell scripting languages such as bash, interpreted languages such as Python, and JITed languages such as Java.

5.4 Design Flexibility

Yanc offers maximum flexibility in the overall architecture of SDN applications (Figure 5.4. One size does not fill all when it comes to an SDN application. Some make sense to run constantly as daemons. Others are more suited to periodic tasks (cron jobs). And others, are best left as ad-hoc utilities which can be run when a human administrator decides to.

5.5 Other Technologies

By virtue of the fact that *yanc* is a filesystem, it and its application benefit automatically from technologies which were not necessarily written with SDN in mind.

5.5.1 Inotify

Inotify (and its successor fsnotify) is a Linux kernel system for userspace application to register for filesystem events. Such events are a file being opened, a new file being created, a file

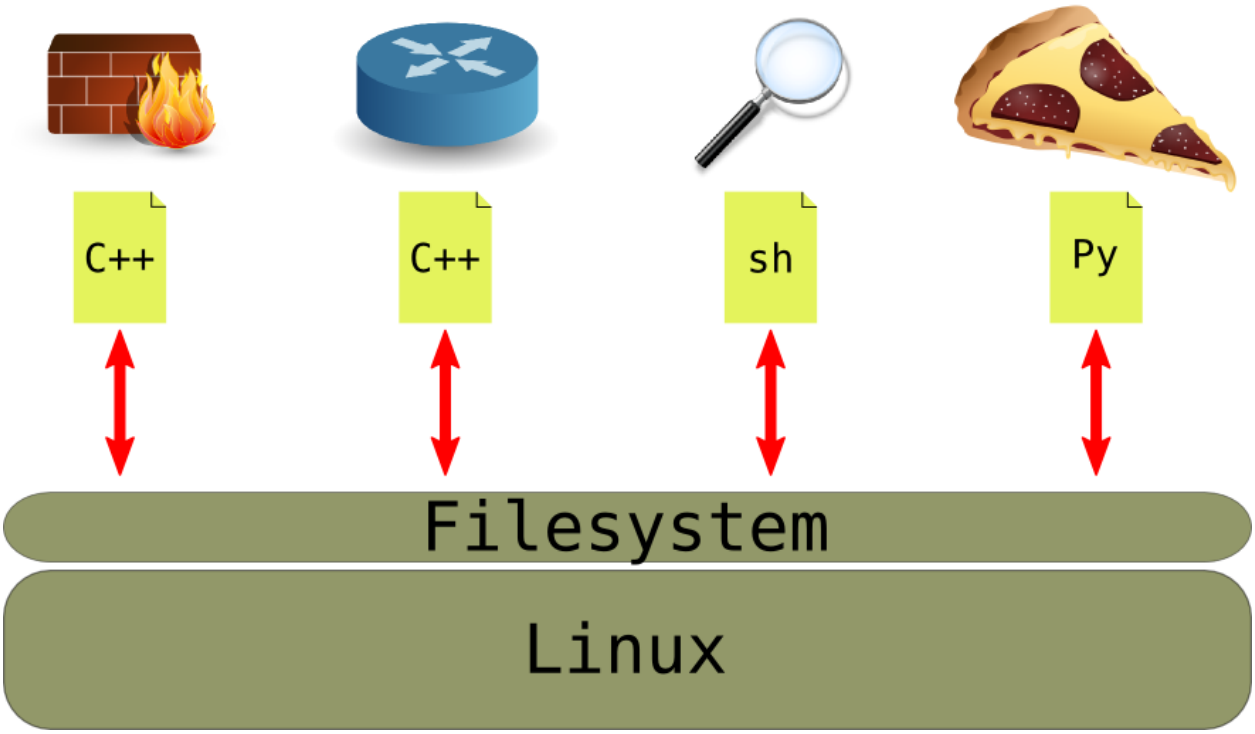


Figure 5.3: A *Yanc* application is only limited by the filesystem API, which is supported by almost all programming languages.

