UNIVERSITY OF CALIFORNIA

SANTA CRUZ

Analysis of Applicability and Usability of Programmable Networks in Modern Networks

A thesis submitted in partial satisfaction
of the requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Lincoln Thurlow

December 2016

The Thesis of Lincoln Thurlow
is approved:

Professor Katia Obraczka, Chair

Professor Bradley Smith

Professor J. J. Garcia-Luna-Aceves

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

Analysis of Applicability and Usability of

Programmable Networks in Modern Networks

Lincoln Thurlow

Software-Defined Networking and Network Function Virtualization lack a unified solution for general network programmability. OpenFlow, Software-Defined Networking's de-facto standard for network programmability, is based on rule-to-packet header matching, in which an action is executed when a rule involving the packet's header field is satisfied. However, this model of matching is extremely limited in terms of its expressibility, scalability, and distributability. This is especially evident when satisfying the requirements of a more diverse set of applications. To fully utilize the network as a programmable platform, an architecture that supports an unconstrained, robust computational model is required. A programming platform that can support these attributes must therefore live in software. To make the network fully programable, it is necessary to look at previous Active Networks research as well as current Software Defined Networking research.

We propose that by utilizing the latest virtualization techniques, a new programmable networking architecture is possible and feasible for today's networks. Such a programmable network architecture would be capable of handling the unique problems of executing code in the network. We call the architecture we have developed to meet these criteria, SCAN.

1 Introduction

Driven by increasingly more complex and resource-demanding network services and applications, there has been a strong push, both from academia and industry, towards "softwarization" and virtualization of the network. Software-Defined Networking (SDN) implementations have validated that control mechanisms implemented in software are capable of performing line-rate forwarding. The evolution of software displacing previously hardware specific functions for increased programmability and functionality encourages exploration into creating a complete and fully programmable network. Previous research in SDN, active networks, and virtualization provide the foundation for a fully programmable network architecture to solve the inherent issues faced by today's networks.

The recent popularity of OpenFlow[53], a Software-Defined Networking implementation enables network administrators a new found ability to program and manage the network through a logically centralized controller. The controller is OpenFlow's mechanism for separating the control plane from the data plane, the controller communicates with enabled switches through the OpenFlow protocol. Switches are programmed with their forwarding logic based on the rules that are inserted into their forwarding table. The flow table consists of rules, actions, and statistics. Rules dictate the logic of network, while actions dictate how the rules will be interpreted (drop, forward, etc). Statistics allow for rules to be made that account for network conditions given past and present events. Rules provide the core functionality of OpenFlow's programmability. The rule-to-packet header matching limits how well the centralized table which stores the rules can scale[36] for physically distributed for larger environments[44][82]. There currently has not been a proposal for logically decentralizing the control plane. One reason for this may be because OpenFlow is tied too closely to its centralized model. Additionally, OpenFlow rule matching lacks a programmer friendly and intuitive interface. Programming languages such as Frenetic[28] and Nettle[84] attempt to fix this problem by providing a simple declarative programmable interface to programmers. Theses languages sit on top

of OpenFlow to abstract away the error-prone nuances of programming rules in the network.

The original architecture for active networks called for the decoupling of network services from the underlying hardware. Decoupling services and hardware allowed for the idea that arbitrary code could be run in the network to service the programmer. Services are then packets which contained executable code, these packets are referred to as *capsules*[81], and the payload containing the executable code as *active code*[20][61]. Capsules can optionally be executed at each hop utilizing the resources of the local node for both processing and storage. Additionally they have access to network primitives, primitives are variables which represent the basic functionality. This functionality is most simply the ability to modify the routing route tables, policy mechanisms, protocols, and other basic network functionality. The power of programming the network with active networks lies in the ability of code to utilize both the exposed primitives, and temporary storage of the device. Allowing code to use storage opens up the possibility for more dynamic caching mechanisms as well as the change in perspective of networking fundamental from an end-to-end to a node-to-node model.

Previous research in the area of active networks focused on utilizing code to replace the existing framework by providing secure[37][86] and efficient[74] languages. Given the hurdles of proving a secure architecture, the absence of a "killer app" to motivate the substantial complexity of active networks eventually lead to the downfall of the architecture [60][87][26]. More recent attempts of building active network architectures have focused more on performance and less on the applications of programmability [41][40][11]. The P4 architecture is of special interest as it uses many of the active networking concepts, but strictly limits open unrestricted code in place of a rule-based compiler that is very similar to the OpenFlow model. P4 is called by its authors as OpenFlow 2.0.

Network Function Virtualization (NFV) is a more recent architecture built around virtualizing functionality intended for dedicated hardware into software for generic hardware. NFV enables network resources to be virtualized and dynamically allo-

cated into the network, generating a more robust network by allowing idle resources to be repurposed for greater efficiency. The growth of NFV can largely be explained by the increasing number of middlebox and software applications developing for the network. Applications such as load balancers, firewalls, and caches are all critical to the operation of any sized network. The architecture provides the functionality of a programmable platform which can spawn network specific tasks in software throughout the network based on triggering threshold values (throughput, delay, etc). Generally these middleboxes would require dedicated hardware but with the support of hardware virtualization[59][23][39], software can now execute the same tasks with minimal penalties, making NFV an extremely adaptive architecture for dynamic networks. The abundance of virtualization techniques within the networking field has lead to the growth of what we now term "The Cloud". Without the ability to break network components down to a minimum set of primitive operations which can be utilized though programmatic interfaces, sites such as Amazon Web Services (AWS) and Microsoft's Azure would not be possible.

The push for more software in the network has been caused by the pull from more powerful applications. The overall ossification of the network protocol stack has forced functionality that should live in the network stack to be developed on top. By moving these protocols to the top of the stack, many inefficiencies are exposed to programmers and end-users. What is needed for developing a more adaptive stack is an architecture which allows programmability to live in the stack. The current network needs a unifying architecture encompassing SDN, NFV, and active networks architectures by providing the shared goal of programmability, but extending the control mechanism to a decentralized environment, and providing the agility of NFV in these dynamic environments.

There already exists a hierarchy to describe the different levels of programmability in the network. At the highest level is programmable networking. The programmable network architecture describes the decoupling the hardware from software, providing an open interface, virtualizing the network, and combining coexisting architecture and frameworks into a single architecture[16]. It is essentially an umbrella term

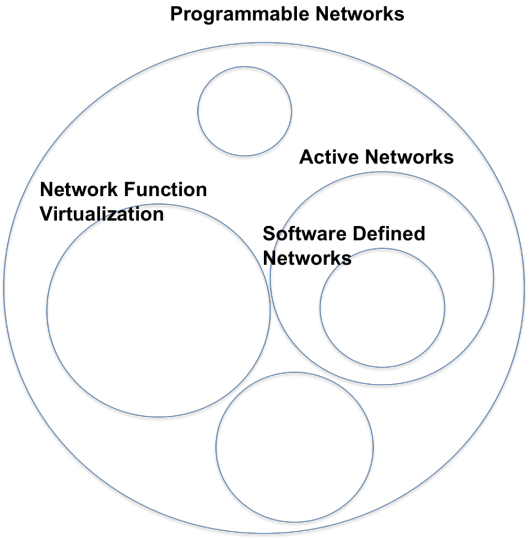


Figure 1: The Programmable Network Research Canopy

to describe all aspects of interfacing and programming networks together. Directly under the canopy of programmable networks are active networks which describe the decoupling of services from hardware. Below active networks are SDN and NFV networks. SDN define the separation of control and data forwarding, which can be seen as a subset of what can also be done with active networks. NFV networks is the decoupling of services from hardware using virtualization. We show our own view of the research canopy in Figure 1. Empty bubbles are illustrative of other topics not included herein which also exist under the programmable networks research canopy. We discuss an approach utilizing our own programmable network prototype, SCAN, for using code execution within the network to empower programmability for decentralized and dynamic networks. We start by defining the foundations and discuss the nature of the architecture. Next we detail our preliminary implementation of SCAN and we discuss at a high level the problems of security and performance for any programmable network architecture.

2 Architecture

To develop a fully programmable networking architecture in a logically decentralized manner we need to base our design on a principle which itself is decentralized. For

this reason we base SCAN on an architecture which allows for mobile and executable code in the network. Code execution in the network allows for dynamic protocols and functionality to be prototyped and implemented very quickly. The code carried in packets can be thought of as the basic logic required to implement a service. The logic for the service is decoupled from hardware and by extension, nodes in the network; allowing nodes to modify services and logic on a node-by-node basis. The drawback to this design is that code execution in the network is slow, inefficient, and opens a can of worms for security. For this reason any architecture which incorporates dynamic elements in the architecture must also present corresponding static elements where programmability is restricted, but provide a fast and efficient mechanism for forwarding in the network. The programmable network architecture we present is designed with the essence of an operating system for the network.

The programmable network architecture relies heavily on virtualization. Virtualization provides a simple abstraction layer to build a generic architecture that acts as the interface to the network operating system. There are additional benefits to virtualization; hardware virtualization allows software access to low-latency and high-throughput functionality while software virtualization provides isolation. Both types of virtualization support departing from fast and expensive dedicated hardware to cheaper and slower commodity hardware. Building the programmable network architecture on top of virtualization enables a fully programmable network to achieve speeds previously not possible with active networks architectures. It is important to note that this functionality comes at the cost of additional complexity. The demise of active networks was due its high level of complexity for little to no benefit over the default IP architecture. This implies that a programmable network architecture must be complex when needed, and simple by default. The programmable network architecture is complex to support a wide breadth of functionality and reprogrammability, but this enables a more diverse set of network functionality. Our programmable network architecture also differs from active networks fundamentally by using layered software virtualization throughout the architecture for both isolation and providing a general interface across modules.

2.1 Motiviation

The biggest problem as mentioned above with a programmable network model such as active networks was the lack of a "killer-app" to motivate the implementation of large scale programmable network. The blunt truth of the matter is that there is no "killer-app" to motivate programmable networks. The breakdown for the failures to motivate a programmable network come from:

1. A software solution in the network will only work when efficiency is no longer a monetary issue. This issue is very unlikely to be be resolved any time soon as processor manufactures continue to run into physical barriers (heat, area, etc). This will likely cause the end to multicore scaling[25] as we traditionally know it until a new architecture (q-bits for example) can become feasible.

