# Maturing of OpenFlow and Software Defined Networking through Deployments

Masayoshi Kobayashi[†⋆], Srini Seetharaman[‡], Guru Parulkar[†],
Guido Appenzeller[Ш], Joseph Little[†], Johan van Reijendam[†],
Paul Weissmann[‡], Nick McKeown [†]
[⋆] NEC Corporation, [‡] Deutsche Telekom,
[†] Stanford University, [Ш] Big Switch Networks
Contact author: seethara@cs.stanford.edu, +1-650-335-4130

**Abstract**

Software-defined networking (SDN) has emerged as a new paradigm of networking that enables evolvable and programmable networks allowing network operators, owners, vendors, and even third parties to innovate and create new capabilities at a faster pace. The SDN paradigm shows potential for all domains of use including the data center, cellular, service provider, enterprise, and home. In this paper we present SDN deployments that we did on our campus and other deployments that we led with partners over four years in four phases. Our deployments include the first ever SDN prototype in a lab to a (small) global deployment. The deployments combined with demonstrations of new networking capabilities enabled by SDN played an important role in maturing of SDN and its ecosystem so far. We share our experiences and lessons learned that have to do with demonstration of SDN potential; influence on successive OpenFlow specifications; evolution of SDN architecture; performance of SDN and various components; and growing the ecosystem.

*Keywords:* OpenFlow; SDN; GENI; Deployments; Experience

## 1. Introduction

Software-defined Networking (SDN) is a new approach to networking that has the potential to enable on-going network innovation and enable the network as a programmable pluggable component of the larger cloud infrastructure. Key aspects of SDN include: separation of data and control planes; a uniform vendor-agnostic interface such as *OpenFlow* between control and data planes; logically centralized

control plane, realized using a network OS, that constructs and presents a logical map of the entire network to services or *network control applications* on top; and slicing and virtualization of the underlying network. With SDN, a researcher, network administrator, or third party can introduce a new capability by writing a software program that simply manipulates the logical map of a slice of the network.

SDN has been gaining momentum in both research and industry. Most network operators and owners are actively exploring SDN. For example, Google has switched over to OpenFlow and SDN for its inter-datacenter network [14] and NTT Communications has announced OpenFlow-based Infrastructure as a Service (IaaS) [29]. Similarly many network vendors have announced OpenFlow-enabled switches as products and have outlined their strategies for SDN [34]. The research community has had several workshops on SDN during the past 12-18 months where diverse SDN research topics have been covered.

In this paper we share our experiences with OpenFlow and SDN deployments over the past four years[1]. The deployments represent a spectrum: the first ever laboratory deployment to one that carries both research and production traffic to deployments across the globe spanning many campuses and a few national networks. We present our deployments and experiences in four chronological phases and, for each, we highlight goals, state of OpenFlow and SDN components used, infrastructure built, key experiments and demonstrations, lessons learned. We also present how deployments and associated applications influenced the OpenFlow specification, SDN architecture evolution and rapidly growing SDN ecosystem.

The deployments demonstrated the key value proposition of SDN and network slicing or virtualization, and that is, to enable innovation by allowing network owners and researchers to experiment with new networking capabilities at scale in a production setting. Our experience also shows that deployments that can support both research and production traffic play a critical role in maturing of the technology and its ecosystem, especially if the deployments also support compelling demonstrations. Finally experience from these deployments provided valuable insights into various performance and scalability tradeoffs and provided input to the OpenFlow specifications and evolution of the overall SDN architecture. The following is a list of example outcomes and/or insights that the rest of the paper will elaborate.

- For various deployments reported in this paper, a single server of 2010 vin-

[1]Many people contributed to various aspects of OpenFlow and SDN over the years that our deployments used and relied on. The deployments would not have been possible without the help and contribution of these people. Section on Acknowledgements provides a more complete list and details.

tage was sufficient to host the SDN control plane, which includes the network OS and a set of network control and management applications. This confirmed the insight from the earlier Ethane [5] trial at Stanford that a modern server can provide the necessary performance for network control and management of the whole campus.

- The flow setup time quickly emerged as an important performance metric and the CPU subsystem within a switch or router responsible for control traffic is the determining factor until vendors can roll out products with higher performance CPU subsystems.