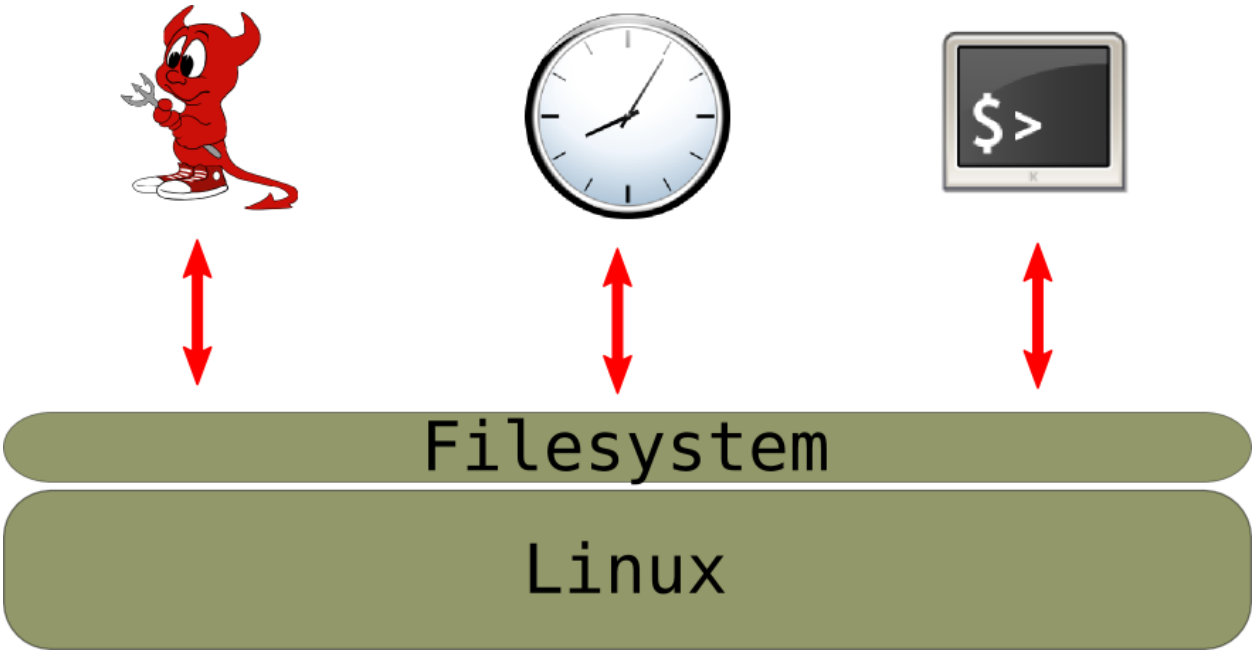


Figure 5.4: A *Yanc* application can be designed in different ways. This includes daemons, periodic tasks, and ad-hoc utilities.

which was opened for writing is closed, and so on. Applications can then use the `select()` system call to wait for events without manually polling.

Yanc applications can make us of inotify to monitor for changes and events in the network. *Yanc* itself, nor its applications are required to implement their own event notification system.

5.5.2 File Permissions

Yanc uses file permissions and access control lists (ACLs) for privilege separation on the network. By using file permissions, an administrator can restrict an application's ability to interact with subsets of the network.

For example, and administrator can run an application under a user account which only has permissions to read from the entire network. This would be appropriate for a logging application to prevent accidental changes.

5.5.3 Namespaces and Control Groups

Linux has recently developed two powerful technologies for even more isolation than what is provided by permissions and ACLs. The first, namespaces, allow resources to be entirely isolated for specific processions. *Yanc* can use namespaces for running tenant applications securely and in isolation from the host system as well as other tenants.

The second technology, control groups, can be used to limit the resources that an application (or set of applications) can use. Control groups complement namespaces by providing resource guarantees and preventing starvation among applications.

5.5.4 Layered Filesystems

Because all filesystems implement the same API — the Linux virtual filesystem (VFS) layer — they are easily layered on one another. A layered filesystem intercepts standard calls such as `read()` and `open()`, performs some optional operations, and then passes the calls to the underlying filesystem.

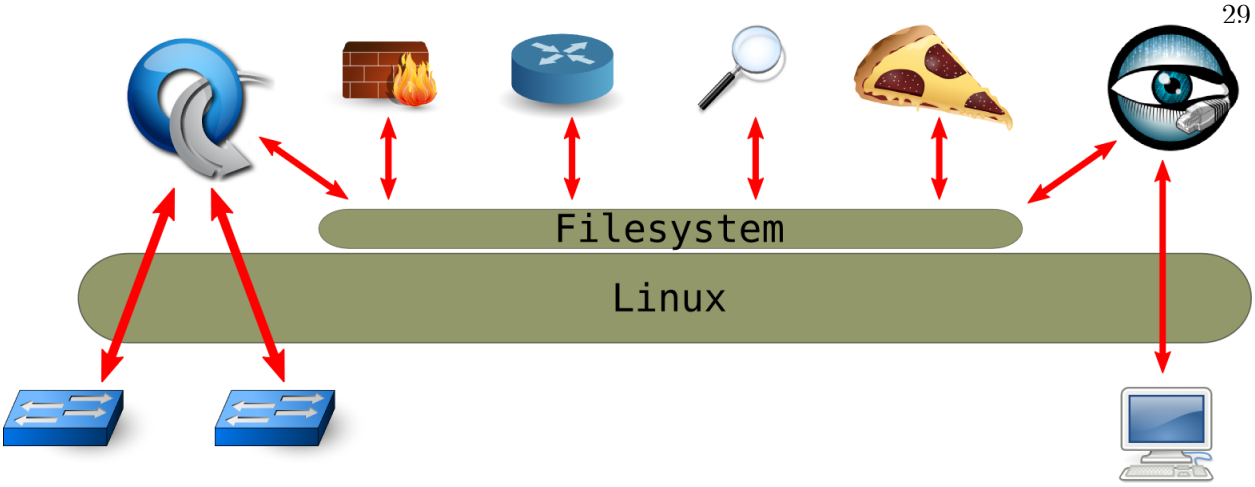


Figure 5.5: *Yanc* applications are decoupled from hardware. Driver applications such as OpenFlow and Bro speak hardware-specific protocols, while other applications simply interact with the filesystem.

Yanc can use a layered filesystem to provide locking and consistency for a distributed controller, logging, and other useful features.

As a proof-of-concept, NFS was layered on top of *yanc* to provide compute offload to a remote server. This is *not* a full-blown distributed controller but does show that layered filesystems can be used to the benefit of a *yanc* software defined network.

5.5.5 Decoupled from Hardware

As shown in Figure 5.5, *yanc* applications are decoupled from hardware. All *yanc* SDN applications read from and write to the filesystem. Special applications, called *drivers* also speak hardware-specific protocols. For example, an OpenFlow driver translates between the *yanc* native filesystem calls to OpenFlow switches. Additionally, a Bro driver can be used to program middleboxes.