2. The need for versatilely greatly outweighs the need for structure (prototype is more important than performance)

So, putting aside the above two problems, lets assume that neither of these issues constrain the development or integration of programmable networks. What can a programmable network be used for, and how will it outperform a more traditional approach? A programmable network can allow networks to adapt to conditions by the code they carry. The network nodes do not need to be aware of the logic of the code, nor does it need to store, or in any means be responsible for the content or information contained by the code. In a simple case this could allow users to create their own multicast groups, cache networks, P2P and sharing networks. This can be accomplished by writing code that essentially acts as an application that sits and waits on intermediate devices for requests. In a more complex programmable network, it would allow for dynamic routing to utilize machine learning algorithms at these intermediate nodes based on network probes to redirect information flows faster than the traditional end-to-end algorithms (such as TCP) from discovering and reacting to network congestion. This type of approach has a smaller feedback loop as it works on a node-by-node basis.

From the perspective of a novice network user, the path for programmable networks is most relevant in the age of government and corporate ownership of the Internet. A programmable network with end-user control over network flows allows the user to mitigate troublesome networks. It may be used as a means to avoiding throttling from a corporation to disrupt traffic to a competitor's site. Or it may be used to route around countries known for invasive network spying and espionage. User-based-routing is foreign as it remove the key tenant of engineering, efficiency, from the equation and replaces it with inefficient human made policies. We will discuss more later the issues of security as it relates to distributed denial of service (DDOS), man-in-the-middle (MITM), forgery, and other attacks that can be launched on a programmable network.

A topic closer to reality and more realistic, involves the need to prototype network conditions without having to modify network equipment and software with every feature update. This is something very doable with today's technology but becomes harder in the heterogenous network made up of different vendors with different methods of updating. Data centers today contain networking equipment across multiple vendors and generations[78]. A programmable network can make use of a single code base that can be pushed and pulled across the network in a revision controlled manner such as those used by git, subversion, and mercurial. This is agnostic to the underlying hardware (with exception to firmware), providing a singular interface to the hardware. The architecture allows for easy diagnostics and debugging of network code in a testbed environment without the fuss of having to first run the code through a network simulator or across a testbed environment.

2.2 SCAN

In this section we discuss how to build our programmable network architecture SCAN. Our network programmable architecture enables general purpose programming of the network using an exposed API which interfaces between the programming language sandbox environment and the network programmable operating system, and the primitives exposed therein. This approach is analogous to previous

active networking research based on code execution in the network. Our architecture modifies the previous framework provided by active networks as the groundwork for a modern network programming platform. First, the network programmable architecture is designed to enable all programming languages. Second, the design relies heavily on virtualization to support the infrastructure and system for general languages. Lastly, the main purpose is to create an architecture with a distributed run-time environment which can natively support distributed computation and algorithms in the network.

Our architecture relies heavily on virtualization, from which we are able to form a modular infrastructure that can support any environment. The architecture supports a move away from special purpose built hardware towards general commodity hardware. Using additional commodity appliances means that the architecture can be utilized in clusters to potentially hit line-rate for forwarding and can adapt to future architectures as the control mechanism for the system is left in a programmable software layer. It is important to note that this functionality comes at the cost of additional complexity, however as the environment is completely modular, all functionality can be stripped and streamlined to present production networks a minimal surface for errors.

By moving computation to the network, algorithms which previously used end-to-end models can be integrated into the network. An example of such functionality is a distributed firewall. We however prefer to think of the architecture as providing a distributed runtime environment for distributed algorithms that support network context awareness. A more appropriate application is one for an intelligent transport system where each node in the graph is a vehicle navigating traffic. Each node must apply computation towards a distributed goal of driving in traffic. This provides for applications where due to the centralized control plane of OpenFlow, there is a limited ability to use or delegate controllers[73].

2.2.1 Programmable Network Operating System

The Programmable Network Operating System (PNOS) is the linchpin of the programmable network architecture we propose. The PNOS can optionally be installed on every node in the network, therefore it is essential that the PNOS is capable of forwarding packets through the critical-path at line rates. To allow for low overhead for forwarding and programmability we require a modular operating system. This design enables trading off performance, security, and flexibility depending on the requirements of the network. Modules are meant to be replaceable, even the core modules for governing the system are modifiable to allow for additional programmability. The main modules for the PNOS design are the dispatcher, run-time environments, and the resource advisor. Each module is responsible for different functionality to permit a programmable network architecture. For example the dispatcher module is used for parsing packet headers to correctly forward packets or code to the correct stack or module. The run-time environment implements the functionality of the system in either dynamic or static components. Lastly, the resource advisor is responsible for managing shared system resources such as CPU, memory, and storage. Additional modules can also be added as either a shim between modules to increase functionality or as a complementary module.

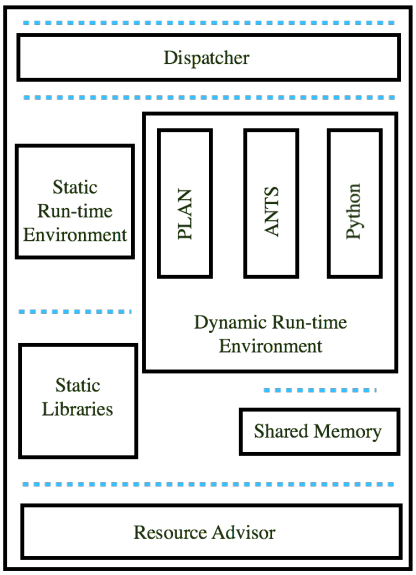


Figure 2: The PNOS design

2.2.2 Dispatcher Module

The dispatcher component of the PNOS is the first module to interact with an incoming packet. The dispatcher must quickly sort packets based on whether the packet requires a dynamic run-time environment such as a language sandbox or a static environment such as the IP stack. The design of the dispatcher is based on a packet parsing module. The parsing model can be integrated with either hardware supported parsers [12] or done in software with languages like P4[11]. The dispatcher's modularity and programmability are designed to fit within the scope of the P4 language and the widening movement towards programmable parsers for prototyping and implementing new protocols. A bootstrapping module can be placed in the dispatcher to allow new protocols to write to the dispatcher allowing new packets to be parsed on-demand, then communicate to the resource advisor to install a temporary run-time environment. This mechanism would rely heavily on permissions, trust, and lightweight run-time environments, however in prototyping networks, it may be more crucial to implement functionality over practicality. For environments which rely on performance, the dispatcher can be programmed for static protocols to achieve high throughput and low latency forwarding. This type of environment would be more of a P4 style node where the P4 code is compiled and loaded onto the node ahead of time. The P4 language requires programmers to generate a match-action pairing (same as OpenFlow) as shown in Figure 3 based on the headers of the incoming packets on what actions should be taken on reception. In PNOS a rule set would include pointers for interfacing packets between the P4 module and the run-time environments.

2.2.3 Run-time Environments

In section 2.2.3.1 we discuss how SCAN would handle dynamic environments such as handling executable code in the network. In section 2.2.3.2 we describe the more performance based static environment in SCAN.

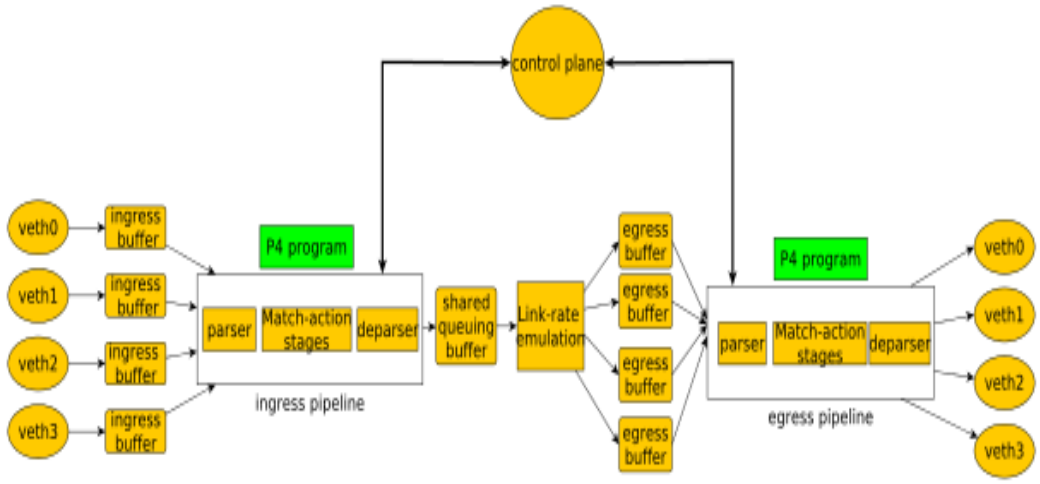


Figure 3: The P4 language processing diagram

2.2.3.1 Dynamic Environments

The dynamic environment of the operating system consists of language sandboxes which provide an interface between the code and the network functionality exposed by the resource advisor. The configurable dynamic environment allows knobs for adjusting the number as well as the type of sandboxes available. These knobs could for instance be used to disallow C or C++ code from being run in the network due to security concerns. Or if network administrators only wish to run a single language across the domain for simplicity and management reasons.

Code in the sandbox is also limited to a similar set of constraints. Depending on the openness of the network this code can load third-party programming packages through secure channels such as in shared memory or a network directory service (such as Chord[80], Tapestry[92], or Pastry[71]) interfaced through the resource advisor. If allowed, code can make foreign requests to a directory service for specific functions in C++'s boost library or a non-standard Python library. The purpose of this functionality is reduce overall code size while supporting functionality without direct approval. Each node only has to maintain the packages which it requires, this approach creates an easy framework for prototyping. The verification could be done as it is now through signing of code and verification of the signature.