- The larger national deployment as well as our production deployment in Stanford CIS building confirmed that the number of flow table entries supported by most commercial switches is a limitation in the short term. Future OpenFlow switches would have to support much larger number of flow table entries in hardware for performance and scale.

- Our deployment experience combined with various demonstrations and applications provided valuable input to OpenFlow specification process in the area of failover, topology discovery, emergency flow cache, barrier command for synchronization, selective flow table entry expiration, port enumeration, bandwidth limits and isolated queues for slices, query of individual port statistics, flow cookies, flow duration resolution, and overall clarification of many aspects of OpenFlow specification.

- Our deployments also helped build a preliminary version of SDN monitoring and troubleshooting infrastructure, which we used to debug many problems over the years. For this debugging, we used the following two key architectural features of SDN: 1) OpenFlow switch behavior is essentially decided by its flow table, 2) SDN provides a central vantage point with global visibility. This among other insights is showing a way to building a SDN-based automated troubleshooting framework, which would represent a huge step forward for networking [43, 24, 21, 17].

- Most of early demonstrations and applications built on NOX [28] would build their own network map and then implement their applications functions as functions on the network map. This pattern suggests that the network OS itself should offer network map as the abstraction to applications which represents an important step forward in SDN architecture evolution [40].

The early deployments we report in this paper, as well as other SDN developments, showcase much of SDN's potential. However, SDN will require sustained

research and more diverse deployments to achieve the full potential. This paper simply snapshots the first few years of deployments and by no means represents a final word.

The paper is organized as follows [2]: First, in Section 2, we provide the context that led to OpenFlow and SDN and our plans to take on a trial deployment. We, then, present four phases of deployments: proof-of-concept (Section 3); slicing for research and production (Section 4); end-to-end, that is, campus to backbone to campus (Section 5); and production use (Section 6). We then present how the deployments influenced OpenFlow specifications (Section 7), evolution of the SDN architecture and components (Section 8) and led to a measurement and debugging system (Section 9). We share our concluding remarks in Section 10.

## 2. Origins and Background

OpenFlow and the larger architecture, later named SDN, were conceived in late 2007 and owe inheritance to SANE [6], Ethane [5], and 4D [15] projects. These earlier projects had argued for separation of data and control planes and logical centralization of the control and management plane. At the time Ethane had also demonstrated how the approach can be successfully applied to access control in a campus enterprise network with sophisticated policies. The SDN architecture generalizes Ethane beyond access control and also enables innovation by lifting the entire control plane out of individual routers/switches and implemented as a centralized control plane in a network OS, that constructs and presents a logical map of the entire network to services or network control applications on top[3].

Early deployment plans of SDN were influenced by Ethane trial at Stanford University and the GENI design [16]. The Ethane trial showed among other things that experiment with a campus production network has many advantages. Researchers can more easily use their own campus network and can opt-in their friends as real users for experiments leading to more credible results and more impact. Early GENI design and planning argued for a sliceable and programmable network and computing servers to enable multiple concurrent experiments on the same physical infrastructure based on successful Planetlab experience [37]. Therefore we set the following goals for our SDN deployments:

[2] This paper does not provide a HOWTO for deploying OpenFlow. Please refer to [35] and [9] for an in-depth tutorial on using OpenFlow and for a tutorial on deploying OpenFlow in a production network.

[3] Any reference to "application" in the rest of the paper corresponds to *network control and management applications* implemented on an SDN controller. They represent the network service and functions, and not computer applications

- Demonstrate SDN architecture generality and its ability to enable innovation
- Enable at-scale experiment with campus production networks
- Enable multiple concurrent experiments using slicing and virtualization on the same physical SDN infrastructure

NSF's GENI [12] and SDN have enjoyed a synergistic relationship. The GENI initiative has been funding and enabling SDN deployments at Stanford and around the country. On the other hand, SDN deployments form a major part of GENI's networking substrate and thus, they enable GENI in a big way. While the goals were synergistic with GENI, they also represented significant departure or extensions to early GENI in which the design was more focused on offering researchers a slice of virtualized servers (a natural next step beyond PlanetLab) and did not have a clear and compelling approach to creating a slice of programmable virtual network. Furthermore, early GENI was primarily about a set of servers interconnected by a wide-area network and surprisingly ignored campus networks that researchers use everyday.