Chapter 6

The Yanc Filesystem

6.1 Top Level Directory

The *yanc* root directory (Figure 6.1) is typically mounted on `/net`. Directories such as `switches`, `hosts`, and `views` are lists. Each subdirectory created in **e.g.,** `switches` represents a switch object.

The names of each object arbitrary. In Figure 6.1 *sw1* and *sw2* are displayed, but in practice the switch's MAC address is typically used.

6.2 Switch Directory

A *yanc* switch directory (Figure 6.2) currently more or less represents an OpenFlow switch. However the directory and its types are extensible so this directory can easily be used for "switches" in other systems such as Bro.

The switch directory contains a few fields such as `actions` and `id`. It also contains subdirectories for child objects such as `flows/`, `packet_in/`, `packet_out/`, and `ports/`.

6.3 Port Directory

The port directory (Figure 6.3) is also someone representative of OpenFlow. Of note in this directory is a symbolic link named `peer` which points to another port (on another switch) when a physical carrier is detected.

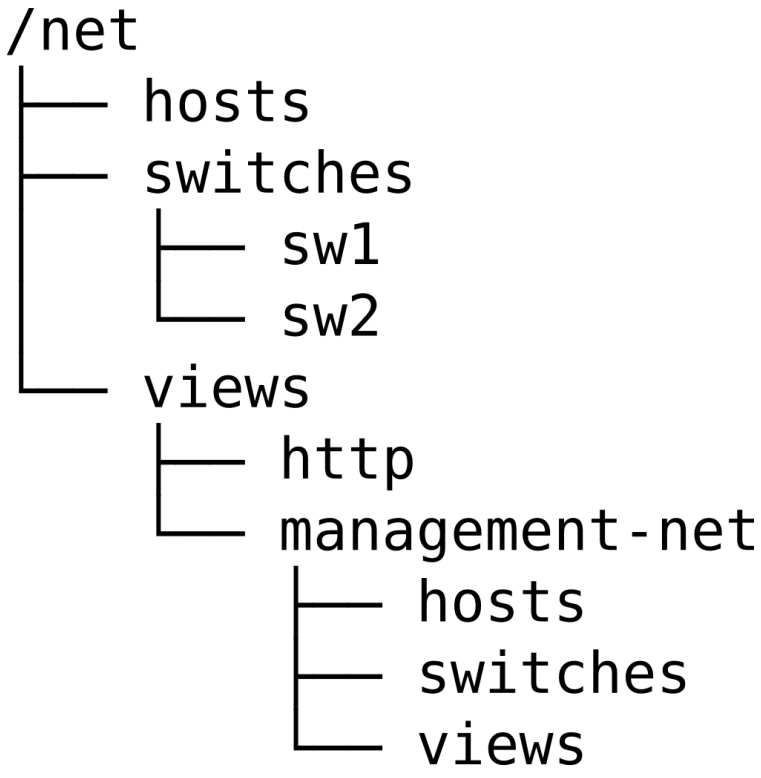


Figure 6.1: The *yanc* top-level directory is typically mounted on `/net`

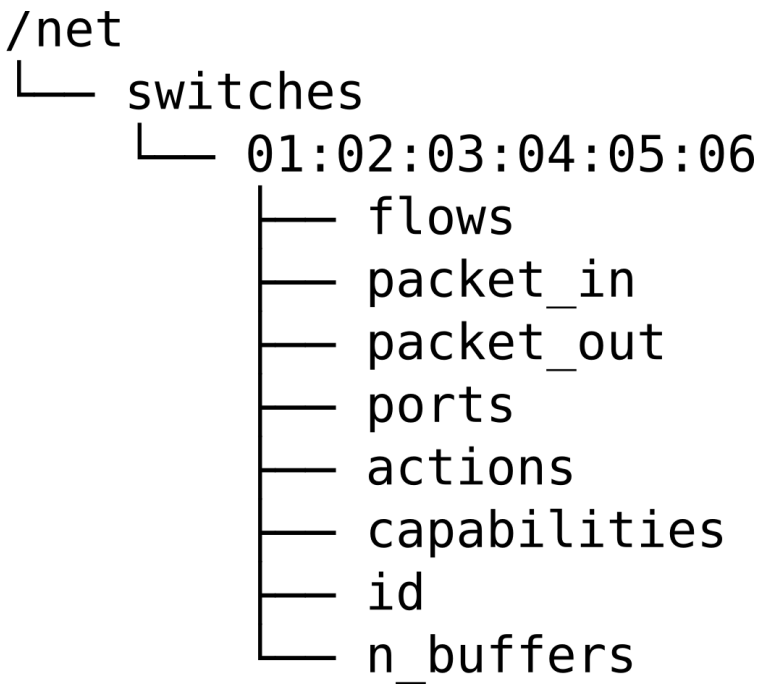


Figure 6.2: A *yanc* switch directory

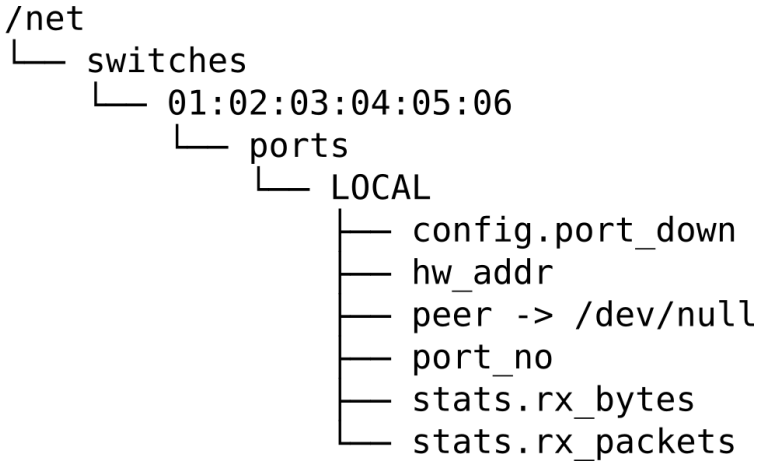


Figure 6.3: A *yanc* port directory.