In the case of security, the sandbox also protects the system from the code. De-

pending on the language and the native support for virtualization, the sandbox may be run within another virtual machine to further isolate the execution of code from the system. The language sandboxes are designed to be light-weight virtual machines with optional virtual machine monitors (VMM) to provide better isolation when required. Systems like Docker[54] can be used as a means to load similar sandboxes across the system with the same settings. Depending on the requirements of the system, internal or external VMM [70] can be used to further mitigate virtual machine escapes. For example, code using Java will use the Java Virtual Machine (JVM), depending on environment's requirement on security the JVM may need to be inserted inside another virtual machine to stop code capable of breaking the JVM from also hijacking the resource advisor and the system as a whole.

The dynamic environment enables the capabilities of a NFV system by creating a means through which the operating system can expand and reduce the expressiveness or computation of the system. Many modern NFV systems make use of Kernel Virtual Machine (KVM)[42], Xen[8], or VMWare's ESX. The PNOS resource manager is very closely modeled after these types of hypervisors for managing resources across multiple virtual machines and resources.

2.2.3.2 Static Environments

Due to the integral software used in the critical elements for forwarding and routing in network devices, some amount of programmability must be replaced with high performance functionality. Static environments restrict foreign code execution and are most commonly implemented using a REST API interfaces. The static environment interface can still allow sets of programmability in much the same way as IEEE P1520[10]. By allowing the PNOS to accommodate both dynamic and static environments, the network can move from the dynamic and slow prototyped network to a more stable and quick production network by migrating more functionality from the dynamic to the static environments as shown in Figure 4.

Static Libraries can be loaded by their static environments to implement the core functionality of the previously dynamic environment. An out-of-band method of au-

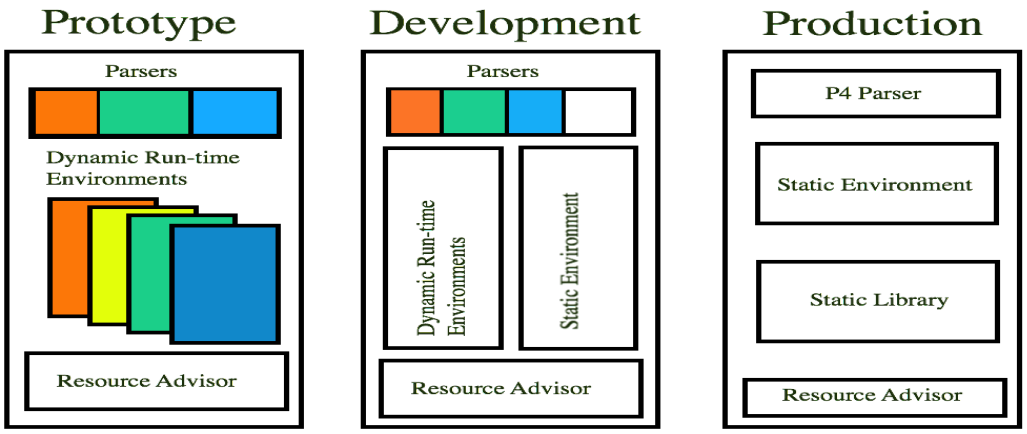


Figure 4: The benefit of dual environments: a network architecture that changes from prototype to production.

thentication securely run from startup (such as AEGIS[5]) can be used to provide a secure mechanism for allowing trusted code libraries to be loaded. This mechanism allows code to be added without penalizing performance at the cost of programmability. However, like many designers of prior active network architectures, this can lead to a lag between the need for functionality and utilizing that functionality.

The static environment's purpose in the operating system is to provide a high throughput interface. In many networks, the static environment will be the only environment being used by PNOS for performance benefits as also mentioned back in section 2.1.

2.2.4 Resource Advisor Module

The resource advisor acts as the hypervisor for system resources. The resource advisor is responsible for system integrity, making sure that virtual machines and sandboxes are created and destroyed, managing storage, and inter-module communication. The resource advisor is made up of 3 sub-components: the scheduler, monitor, and package manager. Each sub-component is modular. This enables mixing and matching between sub-components to optimize the overall usability of the system. The resource advisor is the kernel of the PNOS architecture.

The scheduler is the apparatus delegated to enforcing performance policies. The

monitor sub component manages security and accesses to shared memory and handles permissions and resource management for each virtual machine. The package manager handles the management of modules and components installed on the node as well as shared libraries on the system.

As the most critical region of importance in the PNOS architecture, the resource advisor is intentionally left ambitious and vague. The focus at a high level on this work is on the dynamic and static environment duality. The resource advisor is as stated above, the wizard behind the curtain. The resource advisor is left out largely due to the intricacies of implementations, many papers in the field deal with the fine details of their network operating systems [65][38][24][57][9], here however the focus is on what the high level design of the system would look like and require.

3 Implementation

We have implemented a lightweight programmable network implementation in Python to mimic the behavior of a programmable network architecture without implementing the complete PNOS necessary for a production implementation. Our implementation focuses on utilizing additional functionality over performance and security and does not utilize either software or hardware virtualization.

For our test we used a single desktop computer with 16 GB of Memory and an Intel i7 with 4 cores for using Kernel Virtual Machine (KVM) as the hypervisor. We installed 4 KVM hosts, each running Ubuntu 14.04 with 1 VCPU, 4 GB of memory, and 12 GB of storage. A server on every node is set to listen on port 50000 for executable code and bootstrapping. The server works like Quagga in that it accesses and modifies linux kernel network variables on a node. When the server receives code, it calls the code's native compiler and spins off a new process for execution. In our implementations below we developed our prototypes using the Python language. The native code being executed however can be any language with a compiler installed on the current device.

Even though this testbed is a single node with four virtual hosts, it is still possible to illustrate the use case for programmable networks over an OpenFlow enabled

network. Through the hypervisor and each KVM host we modified link capacity, loss, delay, and jitter using *tc qdisc* to demonstrate the robustness and dexterity of code to adapt to the network conditions.

3.1 Distributed Bellman-Ford

We started by implementing a decentralized routing algorithm using shortest prefix matching for IPv4. Our implementation loosely follows Routing Internet Protocol (RIP)[50]. The Bellman-Ford algorithm is a decentralized routing algorithm where no nodes in the network keep state information about other nodes in the network (this is contrast to link-state routing and Dijkstra's routing algorithm). Nodes running Bellman-Ford will send updates to their neighbors. The neighboring nodes will process updates and for each node in the network will use the received updates to determine which neighbor the node should forward messages to in order to get to the destination. A node in a Bellman-Ford algorithm is likely to have a table with each destination, associated with each destination is a neighbor node with which messages should be forwarded. RIP is the implementation of Bellman-Ford with added features to prevent certain situations from occurring such as the counting-to-infinity problem. The counting-to-infinity problem is caused when a node receives an update. Unknown to the current node that another node in the network has been disconnected. The current node uses its value for the disconnected node and sends it back to its neighbors. When a neighbor of the disconnected node receives this update, it then broadcasts to its own neighbors that the cost to the disconnected node has incremented by the cost going through the other nodes. This will then loop as everyone's cost in the path increments, eventually reaching "infinity."

3.1.1 Implementation

Our Bellman-Ford implementation begins by having one node send out bootstrapping code, which contains the startup code for Bellman-Ford. The bootstrapping sends the code to port 50000, which spins off the new process with the process initialization code. The Bellman-Ford code spins off a new thread to listen on port

60000 for updates, but interfaces with the original server on port 50000 for system modification to the routing table. Two threads are created for handling communications between nodes. The *sender* and *receiver* threads responsible for sending and receiving updates to and from neighboring nodes. The *receiver* threads check interfaces and networks to attach to based on parameters passed by the initial bootstrap as well as those exposed by the system to start listening. The *receiver* thread then creates a single queue for all interfaces to process messages according to a FIFO ordering. The bootstrapping mechanism that manages the *receiver* thread can be given parameters to initialize only subsets of the network, interfaces, or VLANs to extend the flexibility and programmability of the network. Once the *receiver* thread has been initialized, the *sender* thread is created to generate update messages every 30 seconds. We adapted the algorithm to implement sequence numbers for freshness and implemented poison-reverse on the *receiver* thread. Once we started the bootstrapping processes it took only a few seconds for the code to be sent and run on every connected node in the network, and begin their separate Bellman-Ford processes. With our small network, we did multiple tests for correctness of our algorithm by forcing counting-to-infinity scenarios without success of observing such an event.

Distributed Bellman-Ford is an ideal implementation for programmable networks because of the decentralized nature of the algorithm. It is possible to implement Bellman-Ford using OpenFlow, but at the cost of having every message needing to be sent to and from the controller. Sending every message to the controller is necessary due to the possibility of state change updates between nodes and having the logic required for parsing these events stuck in the Controller. If switches had the ability to apply computation in an OpenFlow environment they may be able to verify that certain updates do not change state and therefore are not required to be forwarded to the controller.

3.2 TCP Snoop

TCP Snoop was developed by the Daedalus group at Berkeley for improving the throughput of TCP connections over wireless links[7]. Our choice to implement Snoop was based around the idea of an adaptive protocol that would sit in the network to provide greater reliability and performance guarantees. As we discuss below, we used Snoop to automatically install on nodes with high loss to provide a hop-by-hop guarantee, utilizing local caches on each node to store TCP packets. OpenFlow is unable to replicate such a protocol without use of a separate cache due to the sheer number of links connected to the centralized controller which may require packet caches. With NFV implementations, this would also require additional setup by spinning up the node on the hypervisor and creating virtual links over the network, adding what could be far greater latency and delay depending on the distance. By executing Snoop on the node itself we don't require a virtual overlay topology which may do more harm than good.

Snoop was designed with mobile and wireless networks in mind, it is made up of 3 entities. The Fixed Host (FH) is a wired host and as the name describes, is not a mobile host. The Mobile Host (MH) on the other hand is mobile and wireless, making it more prone to higher loss and greater delay. In between the FH and the MH is the Base Station (BS). Snoop is run on the BS to provide a seemingly reliable connection between the FH and the MH. The BS is responsible for tracking the TCP data and acknowledgements packets that are forwarded through it, keeping a single logical TCP connection between the FH and the MH rather than splitting the TCP connection into two separate connections as done with Indirect TCP[6]. Snoop provides better throughput by caching TCP data packets and resending lost data much quicker than the FH timer would timeout. This is accomplished by setting state on the BS to track the round-trip time (RTT). The BS's RTT is strictly less than the FH's to the MH as the BS is between the FH and the MH and will therefore trigger first in the case of loss in the network.