SDN deployment for research as well as production use meant we required vendor-supported commercial OpenFlow switches and robust SDN controllers that serve as the "network operating system." This early reliance on and collaboration with vendors proved invaluable as would be evident later in the paper. The synergy with GENI led us to propose *Enterprise-GENI* as a SDN deployment with slicing and programmability on our campus for research and production use. We further proposed to create an Enterprise-GENI kit to help other campuses replicate it in subsequent years and this helped us with SDN growth especially in research and education community[4].

### *2.1. SDN Reference Design*

Fig. 1 shows an early SDN reference design wherein the data plane and the control plane are decoupled. The data plane (comprised of the simplified switches) is modeled or abstracted as 1) a set of flow table entries that decide how incoming packets matching one or more flow table entries are forwarded, and 2) an embedded control processor that interfaces to the control plane and manages the flow table on the switch. The control plane uses the OpenFlow protocol to program the data plane and learn about the date plane state. The control plane is implemented using an OpenFlow controller or a network OS and a set of applications on top. In

[4]Chip Elliott, Director of GPO, encouraged us to think about kits for other campuses.

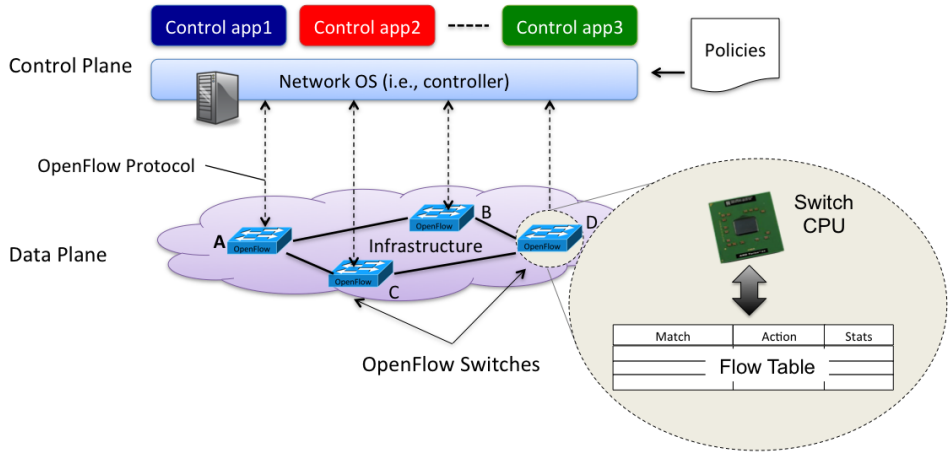



Figure 1: Early SDN Reference Design

our design, the controller takes as input the network policies that dictate what subset of the resources are controlled by which control application. This subsection describes basics of SDN as background for the rest of the paper.

**Flow processing logic.** When a packet arrives at a switch, the switch inspects if there is a flow entry (also called as *rule* in the rest of the paper) in its flow table that matches to the header fields of the packet. If there is a match, the flow table entry specifies how to forward the packet. If there is no match, the switch generates a *packet_in* message that is sent using the OpenFlow protocol to the controller. The controller passes the packet_in event to the appropriate control application(s) based on the policies programmed – which applications see what events. The applications process the event using the network state and their application logic and may send back a message with actions to undertake; the action can involve a *packet_out* (sending out the packet that caused the packet_in), or a *flow_mod* (adding or modifying a flow entry), or both.

**Control plane operation.** The control plane can be *reactive* in handling new flows only after they arrive or *proactive* in inserting flow entries even before the flow arrives, or use a combination of the approaches based on specific flows. OpenFlow protocol allows a controller to query the switch through *stats_request* message and obtain greater visibility into the traffic characteristics. The reactive mode, stats_request, and topology discovery algorithms can together provide the control plane with unprecedented visibility and control over the network dynamics.

**Topology discovery.** The control plane can behave in a centralized manner and manage a large collection of inter-connected switches that make up an *OpenFlow island*. In such a deployment, the controller can use the packet_out message to instruct the data plane to send out LLDP packets from switch ports to network links (Note that an incoming LLDP packet at a switch port will be forwarded to the controller as a packet_in message). Based on these LLDP packet_ins, the control plane can infer the connectivity within the island and construct the network topology. This is the popular approach where no additional support is needed from the switches in inferring the topology of the network.