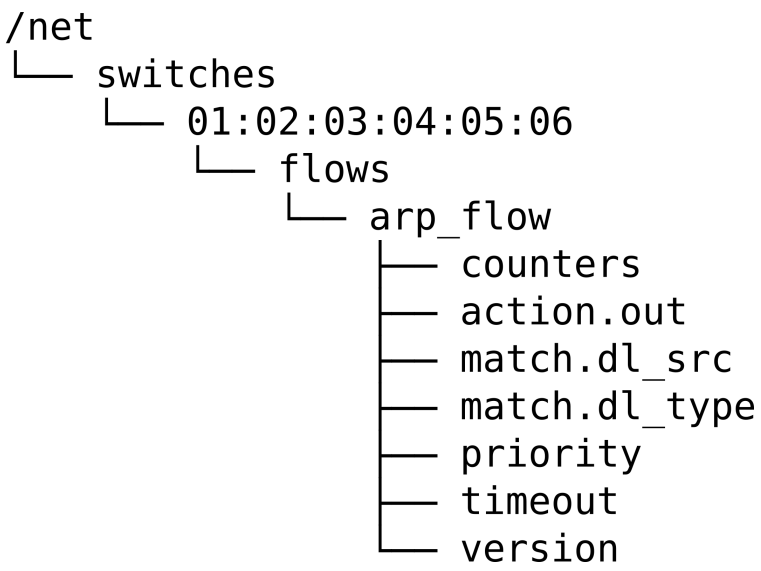


Figure 6.4: A *yanc* flow directory.

6.4 Flow Entry Directory

The flow entry directory (Figure 6.4) is used to create and modify flow entries on a specific switch. Files of the name `match.*` and `action.*` are optional. Their absence is indicative of a wildcard and presence an exact match.

6.5 Packet In and Packet Out Directories

The packet in and packet out directories (Figure 6.5 and Figure 6.6) are used for retrieving and sending packets from and to switches, respectively.

6.6 Data Types and Extensibility

In the *yanc* filesystem, each file is a particular data type and the type is indicated by POSIX extended attributes. The most used types are currently an unsigned 8-bit integer, an unsigned 16-bit integer, an unsigned 32-bit integer, an unsigned 64-bit integer, a string type, a boolean type, and a raw data type.

It is an error to write a string to a file which cannot be properly converted to the type

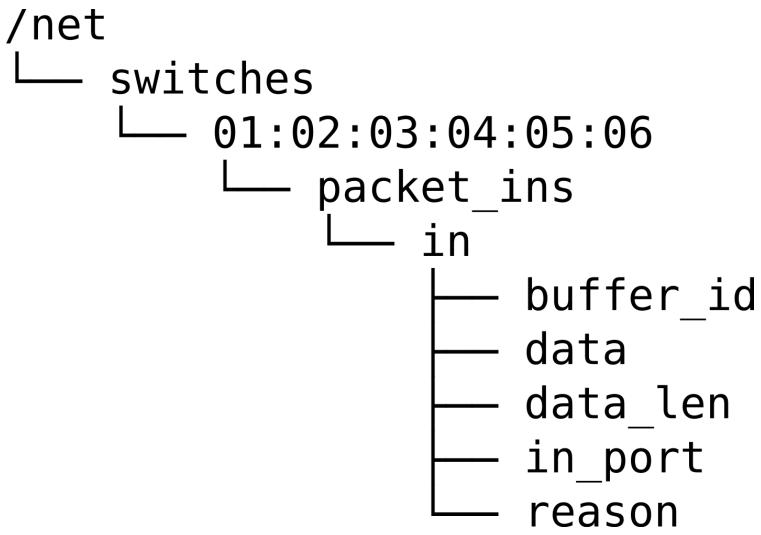


Figure 6.5: A *yanc* packet-in directory.

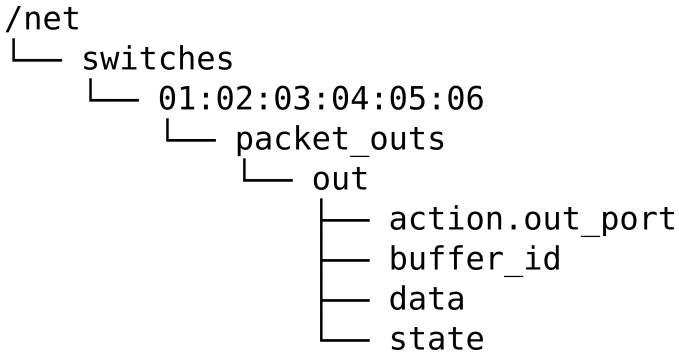


Figure 6.6: A *yanc* packet-out directory.

indicated by the extended attribute (`EINVAL` from `errno.h`).