The Snoop protocol is broken down into two algorithms, *snoop_data* and *snoop_ack*. The *snoop_data* algorithm is responsible for handling the data communication be-

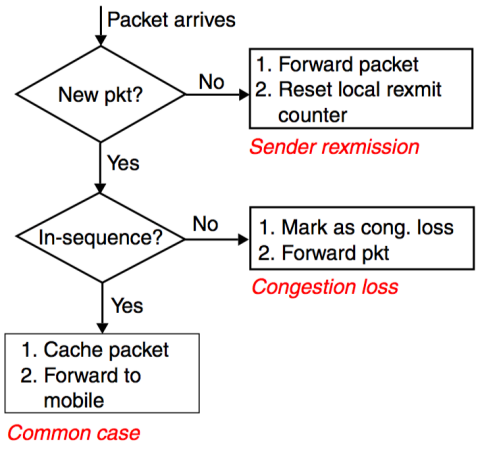


Figure 5: Flow chart for snoop_data()

tween the FH and the MH. Data packets that are received from the FH and MH that contain data are cached at the BS. If the data packet is out of order first it is checked to determine if the sequence number is larger than the previous acknowledgement. If so the data packet is forwarded on as it is likely that the MH also did not receive this packet. If on the other hand the sequence number is less than the previous acknowledgement number, it is likely caused by a acknowledgement being lost, so an acknowledgement is generated by the BS for the last sequence number seen. The flow chart from the original paper describing *snoop_data* is shown in Figure 5

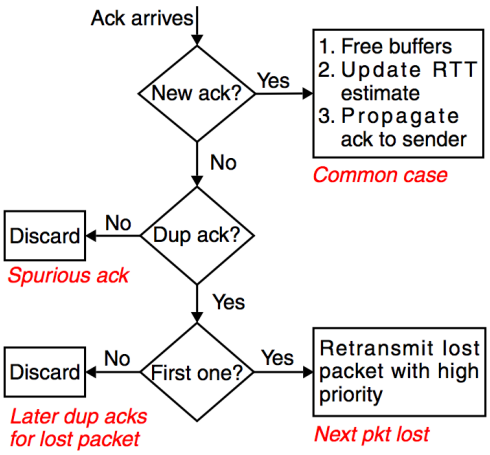


Figure 6: Flow chart for snoop_ack()

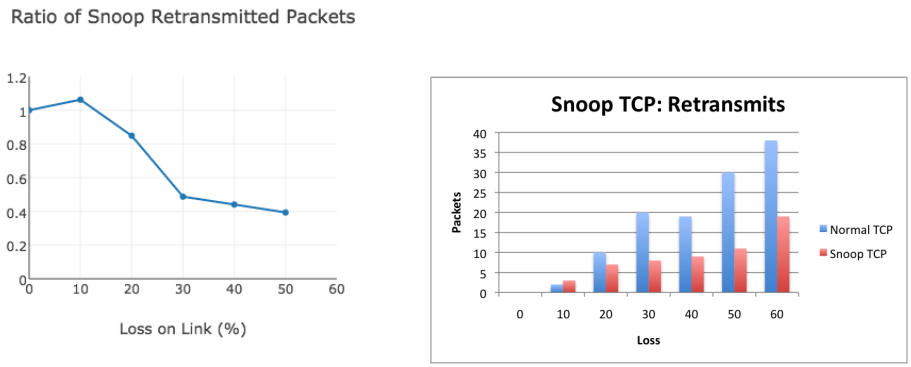
The *snoop_ack* algorithm is responsible for acknowledgements received by the BS.

When a new acknowledgement is received, as before, if the acknowledgement is in-order the acknowledgement is forwarded. Accepted acknowledgements are also responsible for clearing the associated cached data in the BS. If the acknowledgement number of the acknowledgement is less than the previously seen acknowledgement number, the packet is discarded. A more common case is if the current acknowledgement number is the same as the last seen acknowledgment number, indicating a duplicate acknowledgement. When the BS receives a duplicate acknowledgement, the first thing that is done is to check whether this is the first duplicate for a given acknowledgement. If it is, it must be forwarded onward because the previous acknowledgement deleted the corresponding data from the cache. If the acknowledgement is not the first duplicate to reach the BS, it will be discarded along with all future duplicate acknowledgements for the same sequence number. This is also outlined in Figure 6.

3.2.1 Implementation

Our Snoop implementation differs in a few ways. First is we actively spoof packets that are received at the BS. Duplicate acknowledgements with the sACK option set, are dropped, and a new acknowledgment packet is generated based on the sACK option. While this is happening, the BS is actively attempting to get the initial acknowledgement by re-sending the cached data, this differs from Snoop because our acknowledgments do not immediately clear the cache. So in our implementation, a duplicate acknowledgement is rarely forwarded. A duplicate acknowledgement is only forwarded if no acknowledgement has been received within 3 * RTT. If a data packet has not been acknowledged after 1.5 * RTT we proactively re-send the data. Instead of allowing acknowledgements to clear the data cache, we hold onto the data until the timestamp created for the packet on reception exceeds 3*RTT and has also been acknowledged. If by the 3 * RTT timer the data has not been acknowledged, all cached data and acknowledgements are cleared and the duplicate acknowledgement is forwarded. The design of programmable network makes this adjustments easy to make and implement in code for quick prototyping.

Our implementation used NetfilterQueue[29] to integrate with iptables to move packets of a certain type from kernel-space to user-space to be handled by an application. This makes our implementation incredibly slow, a more apt package to use would be netmap[69] or using Linux TUN/TAPs, we chose NetfilterQueue for extremely quick prototyping without modifying the underlying system at all.



(a) The ratio of packets retransmitted by the Snoop protocol compared with TCP Cubic

(b) The total number of retransmits per loss

Figure 7: Our prototypes results implementing TCP Snoop

To test our implementation we used iptables to create loss. We measured the number of retransmitted data packets using tcpdump and tcptrace[62] as our performance metric to evaluate our implementation. Our results from these tests are shown in Figure 7. Our results show that our Snoop implementation sends more packets when the loss is low, but when loss is greater than 10% on the link, Snoop is able to send up to 60% less packets than when using the default TCP implementation. One note of interest is that shown in both Figure 7(a)(b) is that loss at 10% causes a greater number of TCP snoop packets to be sent than normal TCP. The increased number of packets is due entirely to our hyperactive resend policy at the BS. The hyperactive sending has a greater benefit for when less is greater than 10%.

4 Programmable Networks

4.1 Security Concerns

The current state of virtualization today provides the possibility for a viable solution to provide secure executable code in the network. Many approaches have been developed for providing secure virtual environments such as LXC[68] or Minibox[48] to prevent malicious code from interacting with the hypervisor. More recent advances in hardware virtualization include Intel's Software Guard Extension (SGE)[52]. SGE provides applications security from untrusted software with higher privileges.

AEGIS[5], used by SANE[2] is a trusted module boot loader that can be used to provide modules are loaded securely from trusted sources. To a certain degree hypervisors or trusted virtual machine monitors such as Terra[32] have been used for the handling of creating a secure environment for virtual machine based architectures. Additionally, providing security through the language is another method which does not require virtualization. Many languages such as PLAN[37], Sprocket and Spanner[74], and Proof Carrying Code[58] were created before virtualization techniques were possible to address the needs for secure programming languages.

Many early languages with a focus on security relied on propositional logic to justify security[33]. That is to say some policy was defined, and the language provided a guarantee that the policy could not be broken. Other logics were devised that focused purely on the problem of authentication such as BAN logic[14]. Languages were defined in meticulous ways to provide security at the cost of generality. Programming languages were restricted to provide guarantees of code execution over the expressibility of the code. Here in lies the problem for the later mobile codes, utilizing a language with stripped down functionality over an insecure expressive language.

Additional measures could be taken by using virtualization with secure languages. Generally a the filesystem, memory, and resources are virtualized. Meaning there is either an interior monitor in the VM which is faster but less secure[76], or an outside virtual machine monitor which is more secure but slower[70]. An outside

monitor is not accessible to the virtual machine and therefore more secure than the interior monitor. Unlike the fruitful characteristics of a proof for a secure language, virtualization has no proof. Virtualization is to a certain extent not a means of security, many examples of jailbreaks or escaping virtual machines have been demonstrated[13][27], but in conjunction with secure languages can be used in a layered defense implementation.

Any new architecture for allowing executable code in the network would have to use either a secure language or virtualization to provide security guarantees. Virtualization integration in hardware increases the likelihood of implementation compared to a custom defined language. However, virtualization does not provide security. It should be used in conjunction with other methods of providing security. Using stripped down languages such as PLAN or Spanner is one method, another is setting up system level policies to restrict functionality of virtual machines to the system in much the same way as hypervisors.

As briefly mentioned before, programmable networks have to still handle DDOS, MITM, and other type of attacks. Denial of Service and their distributed counterpart attacks focus on the resources of a system and completely exhausting all resources in order to prevent the system from operating normally. The problem with that programmable networks face by resource oriented attacks is the amount of resources required by both the setup and running of code. Programmable networks would require similar methods as current networks to defend against DDOS attacks, however, just as with our modern networks there is no guarantee such an approach would work. It is important to note that a DDOS attack on a programmable network would require significantly less resources on the attacker's side to defeat a programmable network setup. This would mean that programmable networks should use some means of authentication to limit the amount of resources wasted by either requests or bad code. This is one of the benefits to secure languages in that they prevent wasteful code by preventing it from a language syntax point of view. MITM and forgery attacks would be very similar to modern networks and would require the same attitude of authenticating before allowing code or accepting communications.