### *2.2. Performance and Scalability*

Though the key attributes of SDN architecture offer many important advantages, they obviously have some tradeoffs. For example, separation of data and control plane suggests performance penalty in terms of additional delay for control operations especially for flow setup, topology discovery, and failure recovery and (logical) centralization of control plane suggests scalability and single point of failure concerns. One of our goals with deployments has been to study these performance and scalability implications and we share our experiences and lessons learned in this regard.

## 3. Phase 1: Proof-of-Concept

We had simple, yet ambitious, goals for our proof-of-concept phase that lasted most of year 2008. We wanted to create a small SDN deployment using the first prototypes of OpenFlow-enabled switches and controller and build a couple of demonstrations and applications to show the potential of SDN. Our secondary goals included providing useful input to OpenFlow specification process and to commercial vendors offering OpenFlow-enabled switches, and getting more researchers, network operators, and vendors to explore and deploy SDN.

One of the key building blocks for this phase of deployment was the first OpenFlow reference implementation publicly released under the OpenFlow license in Spring 2008. This release enabled early vendors such as HP, NEC, Cisco, and Juniper to build the first set of OpenFlow-enabled hardware switches based on their commercial products with OpenFlow feature for only research and experiments. The other key component was NOX [28] – the only controller available at the time. NOX version 0.4 provided a basic platform to parse the OpenFlow messages and generate events to be handled by different application modules. This version also had the following control applications pre-bundled in the distribution: 1) shortest-path L2 routing, 2) L2 MAC-learning, 3) LLDP-based topology discovery, and 4) switch statistics collection. This served as an ideal platform for experimenters.

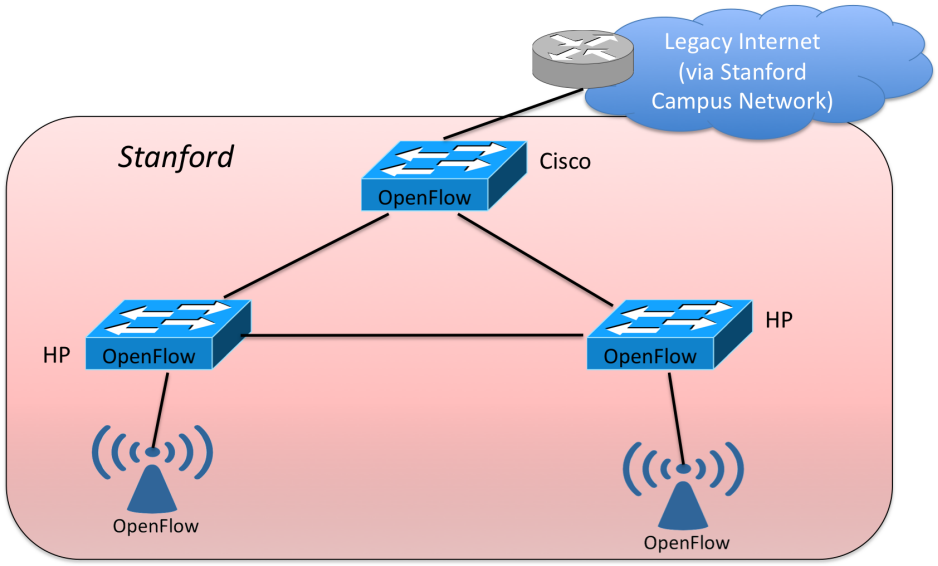



Figure 2: Phase 1 Infrastructure (used for ACM SIGCOMM 2008 demo)

In this section we present the infrastructure built, demonstrations enabled, and key lessons learned in this proof-of-concept phase of deployment.