If the type of a field is changed at runtime, the data associated with it is simply freed because there isn't a sound method for converting between arbitrary data types.

6.7 Atomicity

Atomic operations on a file system are not always straightforward. To assure atomic operations across multiple files in **e.g.,** a packet out directory (Figure 6.6), a `state` field is used. Initially, this field has the value of *staging*. When a user is done filling in the packet out fields, it is changed to *ready*. Later on, the user can check for values of *sent* or *error* to see if the packet out was successfully sent.

Chapter 7

Implementation

The *yanc* system is implemented as a number of independent components. Each one in a separate language and architecture.

7.1 The *Yanc* Core

Yanc itself is implemented in C and uses the Linux FUSE (filesystems in userspace) API. FUSE was chosen for rapid development, but *yanc* would certainly benefit as an in-kernel module from a performance standpoint.

The core contains a single event loop which is handled by FUSE. This loop calls handlers for the various filesystem calls such as `open()`, `read()`, and `write()`. This calls in turn lookup the object (directory entry) associated with a particular path and call the object-specific VFS call. For example, there is a separate `read()` handler for unsigned 8-bit integers from string types.

The high level, network centric objects such as switches automatically create all of their child fields. For example, the body of the switch `mkdir()` function contains:

```
yfs_mkdir_generic(self,  "ports",        00755, &yport_list_fops);
yfs_mkdir_generic(self,  "flows",        00755, &yflow_list_fops);
yfs_mkdir_generic(self,  "packet_in",   00755, &ypkt_list_fops);
yfs_mkdir_generic(self,  "packet_out",  00755, &ypktout_list_fops);
yfs_create(self, "datapath_id",  00644, &yfs_fops_u64, NULL);
yfs_create(self, "n_buffers",    00644, &yfs_fops_u32, NULL);
yfs_create(self, "n_tables",     00644, &yfs_fops_u8,  NULL);
yfs_create(self, "capabilities", 00644, &yfs_fops_u32, NULL);
yfs_create(self, "actions",      00644, &yfs_fops_u32, NULL);
yfs_create(self, "flags",        00644, &yfs_fops_u16, NULL);
yfs_create(self, "miss_send_len",00644, &yfs_fops_u16, NULL);
```

There is a single main object type, a directory entry, which is used heavily in the filesystem.

```
struct yfs_node {

        uid_t        uid;
        gid_t        gid;
        mode_t       mode;
        yfs_fops_t*  ops;

        union {
        struct {
                rb_tree_t     d_rb_tree;
        };
        struct {
        union {
                void*         f_data;
                uint8_t       f_u8;
                uint16_t      f_u16;
                uint32_t      f_u32;
                uint64_t      f_u64;
        };
        };
        struct {
                char*         l_target;
        };
        };

        rb_tree_t             xattrs;
};
```

The `yfs_node` structure is used for files, directories, and symbolic links. Directories store their children in a red-black tree (`d_rb_tree`) and all entries contain another red-black tree for

storing POSIX extended attributes (`xattrs`).

7.2 OpenFlow Driver

The OpenFlow driver implements much of the v1.0.0 [17] protocol. It is a self-contained application written in C++. It makes heavy use of inotify for watching flow entry, packet in, and packet out directories for changes. When a change is made, the driver constructs the appropriate OpenFlow message and sends it to the appropriate switch.

7.3 Discovery

There is an initial utility for doing discovery which is implemented in Python. This utility sends link layer discovery protocol (LLDP) messages by creating packet out directories on each switch. It then waits for corresponding packet in directories to appear containing the responses. Using these messages it can build a topology of the network and connect the `peer` symbolic links discussed in chapter 6.

7.4 Static Flow Pusher

Finally, there is a simple *static flow pusher* application implemented as a bash shell script. This script writes single flow entries to a single switch in the *yanc* filesystem. Its usage is as follows:

```
usage: flow.sh [options] <switch> <name>

  options:

    -r <root>   change path to network root, default /net
    -n          dryrun

  flow options:

    -C, --cookie   <cookie>,  default 0
    --idle-timeout <timeout>, default 0
    --timeout      <timeout>, default 0
    --priority     <num>,     default 0

    -M, --match    <key>=<val>, may be passed multiple times
                   port dl-src dl-dst dl-type
                   nw-proto nw-src nw-dst tp-src tp-dst

    -A, --action   <key>=<val>, may be passed multiple times
                   port dl-src dl-dst nw-src nw-dst tp-src tp-dst
```

Chapter 8

Applications

Writing many more applications on top of *yanc* is still largely future work. To be a viable SDN controller, *yanc* must have applications which correspond to traditional networking logic. This includes a firewall, router, and learning switch. Furthermore, more advanced applications are needed to implement some of the new SDN technologies [33, 36, 39].

8.1 Using the *Yanc* Filesystem

In general, the filesystem can be used by administrators and developers using simple shell commands, scripts, daemons, etc. For example, to bring a port down administratively:

```
# echo 1 > port_1.port_down
```

Furthermore, a flow entry can be created by a simple `mkdir` command and the required fields will be automatically created:

```
# cd switches/00:01:02:0a:0b:0c/flows
# mkdir my_flow_entry
# ls -1 my_flow_entry/
counters
priority
timeout
version
```

Administrators can also use the UNIX core utilities to **e.g.,** find flow entries which affect SSH traffic:

```
# find /net -name tp.dst -exec grep 22 {} +
```

And finally, from there developers can create scripts from the simple building blocks shown above. Below is a simple script to allow ARP broadcast traffic:

```
#!/bin/bash
flowdir=/net/switches/"$1"/flows/"$2"
mkdir "$flowdir"
echo ff:ff:ff:ff:ff:ff > "$flowdir"/match.dl_dst
echo 0x0806 > "$flowdir"/match.dl_type
echo FLOOD > "$flowdir"/action.out
```

8.2 Libraries

Expressiveness is very important in any program. Therefore, when writing an SDN application the source code should not be convoluted with many `read()` and `mkdir()` calls. Rather, network-centric functions are more appropriate.