4.2 Performance Considerations

The additional layers added to provide security and high programability to the network is at the cost of performance. Virtualization technology however has greatly improved with hardware support such as with VT-x [59], AMD-V and SR-IOV [23] as well as being able to provide close to native speeds[91] for containers such that very little performance is lost. NetVM[39] demonstrates how creating a performance based network virtual machine architecture for line-rate forwarding is easily obtainable by utilizing newly available hardware supported virtualization. Performance can also be scaled by utilizing clusters. Clustered software routing such as RouteBrick's[22] RB4 composed of 4 nehaliam servers, running Click[43] forwards rate is 35Gbps for an average workload. In software, forwarding is a CPU bound operation. The more packets (generally smaller size) causes the CPU to work harder and decreases performance, for higher packets per second (pps) forwarding performance drops to 12Gbps in RB4. RouteBricks design scales, adding additional CPUs will increase the CPU-bound operations.

It is also possible to use a Field Programmable Gate Array (FPGA) to run a custom language in hardware. The language can provide performance guarantees by providing a limited set of instructions such as with Tiny Packet Program (TPP)[40][41]. TPP is a set of small messages which utilize 6 instructions: load, store, push, pop, cstore, and cexec. With these 6 instructions network administrators can program how data packets can be forwarded throughout the network at low latency and line rate. The tradeoff to TPP is that there is no option to provide security as a part of the architecture as the assumption is that the network is controlled by a single domain, and the domain will not be compromised.

As with security, performance considerations for code execution in the network can be achieved through the use of efficient systems, languages, or hardware. Using relatively inexpensive generic hardware with complex software that is responsible for forwarding, allows code to be programmable and leaves performance to the cluster as a whole. Or a language running a minimum set of instructions to achieve a small set of programmability can be used at a far cheaper cost.

4.3 Comparison with active networks, NFV, and SDN

The need for a "killer app" is just as relevant for our programmable network architecture as it was for justifying active networks. OpenFlow and NFV have attractive use cases for the data center as well as for production networks. Yet the question we look to answering is: "Does the programmable network architecture have a use case for modern networks that can't be filled by active networks, SDN, or NFV?" We argue the answer to this question is "Yes, in limited circumstances." Programmable networks are not meant for the Google's of the world. For Google, performance is the main concern, but for small networks where some performance can be traded for programmability is where programmable networks can work well.

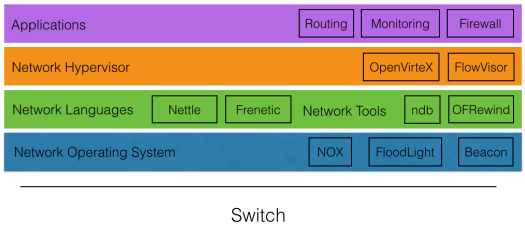


Figure 8: The SDN Stack

OpenFlow's utilization in corporate networks has been very impressive, but it still lacks an easy programmable interface familiar to programmers. Multiple applications have been developed on top of OpenFlow to fill needed functionality gaps left out of the original protocol. For example new powerful declarative languages such as Frenetic[28] and Nettle[84] sit on top of OpenFlow to provide an easy, simple, declarative interface which is not provided by OpenFlow. Even so, a major problem with the OpenFlow protocol is scalability. Due to the very nature of a centralized control plane, many physically distributed, but logically centralized implementations such as Onix[44], HyperFlow[82], and ElastiCon[21] approach it as a distributed system problem trading off consistency, availability, and partitioning. The high level view of SDN and where and how applications work with each other is shown in

Figure 8. The programmable network architecture combines these approaches to achieve a declarative and distributed control plane. The decentralized nature of the programmable network architecture means that distributed algorithms can take advantage of the control plane without managing centralized state, instead state can be managed locally by each node. Even debugging the network using OpenFlow is challenging to infer how race conditions are settled as well as what the state of the network looks like at each switch. Applications such as ndb[35] attempt to debug this complexity and uncertainty from OpenFlow implemented rules.

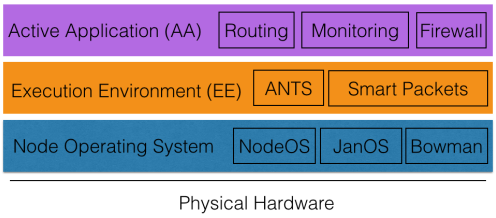


Figure 9: The active network Stack

The active network architecture can also provide a decentralized programmable control plane, but it is limited by the constraints of reprogramming the architecture. Based on the choice of execution environment (ANTS, SANE, etc) and the choice of the NodeOS[66] implementation limits the programmability and flexibility from what is required by the execution environment and what is exposed by the NodeOS. Allowing the programmable network architecture to be completely modular provides this flexibility for prototyping, but it also allows the architecture to be adapted for static environments. We show the high level description of active networks in Figure 9, what we don't show is that in many node operating systems such as JanOS[83], it is only possible to run a single execution environment.

So far, no such Network Function Virtualization implementation has gained as much prominence as OpenFlow has for SDN. This seems to indicate the challenge of creating an implementation that largely contrasts from a hypervisor with pre-built

applications into the network. ClickOS[51] is an implementation which uses KVM and virtual machines with Click installed to deploy software routing throughout the network. What we believe is missing is the ability to deploy this functionality across multiple hypervisors, and allow the functionality to be used for prototyping. In its current iteration NFVs are deployed for production networks supporting middlebox functionality. The programmable network architecture moves the control away from the hypervisor and decentralizes it through code in the network for applications to be implemented or built upon.

The programmable network architecture is different from each of the preceding architectures to make it uniquely distinct in allowing for a more programmable environment. Table 1 summarizes these findings. It is flexible to the conditions of the network and the requirements of the network administrators. Because of this level of programmability, programmable networks can confront obstacles of the other architectures, one example is the need for decentralization of the control plane for scalability.

	active networks	SDN	NFV	programmable network
Software Complexity	OS + Language	Controller	Hypervisor	Hypervisor + Language
Control Plane	Decentralized	Centralized	Centralized	Decentralized
Development Flexibility	High	Low	Medium	High
Architecture Flexibility	Static	Static	Dynamic	Dynamic
Run-time Environment	Distributed	Local	Local	Distributed

Table 1: Comparison of each programmable architecture

Control plane decentralization is a problem with no currently proposed solution so far for OpenFlow. In part this is due to the OpenFlow architecture's popularity with centralized control in the network. Network computations have become more robust as computation and control is spread from a single node, to a group of nodes, until it is spread to every node. Using the programmable network architecture the cost of decentralizing control comes from changing switches from being simple and dumb to intelligent and complex, a shared consequence of using NFV as well. Complexity in the network has been shunned as it violates the end-to-end principle[72], however many of the SDN and NFV solutions also violate the principle by allowing

software control in the network. We see a trend towards intelligence creeping into the network from the edges, as applications start to require additional knowledge to achieve distributed goals. Programmable networks pushes the intelligence into the network making applications more context aware of their environments and are able to become more efficient. As the network intelligence grows, the network can counteract lossy portions of the network by implementing hop-by-hop ARQ, or distributed queues for buffering to adapt to the network conditions. More complex features can be motivated in intelligent transport systems with automated vehicle navigation, having swarms of cars communicate conditions dynamically without centralization to achieve a shared goal. Future internet protocols and architectures can also be supported such as Information Content Networking[31] or new routing protocols such as label swapping[47] quickly and easily using such a programmable network architecture.

5 Emerging Applications

We believe that based on the trend of the growing number of devices in the network, that scalability and robustness for applications will dictate a greater need for a decentralized control plane. Recent research in Content Centric Networking (CCN) will also require a decentralized control plane. The programmable network architecture natively presents local caches that can be used for interests messages. This feature will greatly benefit CCN developers. Lastly, there is a need for a decentralized control plane in intelligent transport systems (ITS) where mobility in hybrid networks makes a centralized control plane infeasible. For each of these applications having an architecture that can scale from the decentralization of the architecture is crucial, having the architecture also support programability means that it can dynamically change to network conditions as well as protocol changes swiftly.

5.1 Decentralized Control Plane

The decentralized control plane is essential to the future of networking. Scale will bring about the end of any centralized solutions. This will force centralized algo-

rithms to operate on small networks while inter-networks would be required to use decentralized solutions. While the decentralized control plane provides a more robust and scalable architecture than the centralized control plane, centralized logic reduces the required complexity of the network. We have seen this behavior before in networking, such as when networks moved from centralized circuits to packet switch networks and then back to virtual circuit overlays. The eagerness of network administrators to return to simple design for control is all too well known. We believe that given a decentralized control plane, central overlays will become a manifestation to return to simple designs.

The decentralized control plane works by allowing each node to be its own control interface as well as influence other nodes. Unlike with OpenFlow, the control plane is not merged with other devices but works independently. This leads to the development of algorithms which can be implemented in a distributed fashion and design protocols with convergent behavior towards a shared goal. We showed one example of this above with our Bellman-Ford implementation. Our implementation allows each node independence and uses a distributed message passing method to achieve convergence towards a shared routing state. The programmable network architecture enables the use of the decentralized control plane as well as a mechanism of upstream nodes to leave soft state, the soft-state allows for caching, but it also allows for combined computation for groups of devices.

5.2 Content Centric Networking

Content Centric Networking was first developed at Xerox PARC and now has a developed protocol, CCNx[63]. CCN changes the network paradigm from being host oriented connections to direct connections to data. The idea is that as we scale our internet, the host centric model becomes more convoluted and requires more and more engineering to maintain the overall system. Instead of having the content for user */lthurlow* to be hosted at 128.114.49.139, */lthurlow* becomes the destination and using *Interest* messages the client can find the data without dealing with the underlying framework of where the host is located. A requester for a given piece

of content would send an *Interest* packet that will follow the route advertisement for the content, and once found, a response message containing the content follows the initial *Interest* request's path back to the requester. The concept is now more reasonable than ever with cheaper networking hardware and storage it has become easy and simple for the network to encompass the requirements of caching in the network.

The programmable network architecture enables the in-network caching as a part of the architecture itself. Devices would not need to be solely CCNx switches and routers, that software would be loaded into the static environment of the programmable network architecture. The more interesting application of using the programmable network architecture for CCN is in how routing and control will be done, with such a large scale, a centralized solution would not work, therefore using the decentralized nature of programmable networks for control of the forwarding and routing of content is more appealing.