### *3.1. Infrastructure built*

We started with a small SDN/OpenFlow network testbed in our laboratory at Stanford University as shown in Fig. 2. The network consisted of prototype OpenFlow switches from HP and Cisco, along with 2 WiFi access points that ran OpenFlow reference software switch developed in our research group. HP and later NEC implemented OpenFlow within the context of a VLAN, i.e., OpenFlow traffic was assigned to pre-specified VLANs and managed by the OpenFlow controller, while all legacy traffic was assigned to other VLANs. We refer to this approach of having OpenFlow VLANs and legacy VLANs co-exist on a switch as a *hybrid model*.[5] This proved to be an important feature in the long run for deploying OpenFlow alongside legacy networks on the same physical infrastructure. Each OpenFlow switch was associated with NOX. The controller had additional modules included based on the exact experiment undertaken in the control plane.

[5]At this time, we did not have a more general solution for slicing and virtualization of SDN networks.

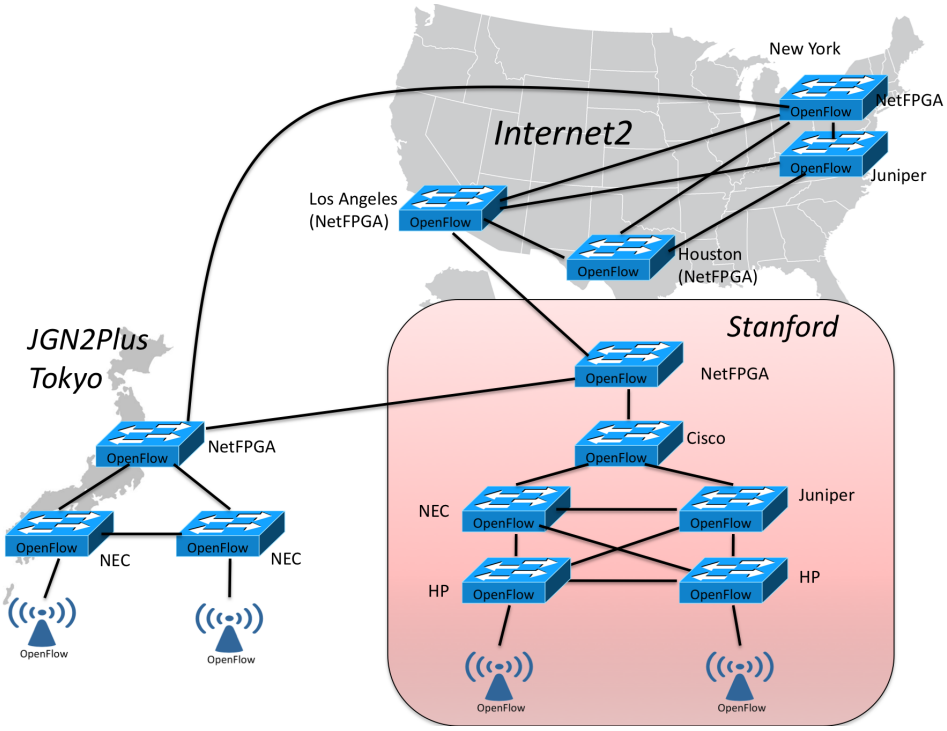



Figure 3: Extended Phase 1 Infrastructure

Phase 1 infrastructure grew quickly as shown in Figure 3. The Stanford team reprogrammed a group of three NetFPGA [26] boxes deployed in Internet2 [19] PoPs to be OpenFlow switches and also helped deploy Juniper's OpenFlow-enabled MX switch to create a small SDN island within the Internet2 network. Japan's JGN2plus [20] also deployed OpenFlow-enabled NEC switches and a NetFPGA-based OpenFlow switch in its own network to create another SDN island in Japan. We interconnected these OpenFlow islands by point to point tunnels as shown in Figure 3 to create an OpenFlow network with a small global footprint spanning Stanford, Internet2, and JGN2plus relatively quickly for research and experimentation.