To that end, as *yanc* develops, so are libraries for wrapping filesystem calls. For now, there is a native library as well as a Python library:

```
#ifndef _YANC_H_
#define _YANC_H_


int new_switch(uint64_t, uint8_t);
int write_flow(const char* path, match_t*, action_t*);


#endif/*_YANC_H_*/
```

```
#!/usr/bin/env python3


def new_switch(id, n_tables=1):
    pass


def write_flow(switch, matches=[], actions=[]):
    pass
```

Chapter 9

Distribution

There are three reasons for why a distributed controller would be desirable in a software defined network. The first is for improved performance and reduced latency. Placing controllers at optimal locations around the network will improve the controllers' response time to events such as table-misses.

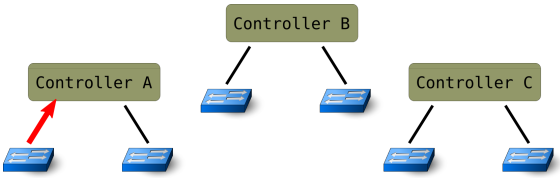
The second reason is for administrative organization. A college campus, for example, might want a controller located in each building of a campus-wide network. This setup would give the department(s) located in each building slightly more control over their intra-building traffic.

Finally, the third reason for a distributed SDN controller is fault tolerance. While controllers can be distributed for low latency, if one fails then control traffic will seamlessly transfer to another controller.

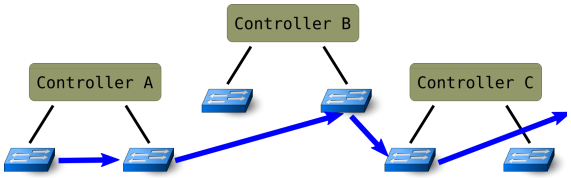
The goal of *yanc* is to rely on an external, layered filesystem for providing the appropriate semantics for distribution. However, such a filesystem does not currently exist which is appropriate for a distributed software defined network. Therefore, below is a description of one possible design.

Shown in Figure 9.1a is a distributed controller. Each switch determines its *primary* controller from the list of available controllers. Controllers can communicate directly to every switch on the network, and switches can change their primary controller at any time. However, the primary controller owns the locks for its local switches. If another controller wishes to modify a *remote* switch, it *should* acquire a lock from that switch's primary controller.

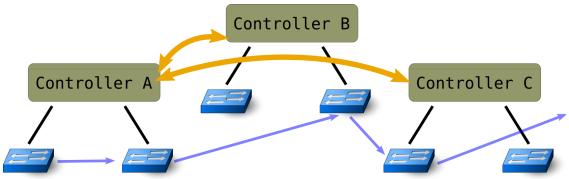
In Figure 9.1a, Controller A receives a packet in event from one of its local switches and



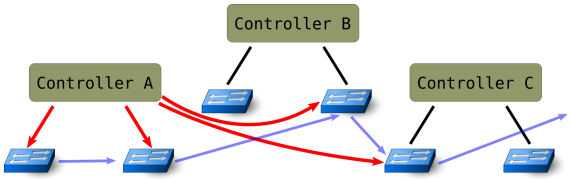
(a) A controller instances receives a packet in event.



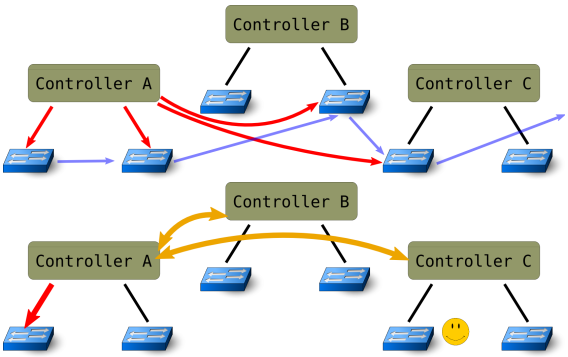
(b) The controller calculates an optimal flow.



(c) The controller acquires locks for remote switches.



(d) The controller writes changes to the network.



(e) The controller releases the packet an communicates changes to its peers.

Figure 9.1

immediately calculates an optimal path in Figure 9.1b. This path includes switches which are local to Controllers B and C, so A asks for locks for the necessary switches in Figure 9.1c. Once acquired, Figure 9.1d, Controller A writes changes across the network. Finally, in Figure 9.1e, Controller A can release the original packet and communicate its changes back to B and C.

Chapter 10

Future Work

There is much work to be done in order for *yanc* to be a production quality software defined network controller. As discussed in chapter 8, a full range of applications is required because the core itself is meant to be as generic as possible; it is not sufficient (by design) for handling even a basic network.

However, there are two main areas of the *yanc* core filesystem which require more work.