5.3 Intelligent Transport Systems

One application which we perceive will require a decentralized control plane is the intelligent transport system (ITS). An ITS would be developed with autonomous vehicles in mind. The infrastructure of the ITS would be a hybrid systems with ad-hoc networks made between clusters of autonomous vehicles and the infrastructure of road side units. Road side units will occasionally communicate with vehicle clusters of information related to other clusters further ahead. The information can contain traffic information such as accidents, lane closures, or non automated vehicles as illustrated in Figure 10. Each cluster can designate leaders such as using a paxos protocol[45], these leaders can act as distributed controllers for the local clusters on the roads, communicating with the road side units also acting as controllers need to delegate control to each elected leader for the cluster to establish communication.

The need for control delegation is closely related to work on OpenFlow's east-west interface[67] to create a decentralized OpenFlow implementation to allow communications between separated control planes[73]. However, even within the cluster there

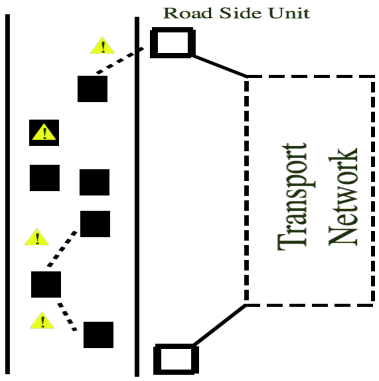


Figure 10: An ITS system

needs to be an essence of delegation of tasks as either more vehicles join the cluster, or are out of range of the leader, so to achieve this objective multiple methods need to be developed for OpenFlow solutions to work. The programmable network architecture does not suffer from the centralized control and therefore the ownership of control. In the decentralized architecture elections are not necessary for delegating control, instead a distributed adaptive routing algorithm can be used. In conjunction with the inherent caching mechanism allows the architecture to store messages closer to the edge of the cluster to quickly exchange messages with road side units.

6 Related Work

6.1 Active Networks

Active networks were first proposed in the seminal paper by Tennenhouse and Wetherall[81]. They present the active networks as an architecture for decoupling network services from the underlying hardware to accelerate the pace of innovation in the network. Tennenhouse and Wetherall's architecture is split into two approaches towards implementation. The first is the discrete approach, which uses programmable switches. The second is the integrated approach, which uses capsules. The programmable switch used an API model for calls between applications and switches that eventually lead to the IEEE P1520[10] standard for programmable Asynchronous Transfer Mode switches. The capsule based approach however used

capsules which carried executable code. The capsule based approach led first to the Active IP option[88] from Wetherall and Tennenhouse using the option field in IP packets to indicate active packets. The payload of the capsule would contain Tcl code. While Active IP had the potential to be language agnostic, no mechanisms or architecture for handling other languages was approached or discussed. Wetherall et. al. Then developed ANTS[89], a complete active network environment based on the Java programming language. ANTS was the first complete active network implementation, it used the JVM to sandbox code and guarantee performance and security constraints. ANTS introduced soft-storage for protocols, however storage was limited to only the same protocol.

The active SwitchWare architecture was designed by Alexander et al.[3]. Unlike ANTS, SwitchWare used PLAN[37], a typed lambda calculus language. PLAN was designed as a standalone strongly typed language. The language was deliberately limited in functionality making arbitrary protocols impossible to implement with PLAN. SwitchWare at a high level does not allow PLAN code to store or change state on active nodes. To better address security concerns, SANE[2] was developed to handle authentication and secure bootstrapping of modules in the SwitchWare environment. Eventually a more secure version of PLAN was developed with SNAP[56]. SNAP allowed the compiler to *a priori* determine runtime bounds for CPU, Memory, and Bandwidth. The trade-offs made by SwitchWare was to secure the network at the cost of flexibility, whereas ANTS provided flexibility at the cost of security and performance.

Netserv[79][46] was the latest active networking architecture. Based off the discrete approach, NetServ uses out-of-band singling to load modules into NetServ devices on the internet rather than allowing executable code. The architecture allows modules to be loaded to remote hosts in other domains, while providing authentication through Public Key Infrastructure. The modules are run on top of the JVM to provide security to the underlying virtual machine and host machine.

Within the active networking research area there was even more work done on active networking languages than on architectures. Smart Packets[74] was designed as a

language to address network administration. Smart Packets was the culmination of two separate languages, Spanner and Sprocket. Spanner was the low level CISC assembly language, which Sprcoket, a C type language compiled into. For general programmability Sprocket was recommended, but for speed and security Spanner was the more effective language. Both were made to compile into small byte code, the design was that active programs would not exceed an ethernet frame and fit within a single packet using the ANEP[4] protocol. The protocol dictated the header fields for active packets, and looks very much like an IP header with additional fields including its own set of source, destination, and checksum. It also contains an authentication field to provide a means for authenticating active packets. More recently, Tiny Packet Programs (TPP)[40] illustrated that with a small subset of x86-like assembly level code can be executed on netFPGA[49] devices at line rates.

6.2 Active Node Operating Systems

Node operating systems started showing up later once a model was built for the active networking architecture with node operating systems at the bottom, execution environments which handled code execution and the interface with the node operating system, and on top were the active applications, the logical abstraction of the code being executed in the execution environment (refer back to Figure 9). Peterson et al. implemented their NodeOS interface[66][65] for the Scout [57], JanOS[83], and exokernel[24] operating systems. SPIN[9] and xkernel[38] were other operating systems built for general purpose network operating systems that made little impact on active networking development. Part of the design of the NodeOS interface was to separate the operating system from the run-time environment to provide support for multiple languages. The interface then abstracts threads, memory, input/output, and files to the execution environment to provide the necessary communication to enable active networks. Operating systems specifically designed for active networks such as JanOS and BowmanOS[55]. JanOS was developed for ANTS and Java based languages using the ANTSR run-time on top of the JanOS virtual machine. Bowman however allows for multiple execution environments, but the number of

fine-grained abstractions exposed to the execution environment is more limited than both JanOS and the NodeOS interface. There was also the FAIN[30] which from both an architectural and operating systems point of view approached the problems completely different. The architecture of FAIN was based on one where Internet Service Providers (ISP) would be running active networks and customers of the ISP would request services. This model reflected much more of a NFV network of today than an active network of the past.

6.3 Software Defined Networking

The predecessor to OpenFlow[53] was Ethane[17]. Ethane separated the control plane using a controller and an Ethane switch which contained the flow tables as a means to enforce security policies in the network. OpenFlow used a similar model compromised also on centralized logic in a controller but focused on network management and control through the OpenFlow protocol. Each switch using the OpenFlow protocol contains a flow table, the contents of the flow table are made up of a rules, actions, and statistics. A rule is based on matching the header of an incoming packet or flow. If a match occurs, the action associated with the rule is triggered. The action can drop, forward, or otherwise modify the frame or packet. The control for the network is stored in the controller, NOX[34] was the first controller software developed for the OpenFlow protocol. Additional controllers were later developed to expand the support based beyond the C programming language to other languages such as Java and Python[75].

Higher level programming languages like Nettle[84] and Frenetic[28] were developed to make network programming using OpenFlow more declarative. These languages also helped programmers avoid issues with nuances of OpenFlow such as rule ordering and race conditions in OpenFlow. However additional tools such as ndb[35] and OFRewind[90] were created to assist programmers to trace and debug issues with OpenFlow rules that may be caused by unseen network problems.

To broaden the applicability of a centralized control plane, multiple implementations were developed to distribute the control plane across multiple OpenFlow con-

trollers. Onix[44] creates a distributed system onto of multiple controllers. Onix introduces the Network Information Base (NIB), the NIB is the distributed control plane, however the NIB trade-offs consistency, availability, and partitioning of the OpenFlow control logic. Hyperflow[82] is another method for distributing the control plane across multiple controllers. Hyperflow modifies NOX and implements a network-wide control plane, controllers manage only the set of local switches, and consistency is maintained through managing event publishing and replays during failures.

Programming Protocol-Independent Packet Processors (P4)[11] is the newest development in the SDN arena. P4 was developed at Stanford as a successor to OpenFlow due to the large amount of changes to the protocol and the amount of time it has taken for changes to be pushed to the protocol. P4 places a compiler that is very close to C on network devices. The compiler takes the C-like code and creates a JSON file which maps outputs between the various stages in the pipeline as shown in Figure 3. The end result is compiled P4 code which uses a set of tables to forward packets from one element in the pipeline to another element. The advantage over OpenFlow is that with P4 code can be written to develop the protocol itself. The protocols are dictated by code, very similar to active networks, with the key distinction that code is not active. The compiled code is much closer to the discrete approach with programmable switches and since the code is very close to C, it is very quick. There are limits to P4 such that modifications to the protocol cannot occur live, and interfaces cannot currently be for both control and data.

6.4 Network Function Virtualization

Predecessors to the NFV architecture started with the Genesis kernel[15] which allowed creating routlets spawn as child processes much in the same way as implementing network functions. Routlets could be created, managed, and then destroyed when no longer necessary to provide network functionality. However, virtualization came much later to research, following the initial white paper in the field [18], implementations such as netvm[39] and clickOS[51] add the aspect of virtual environments

with programmable interfaces to extend the virtualization of network functionality. Many companies use proprietary products such as Cisco's Embrane or VMWare's ESX platform as more of the NFV field has shifted to industry.

6.5 Secure Code

Omniware[1] and Proof Carrying Code[58] were other systems and languages not directly designed for active networking which provide safety to the host system. Omniware used the OmniVM, a virtual machine to segment address spacing and enforce these permissions. Omniware was designed as a virtual machine sandbox for languages, allowing non-safe languages to be executed with safety to the host system. Proof Carrying Code however was designed as a language that would require a mathematical proof of axioms and rules from the code producer. The system executing the code then verifies the code's proof of its safety policies, and when validated is allowed to execute. The emphasis is on the ability to encode safety predicates into the proof to guarantee that the code will abide by the constraints at run time.