### *3.2. Key Demonstrations*

As with any new technology, it is important to demonstrate innovative applications. In the case of SDN this meant demonstrating something that is difficult to realize without SDN and would show its overall power. We chose to implement seamless mobility of VMs across L2 and L3 network boundaries using SDN's two key features: Layer-independence of the OpenFlow wherein a flow can be defined

using a combination of L2-L4 headers; and OpenFlow controller's visibility of the global network map. This means when a VM moves, the controller can reroute all its flows to its new location irrespective of its new L2 and L3 location. We demonstrated the VM mobility capability for a multiplayer game application with mobile VMs to support mobile players without ever interrupting the game. The application was highly visual and it showcased VM mobility supported by SDN in a very compelling way. We were able to provide least latency between the gaming clients and the backend server, either by routing the gaming traffic through routes with least hop count or by moving the VM hosting the game server closer to the client. This functionality was built using Python scripting within the NOX controller. We demonstrated mobility of VMs within Stanford (Figure 2) at ACM SIGCOMM 2008 (the best demo award) and across the wide area network between Stanford and Japan (Figure 3) at GEC-3 (GENI Engineering Conference 3) and Supercomputing 2008 [8]. More importantly the SDN part of development was trivial on the controller – a small and simple piece of code was needed to reroute flows when a VM moved and that showed the power of SDN.

Stanford team did another visually compelling demonstration on this infrastructure that included a Graphical User Interface (GUI) to manage all the traffic in a network, even to the extent of enabling a network operator to drag-and-drop flows in arbitrary ways; e.g., the operator can create a flow for fun or demonstration that starts from a client at Stanford, then goes to Internet2 New York, then to Japan, and ends at a Stanford server, and then arbitrarily change the flow's route by doing drag-and-drop.

The first set of demonstrations convinced us and many others of the power of SDN, especially in enabling innovation. This was the beginning of many organizations wanting to build SDNs for their own research.

### *3.3. Lessons Learned*

We summarize a number of important lessons from this phase of deployment. Many seem obvious, especially in retrospect. Still we had to learn them ourselves and they served us well in subsequent years.

**Performance and SDN Design.**

- Phase 1 infrastructure exposed *flow setup time* as a very important performance metric at least for the short term. We define *flow setup time* as the time from when the (first) packet (of a flow) arrives at the switch to when the action of forwarding that packet is completed. In reactive control mode, which is typically used in most academic experiments, this indicates the du-

ration each flow needs to wait in an OpenFlow network before traffic can start flowing[6]. This can be a significant overhead for short-lived flows.

- We also learned that most current switches have lower performance CPUs for control operations. Since these switches were designed for MAC-learning or destination-IP based forwarding, their embedded control CPU does not have enough performance to handle a large number of OpenFlow messages. The stats_requests and reactive mode of flow setup stress the CPU, leading to increased flow setup times, overall poor response time to control plane messages and at times to network outages. In the short term, the limitation can be mitigated by enforcing a limit on the messages sent over the OpenFlow control channel, aggregating rules, or proactively inserting rules. In the long term, we expect OpenFlow switch vendors to use higher performance CPU and fine tune the performance for the OpenFlow use cases – relatively easy for vendors to do going forward.

**Living with Legacy Networks.** In the real world abound with legacy networks, it is not possible to have OpenFlow-only network or a contiguous OpenFlow network. We had to find ways to make OpenFlow and legacy networks interwork and the following are some example issues to be addressed and solutions that worked for us.

- We demonstrated usefulness of the hybrid switch to support both legacy and OpenFlow traffic on the same switch, which has now become the most common model for OpenFlow switches. In the mid to long term we hope switch vendors would develop and network operators deploy OpenFlow-only switches that provide all the necessary functionality for a production network and thus realize all the value of OpenFlow and SDN including simpler forwarding plane and lower capex.

- We used tunneling to interconnect the individual OpenFlow islands. Using NetFPGA to encapsulate the dataplane traffic as part of tunneling can at times cause the packet size to exceed the MTU at an intermediate router, thereby leading to packet drop. We initially resolved this by tweaking the MTU value at the end servers. We later resolved this by creating a software-based tunneling mechanism which provided Ethernet-over-IP encapsulation (using the *Capsulator* software [4]) and used IP fragmentation to circum-

[6]Recall that in the case of proactive mode of operation, the rule is already programmed in the flow table and thus the incoming packet does not have to wait.

vent the MTU issue. Note that software-based tunneling, however, incurs a performance penalty.

- For topology discovery of a network that transits non-OpenFlow (i.e., legacy) switches, the controller has to beware of the relatively common possibility that the intermediate legacy switches may drop LLDP packets used for discovering links in the OpenFlow domain. This can be mitigated by using non-standard LLDP destination address within the OpenFlow domain.