10.1 Composition

A goal of *yanc* is to enable full control planes to be constructed from logically distinct applications such as a distinct router and load balancer. Another is to allow network slices so that portions of the network can be controlled be different parties. These things require flow *composition* to allow multiple actors to write rules which can be composed constructively rather than interfering with one another. Furthermore, composition allows multiple actors to handle events such as table-misses and packet outs from other applications.

The approach with *yanc* is to allow the administrator to define composition ordering in response to various events on the network. These events include flow-mod, packet-in, packet-out, and switch-mod. Unlike Floodlight [3], which uses hard-coded dependencies on external modules within a module's source code, *yanc* allows applications to ship with *suggested* dependencies, but can ultimately be fine-tuned by an administrator.

For example, in Figure 10.1, an administrator has determined that $< event >$ should first be

handled by the slicing application, then the firewall, and finally a router.

10.2 Performance

As a filesystem, each operation is subject to a context switch. Furthermore, in the current FUSE implementation, the context switch penalty is doubled because the filesystem itself is implemented in userspace. Comparing the context switch latency of about $10\mu s$ to network latencies in the ms range, the filesystem overhead is not a bottleneck once a packet is pushed from the data plane to the control plane.

However, it is still desirable to reduce the filesystem overhead as much as possible. The first and obvious change would be to port the filesystem from FUSE directly into the Linux kernel. The disadvantage of this would be more difficult development and replacing some of the GNU libc function calls with their kernel equivalents (or even implementing equivalents).

Applications can also be designed to reduce the total number of filesystem calls. For example, the OpenFlow driver was initially designed to do an `open()` → `read()`/`write()` → `close()` on every read/write operation. This proved to be inefficient. Instead, the OpenFlow driver now holds all files open for the duration of its runtime. It then caches the value stored in a particular file and only performs a `read()` when an inotify event has determined a writable file was closed and marked the cache dirty. All `write()` calls still happen immediately, but are not accompanied by `open()` and `close()`.

Applications can also employ a certain degree of parallelism when making many changes to the filesystem. For example, asynchronous IO can be used to fill in the fields of an object without waiting for each write operation to complete in serial. Additionally, objects can be created and filled out by multiple threads as the *yanc* core is fully thread-safe.

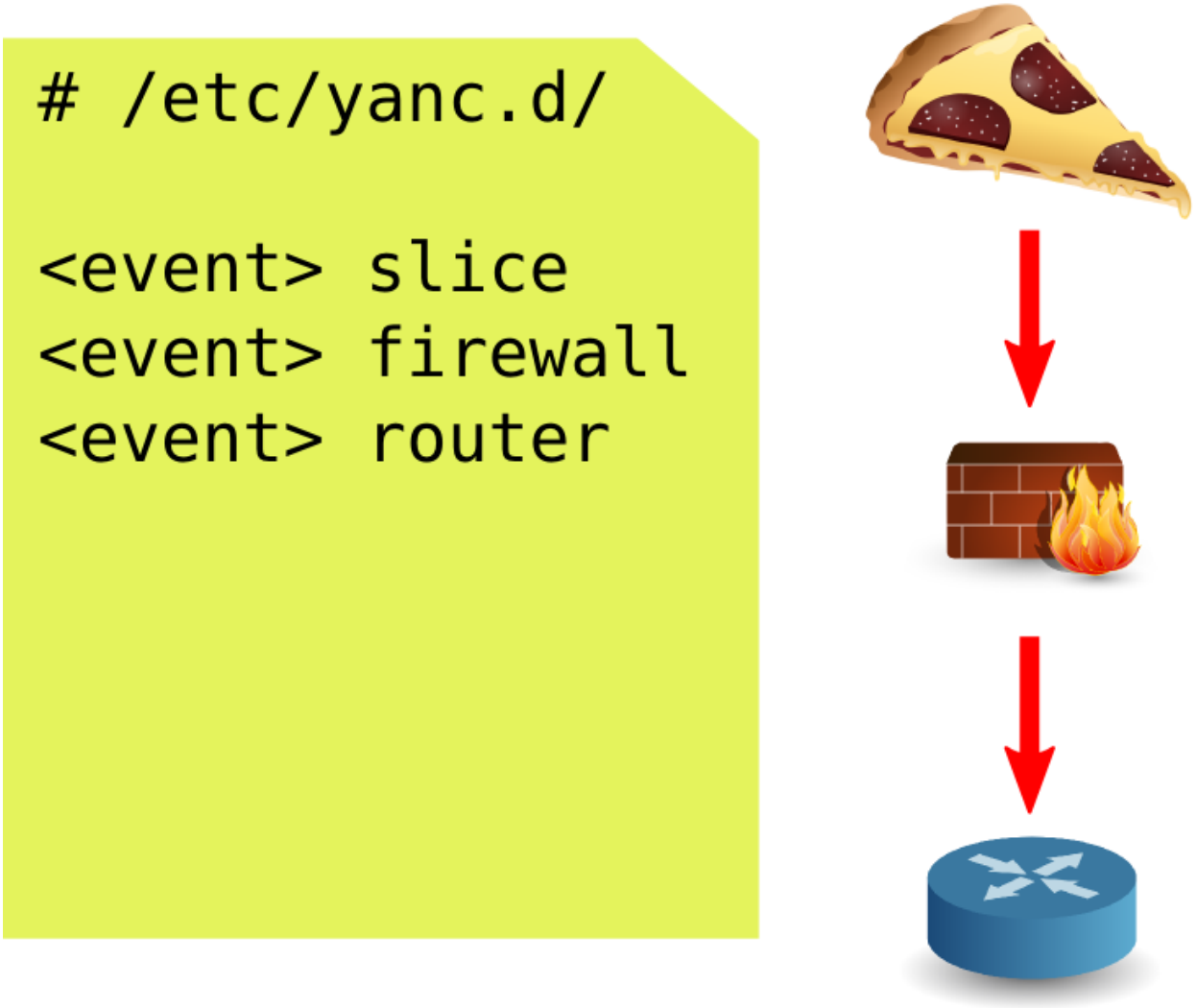


Figure 10.1: SDN composition is defined by the network administrator(s).