Minibox[48] is a good example of a sandbox environment for executing of untrusted x86 code on commodity hardware. Minibox protects the guest operating system using memory isolation and validating calls between the isolated exaction environment and the guest operating system. SIM[76] is an alternate method for one-way protection of the hypervisor placing a monitor into the guest virtual machine, SIM uses hardware virtualization and memory protection to allow itself to run protected from guest operating system instructions while guaranteeing safety of the hypervisor.

6.6 Software Performance

One of the most important points of securing active code is what the cost is to performance. Using methods like RouteBricks[22], which decentralizes the workload into a cluster, then load-balances pathways in the cluster to provide high throughput for software routing. RouteBricks valiant load balancing as a key component to maintaining an even workload across each node in the cluster. Scaling more servers

in a RouteBricks cluster is possible by extending the number of servers in the intermediate mesh as the bottleneck is not inter-cluster communication but the shared bus between CPUs and Memory with a node in the cluster. When testing smaller packet sizes to stress CPU, the performance of RB4, a 4 Nehalem server setup, was around 12Gbps, showing that CPU-bound computation such as active code may be possible for line-rate by extending the RB4 cluster.

7 Conclusion

In this thesis we have shown that executable code is able to operate on the critical path for packet forwarding. We have reviewed processor design and discussed the issues related to power barriers and physical properties of chip layouts that prevent scale-up solutions to working in a programmable network. We proposed a solution to this problem by scaling-out rather than scaling-up utilizing the RouteBrick cluster design. RouteBrick enables forwarding to linearly scale with the number of CPUs and interfaces added to the cluster. We have also discussed P4, a more recent SDN implementation capable of utilizing precompiled code to forward packets at line-rates. While programmable network are not as efficient in forwarding as Application-Specific Integrated Circuit (ASIC) used by industry leaders such as Cisco and Juniper, the benefits of programmable networks to decouple each component of the network allows for fast pace modifications, innovation, and counteracts the ossification that has overtaken the IP/TCP stack.

We have also examined how performance suffers due to the safeguards that must be placed in systems that allow executable code in order to prevent code from exploiting the framework. By virtualizing the environment handling executable code, per-packet latency grows but provides protections against malicious code. Furthermore, with virtualization mechanisms built into many modern processors, the latency caused by virtualization is greatly minimized. The alternate approach for securing systems by either minimizing the instruction set to a trivial number of instructions, or handicapping the language to prevent certain types of control flow logic harm the overall programmability of the system. Language based approaches pre-

vent sufficient high-level declarative programability that is wanted by implementers of programmable networks using executable code.

Never the less, programmability in the network is possible. We have shown that our SCAN implementation is capable of implementing distributed Bellman-Ford and Berkeley's TCP Snoop protocols. SCAN is capable of detecting processes running on remote nodes. It uses this capability to bootstrap the appropriate set of protocols as required by the network or traffic flowing through the node. The SCAN implementation was able to use this benefit to create on-demand TCP Snoop connections to minimize the number of packets dropped over multiple lossy links.

The future of programmable networks is still evolving as the industry attempts to push programmability deeper into the hardware. There are a few current technologies that will begin to shape how we interact with programmable networks, the first is phase-change RAM[85][19], a non-volatile memory. PCRAM or NVRAM, allows for interesting integration into programmable networks. NVRAM allows software implementations to recover within milliseconds after a power failure. Phase-change RAM will also shape the CCN architecture as well as the P4 language which are actively research and make heavy use of software on the critical path for packet forwarding. Intel's Data Plane Development Kit (DPDK) is also a new method for modifying forwarding states using software. The benefit of having additional hardware support places less emphasis on the performance tuning required by a system and more emphasis on the breadth of functionality[93][64]. Companies such as Facebook are making an active push towards what they call "white-boxes", boxes that contain two types of processors[77]. The ASIC that runs the black-box portion is in charge of forwarding packets quickly. The offload processor which makes up the white-box portion is a processor designed to do general functionality and to move packets to the black-box when an operation is completed. We expect to see more designs in the future which adapt these new technologies to make programmable networks more common place in traditional networks.

References

[1] A. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe, "Efficient and Language-Independent Mobile Programs," in *ACM SIGPLAN Notices*, vol. 31. ACM, 1996, pp. 127–136.

[2] D. S. Alexander, W. Arbaugh, A. Keromytis, and J. Smith, "A Secure Active Network Environment Architecture: Realization in SwitchWare," *Netwrk. Mag. of Global Internetwkg.*, vol. 12, no. 3, pp. 37–45, May 1998.

[3] D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunter, S. Nettles, and J. Smith, "The SwitchWare Active Network Architecture," *Network*, vol. 12, no. 3, pp. 29–36, May 1998.

[4] D. Alexander, G. Minden, D. Wetherall, A. Keromytis, B. Braden, C. Gunter, and A. Jackson, "Active Network Encapsulation Protocol," 1997. [Online]. Available: http://www.cis.upenn.edu/~switchware/ANEP/docs/ANEP.txt

[5] W. Arbaugh, D. J. Farber, J. M. Smith *et al.*, "A Secure and Reliable Bootstrap Architecture," in *Proceedings. 1997 IEEE Symposium on Security and Privacy.* IEEE, 1997, pp. 65–71.

[6] A. Bakre and B. Badrinath, "I-TCP: Indirect TCP for Mobile Hosts," in *Proceedings of the 15th International Conference on Distributed Computing Systems.* IEEE, May 1995, pp. 136–143.

[7] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz, "Improving TCP/IP Performance over Wireless Networks," in *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '95. New York, NY, USA: ACM, 1995, pp. 2–11.

[8] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[9] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility Safety and Performance in the SPIN Operating System," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 267–283.

[10] J. Biswas, A. Lazar, J.-F. Huard, K. Lim, S. Mahjoub, L.-F. Pau, M. Suzuki, S. Torstensson, W. Wang, and S. Weinstein, "The IEEE P1520 Standards Initiative for Programmable Network Interfaces," *Communications Magazine, IEEE*, vol. 36, no. 10, pp. 64–70, Oct 1998.

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.

[12] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," in *ACM SIGCOMM Computer Communication Review*, vol. 43. ACM, 2013, pp. 99–110.

[13] M. Brocker and S. Checkoway, "iSeeYou: Disabling the MacBook webcam indicator LED," in *23rd USENIX Security Symposium*. USENIX Association, 2014, pp. 337–352.

[14] M. Burrows, M. Abadi, and R. M. Needham, "A Logic of Authentication," in *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 426. The Royal Society, 1989, pp. 233–271.

[15] A. Campbell, H. De Meer, M. Kounavis, K. Miki, J. Vicente, and D. Villela, "The Genesis Kernel: A Virtual Network Operating System for Spawning Network Architectures," in *Second Conference on Open Architectures and Network Programming Proceedings*, ser. OPENARCH '99. IEEE, 1999, pp. 115–127.

[16] A. T. Campbell, H. G. De Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela, "A Survey of Programmable Networks," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 2, pp. 7–23, 1999.

[17] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," *SIGCOMM Computer Communications Review*, vol. 37, no. 4, pp. 1–12, Aug. 2007.

[18] M. Chiosi, D. Clarke, P. Willis, A. Reid, J. Feger, M. Bugenhagen, W. Khan, M. Fargano, C. Cui, H. Denf *et al.*, "Network Functions Virtualization: An Introduction, Benefits, Enablers, Challenges and Call for Action," in *SDN and OpenFlow World Congress*, 2012, pp. 22–24.

[19] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O Through Byte-Addressable, Persistent Memory," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009, pp. 133–146.

[20] D. Decasper and B. Plattner, "DAN: Distributed Code Caching for Active Networks," in *Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, ser. INFOCOM '98, vol. 2. IEEE, 1998, pp. 609–616.

[21] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. Kompella, "Towards an Elastic Distributed SDN Controller," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '13. New York, NY, USA: ACM, 2013, pp. 7–12.

[22] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, "RouteBricks: Exploiting Parallelism to Scale Software Routers," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 15–28.

[23] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High Performance Network Virtualization with SR-IOV," *Journal of Parallel and Distributed Computing*, vol. 72, no. 11, pp. 1471–1480, 2012.

[24] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr., "Exokernel: An Operating System Architecture for Application-level Resource Management," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 251–266.

[25] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in *2011 38th Annual International Symposium on Computer Architecture*. IEEE, 2011, pp. 365–376.

[26] N. Feamster, J. Rexford, and E. Zegura, "The Road to SDN: An Intellectual History of Programmable Networks," *SIGCOMM Computer Communications Review*, vol. 44, no. 2, pp. 87–98, Apr. 2014.

[27] P. Ferrie, "Attacks on More Virtual Machine Emulators," *Symantec Technology Exchange*, p. 55, 2007.

[28] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A Network Programming Language," in *ACM SIGPLAN Notices*, vol. 46. ACM, 2011, pp. 279–291.

[29] fqrouter, "python-netfilterqueue," https://github.com/fqrouter/python-netfilterqueue, 2013.

[30] A. Galis, B. Plattner, J. M. Smith, S. G. Denazis, E. Moeller, H. Guo, C. Klein, J. Serrat, J. Laarhuis, G. T. Karetsos, and C. Todd, "A Flexible IP Active Networks Architecture," in *Proceedings of the Second International Working Conference on Active Networks*, ser. IWAN '00. London, UK, UK: Springer-Verlag, 2000, pp. 1–15.

[31] J. Garcia-Luna-Aceves, "Name-Based Content Routing in Information Centric Networks using Distance Information," in *Proceedings of the 1st international conference on Information-centric networking*. ACM, 2014, pp. 7–16.

[32] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-Based Platform for Trusted Computing," in *ACM SIGOPS Operating Systems Review*, vol. 37. ACM, 2003, pp. 193–206.

[33] J. Glasgow, G. MacEwen, and P. Panangaden, "A Logic for Reasoning about Security," *ACM Transactions on Computer Systems*, vol. 10, no. 3, pp. 226–264, 1992.

[34] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 3, pp. 105–110, 2008.

[35] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the Debugger for my Software-Defined Network?" in *Proceedings of the first workshop on Hot topics in software defined networks*. ACM, 2012, pp. 55–60.

[36] B. Heller, R. Sherwood, and N. McKeown, "The Controller Placement Problem," in *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, ser. HotSDN '12. New York, NY, USA: ACM, 2012, pp. 7–12.

[37] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles, "PLAN: A Packet Language for Active Networks," in *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '98. New York, NY, USA: ACM, 1998, pp. 86–93.

[38] N. C. Hutchinson and L. L. Peterson, "The X-Kernel: An Architecture for Implementing Network Protocols," *IEEE Trans. Softw. Eng.*, vol. 17, no. 1, pp. 64–76, Jan. 1991.

[39] J. Hwang, K. Ramakrishnan, and T. Wood, "NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms," *Transactions on Network and Service Management*, vol. 12, no. 1, pp. 34–47, 2015.

[40] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières, "Millions of Little Minions: Using Packets for Low Latency Network Programming and

Visibility," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 3–14.

[41] V. Jeyakumar, M. Alizadeh, C. Kim, and D. Mazières, "Tiny Packet Programs for Low-Latency Network Control and Monitoring," in *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*. ACM, 2013, p. 8.

[42] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "kvm: the Linux Virtual Machine Monitor," in *Proceedings of the Linux symposium*, vol. 1, 2007, pp. 225–230.

[43] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–297, Aug. 2000.

[44] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A Distributed Control Platform for Large-scale Production Networks," in *Proceedings of the Eighth USENIX Symposium on Operating Systems Design and Implementation*, vol. 10. USENIX Association, 2010, pp. 1–6.

[45] L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

[46] J. W. Lee, R. Francescangeli, J. Janak, S. Srinivasan, S. Baset, H. Schulzrinne, Z. Despotovic, W. Kellerer *et al.*, "Netserv: Active Networking 2.0," in *International Conference on Communications Workshops*. IEEE, 2011, pp. 1–6.

[47] B. Levine and J. Garcia-Luna-Aceves, "ImprovingInternet Multicast with Routing Labels," in *Network Protocols, 1997. Proceedings., 1997 International Conference on*. IEEE, 1997, pp. 241–250.

[48] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "MiniBox: A Two-way Sandbox for 86 Native Code," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 409–420.

[49] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "NetFPGA– An Open Platform for Gigabit-Rate Network Switching and Routing," in *International Conference on Microelectronic Systems Education*, ser. MSE '07. IEEE, 2007, pp. 160–161.

[50] G. Malkin, "RIP Version 2," Internet Requests for Comments, RFC Editor, RFC 2453, November 1998. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2453.txt

[51] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the Art of Network Function Virtualization," in *11th USENIX Symposium on Networked Systems Design and Implementation*. USENIX Association, 2014, pp. 459–473.

[52] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proceedings of the Second International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013, pp. 1–1.

[53] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Computer Communications Review*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[54] D. Merkel, "Docker: Lightweight Linux Containers for Consistent Development and Deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[55] S. Merugu, S. Bhattacharjee, E. Zegura, and K. Calvert, "Bowman: A Node OS for Active Networks," in *Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, ser. INFOCOM '00, vol. 3. IEEE, Mar 2000, pp. 1127–1136 vol.3.

[56] J. Moore, M. Hicks, and S. Nettles, "Practical Programmable Packets," in *Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, ser. INFOCOM '01, vol. 1. IEEE, 4 2001, pp. 41–50 vol.1.

[57] D. Mosberger and L. L. Peterson, "Making Paths Explicit in the Scout Operating System," in *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, ser. Operating Systems Design and Implementation. New York, NY, USA: USENIX Association, 1996, pp. 153–167.

[58] G. C. Necula and P. Lee, "Safe, Untrusted Agents using Proof-Carrying Code," in *Mobile Agents and Security*. Springer, 1998, pp. 61–91.

[59] G. Neiger, A. Santoni, F. Leung, D. Rodgers, and R. Uhlig, "Intel Virtualization Technology: Hardware Support for Efficient Processor Virtualization," *Intel Technology Journal*, vol. 10, no. 3, 2006.

[60] B. Nunes, M. Mendonca, X.-N. Nguyen, K. Obraczka, and T. Turletti, "A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks," *Communications Surveys Tutorials*, vol. 16, no. 3, pp. 1617–1634, Third 2014.

[61] E. L. Nygren, S. J. Garland, and M. F. Kaashoek, "PAN: A High-Performance Active Network Node Supporting Multiple Mobile Code Systems," in *Second Conference on Open Architectures and Network Programming Proceedings*, ser. OPENARCH '99. IEEE, 1999, pp. 78–89.

[62] S. Ostermann, "Tcptrace," 2005. [Online]. Available: http://www.tcptrace.org/

[63] PARC, "CCNx Project." [Online]. Available: http://www.ccnx.org/

[64] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal, "Fastpass: A Centralized Zero-Queue Datacenter Network," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 4, pp. 307–318, 2015.

[65] L. Peterson, Y. Gottlieb, M. Hibler, P. Tullmann, J. Lepreau, S. Schwab, H. Dandekar, A. Purtell, and J. Hartman, "An OS Interface for Active Routers,"

IEEE Journal on Selected Areas in Communications, vol. 19, no. 3, pp. 473–487, Mar 2001.

[66] L. Peterson, "NodeOS Interface Specification," Active Network NodeOS Working Group, Tech. Rep., 01 2001. [Online]. Available: protocols.netlab.uky.edu/~calvert/nodeos-latest.ps

[67] K. Phemius, M. Bouet, and J. Leguay, "Disco: Distributed Multi-Domain SDN Controllers," in *Network Operations and Management Symposium*. IEEE, 2014, pp. 1–4.

[68] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan, "Security of OS-Level Virtualization Technologies," in *Secure IT Systems*. Springer, 2014, pp. 77–93.

[69] L. Rizzo, "Netmap: A Novel Framework for Fast Packet I/O," in *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 9–9.

[70] M. Rosenblum and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," *Computer*, vol. 38, no. 5, pp. 39–47, May 2005.

[71] A. Rowstron and P. Druschel, "Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems," in *Middleware 2001*. Springer, 2001, pp. 329–350.

[72] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems*, vol. 2, pp. 277–288, 1984.

[73] M. Santos, B. Nunes, K. Obraczka, T. Turletti, B. T. de Oliveira, C. B. Margi *et al.*, "Decentralizing SDN's Control Plane," in *2014 IEEE 39th Conference on Local Computer Networks*. IEEE, 2014, pp. 402–405.

[74] B. Schwartz, A. Jackson, W. Strayer, W. Zhou, R. Rockwell, and C. Partridge, "Smart Packets for Active Networks," in *Second Conference on Open Architectures and Network Programming Proceedings*, ser. OPENARCH '99, Mar 1999, pp. 90–97.

[75] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov, and R. Smeliansky, "Advanced Study of SDN/OpenFlow Controllers," in *Proceedings of the 9th Central & Eastern European Software Engineering Conference in Russia*, ser. CEE-SECR '13. New York, NY, USA: ACM, 2013, pp. 1:1–1:6.

[76] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi, "Secure in-VM Monitoring Using Hardware Virtualization," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 477–487.

[77] A. Simpkins. (2015) Facebook Open Switching System ("FBOSS") and Wedge in the open. [Online]. Available: https://code.facebook.com/posts/843620439027582/

[78] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano *et al.*, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. ACM, 2015, pp. 183–197.

[79] S. R. Srinivasan, J. W. Lee, E. Liu, M. Kester, H. Schulzrinne, V. Hilt, S. Seetharaman, and A. Khan, "Netserv: dynamically deploying in-network services," in *Proceedings of the 2009 workshop on Re-architecting the internet*. ACM, 2009, pp. 37–42.

[80] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications," *ACM SIGCOMM Computer Communication Review*, vol. 31, no. 4, pp. 149–160, 2001.

[81] D. L. Tennenhouse and D. J. Wetherall, "Towards an Active Network Architecture," *SIGCOMM Computer Communications Review*, vol. 37, no. 5, pp. 81–94, Oct. 2007.

[82] A. Tootoonchian and Y. Ganjali, "HyperFlow: A Distributed Control Plane for OpenFlow," in *Proceedings of the 2010 Internet Network Management Confer-*

ence on Research on Enterprise Networking. USENIX Association, 2010, pp. 3–3.

[83] P. Tullmann, M. Hibler, and J. Lepreau, "Janos: a Java-Oriented OS for Active Network Nodes," *Journal on Selected Areas in Communications*, vol. 19, no. 3, pp. 501–510, Mar 2001.

[84] A. Voellmy and P. Hudak, "Nettle: Taking the Sting Out of Programming Network Routers," in *Practical Aspects of Declarative Languages*. Springer, 2011, pp. 235–249.

[85] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," *ACM SIGPLAN Notices*, vol. 46, no. 3, pp. 91–104, 2011.

[86] I. Wakeman, A. Jeffrey, T. Owen, and D. Pepper, "Safetynet: A Language-Based Approach to Programmable Networks," *Computer Networks*, vol. 36, no. 1, pp. 101–114, 2001.

[87] D. Wetherall, "Active Network Vision and Reality: Lessons from a Capsule-Based system," in *Proceedings DARPA Active NEtworks Conference and Exposition*, 5 2002, pp. 25–40.

[88] D. J. Wetherall and D. L. Tennenhouse, "The ACTIVE IP Option," in *Proceedings of the 7th Workshop on ACM SIGOPS European Workshop: Systems Support for Worldwide Applications*, ser. EW 7. New York, NY, USA: ACM, 1996, pp. 33–40.

[89] D. Wetherall, J. V. Guttag, and D. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," in *Open Architectures and Network Programming, 1998 IEEE*, Apr 1998, pp. 117–129.

[90] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "OFRewind: Enabling Record and Replay Troubleshooting for Networks," in *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIX ATC'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 29–29.

[91] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. De Rose, "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," in *2013 21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*. IEEE, 2013, pp. 233–240.

[92] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: A Fault-Tolerant Wide-Area Application Infrastructure," *ACM SIGCOMM Computer Communication Review*, vol. 32, no. 1, pp. 81–81, 2002.

[93] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen, "Scalable, High Performance Ethernet Forwarding with cuckooswitch," in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*. ACM, 2013, pp. 97–108.