Chapter 11

Conclusion

This paper has presented *yanc*, a vision of how operating system mechanisms and principles can be applied in the context of software defined networking. Effectively, *yanc* realizes a network operating system which can be used in a variety of ways in order to leverage innovation in the operating system space. Thus, more focus can be put in specific control-plane-centric topics such as load balancing, congestion control, and security. The current *yanc* implementation is a step towards realizing the goal of tightly integrating the networking operating system into a traditional operating system.

Bibliography

[1] Big switch networks. http://www.bigswitch.com.

[2] Cumulus networks. http://cumulusnetworks.com.

[3] Floodlight openflow controller. http://www.projectfloodlight.org.

[4] Fuse: Filesystem in userspace. http://fuse.sourceforge.net.

[5] Onos: Open network operating system. http://es.slideshare.net/umeshkrishnaswamy/open-network-operating-system.

[6] Opendaylight: A linux foundation collaborative project. http://www.opendaylight.org.

[7] Pica8: Open networks for software-defined networking. http://pica8.com.

[8] Ryu sdn framework. http://osrg.github.io/ryu.

[9] Network functions virtualisation. In SDN and OpenFlow World Congress, Oct. 2012.

[10] Zheng Cai. Maestro: Achieving Scalability and Coordination in Centralized Network Control Plane. PhD thesis, Rice University, 2011.

[11] Martin Casado, Michael J Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking control of the enterprise. In ACM SIGCOMM Computer Communication Review, volume 37, pages 1–12. ACM, 2007.

[12] Martin Casado, Tal Garfinkel, Aditya Akella, Michael J Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. Sane: A protection architecture for enterprise networks. In USENIX Security Symposium, 2006.

[13] Martin Casado, Teemu Koponen, Rajiv Ramanathan, and Scott Shenker. Virtualizing the network forwarding plane. In PRESTO, 2010.

[14] Linus Torvalds et al. The linux operating system. http://www.kernel.org.

[15] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: a network programming language. In Proc. ACM SIGPLAN international conference on Functional programming (ICFP), 2011.

[16] Open Networking Foundation. Openflow specification. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow.

[17] Open Networking Foundation. Openflow specification. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow.

[18] Open Networking Foundation. Openflow specification. https://www.opennetworking.org/sdn-resources/onf-specifications/openflow.

[19] Aaron Gember, Prathmesh Prabhu, Zainab Ghadiyali, and Aditya Akella. Toward software-defined middlebox networking. In Proc. Workshop on Hot Topics in Networks (HotNets), 2012.

[20] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: a scalable and flexible data center network. In ACM SIGCOMM Computer Communication Review, volume 39, pages 51–62. ACM, 2009.

[21] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management. SIGCOMM Comput. Commun. Rev., 35(5):41–54, October 2005.

[22] Adam Greenhalgh, Felipe Huici, Mickael Hoerdt, Panagiotis Papadimitriou, Mark Handley, and Laurent Mathy. Flow processing and the rise of commodity network hardware. ACM SIGCOMM Computer Communication Review, 39(2):20–26, 2009.

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

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

[25] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. NSDI, Apr, 2012.

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

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

[28] Yohei Kuga, Takeshi Matsuya, Hiroaki Hazeyama, Kenjiro Cho, and Osamu Nakamura. EtherPIPE: an Ethernet character device for network scripting. In ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN), 2013.

[29] Joao Martins, Mohamed Ahmed, Costin Raiciu, and Felipe Huici. Enabling Fast, Dynamic Network Processing with ClickOS. In ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN), 2013.

[30] J Mccauley. Pox: A python-based openflow controller.

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

[32] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software defined networks. In Proc. Usenix Network System Design and Implementation (NSDI), Apr 2013.

[33] Philip Porras, Seungwon Shin, Vinod Yegneswaran, Martin Fong, Mabry Tyson, and Guofei Gu. A security enforcement kernel for openflow networks. In Proceedings of the first workshop on Hot topics in software defined networks, pages 121–126. ACM, 2012.

[34] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. Split/merge: system support for elastic execution in virtual middleboxes. In Proc. USENIX conference on Networked Systems Design and Implementation (NSDI), 2013.

[35] Dennis M Ritchie and Ken Thompson. The unix time-sharing system. Communications of the ACM, 17(7):365–375, 1974.

[36] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the production network be the testbed? In Proc. USENIX conference on Operating systems design and implementation (OSDI), 2010.

[37] David L Tennenhouse and David J Wetherall. Toward an active network architecture. In Electronic Imaging: Science & Technology, pages 2–16. International Society for Optics and Photonics, 1996.

[38] Andreas Voellmy and Paul Hudak. Nettle: Taking the sting out of programming network routers. In PADL, pages 235–249, 2011.

[39] Andreas Wundsam, Dan Levin, Srini Seetharaman, and Anja Feldmann. Ofrewind: enabling record and replay troubleshooting for networks. In USENIX ATC, 2011.