- When we used point-to-point VLANs (in Internet2) to create a virtual link between OpenFlow islands, we had to take caution that the overlay routing performed by the control application does not cause a packet to traverse the same L2 non-OpenFlow link in different direction; such traversing can cause the L2 learning table of the intermediate non-OpenFlow switch to flap. If the traffic volume is high, the flapping can cause CPU overload and consequently lead to a network outage. This can be resolved in some legacy switches by turning off the MAC-learning feature, and in others by re-architecting the topology.

- We noticed that the TCP Nagle algorithm, that is enabled by default, interferes with the message exchange between the OpenFlow switch and controller, thereby creating instances when the flow setup time was large. We resolved this issue by disabling Nagle's algorithm (using the TCP socket option) for the OpenFlow control channel because OpenFlow messages are typically only several bytes long and are time critical, and it is not worthwhile to queue packets until the maximum TCP segment size is reached.

## 4. Phase 2: Slicing and Scaling SDN Deployment

With the SDN proof of concept and its potential demonstrated, we decided to grow the deployment and add slicing to it with the following goals:

1. achieve concurrent experiments over the same physical infrastructure;

2. coexist with production traffic so as to validate the hypothesis that experimentation can occur in everyday networks;

3. evaluate performance of SDN architecture and components in a real deployment setting; and

4. improve the software stack and tools available for SDN deployment.

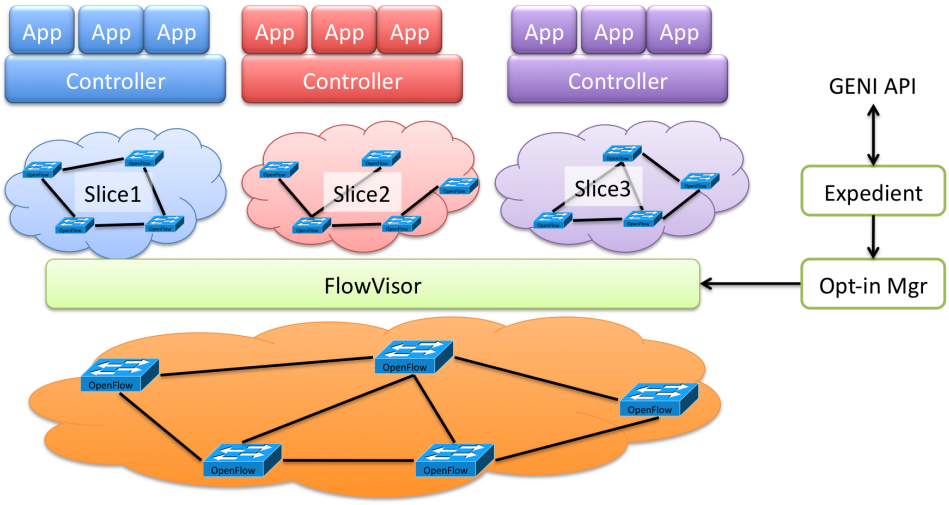



Figure 4: Slicing-enabled SDN reference design

To achieve the above goals, we expanded the topology, added a virtualization layer, helped create and run multiple concurrent experiments, and developed orchestration and debugging tools for better analysis and manageability.

This phase of deployment used OpenFlow version 0.8.9, released in December 2008, that included additional features such as: 1) vendor extensions to allow vendors to define new actions that are not necessarily standardized, 2) hard timeout for flow rules in the cache, 3) allowing replacement actions of previously inserted flow rules, 4) providing match capability for ICMP Type and Code fields, and many other minor updates to the existing actions, messages and protocol formats. The Stanford team again built a reference implementation of OpenFlow version 0.8.9 and that helped vendors quickly release new switch firmware and allowed for a more stable OpenFlow-enabled switch ready for the next level of SDN deployments and experimentation.

One of the main requirements for GENI and for our own deployment has been to support multiple concurrent experiments as well as production traffic on the same physical infrastructure. This is achieved by introducing a network virtualization layer in the control plane that creates virtual networks called *slices* and each slice is typically managed by a different OpenFlow controller. The *slicing* implies that actions in one slice do not affect other slices. We implemented SDN network slicing using FlowVisor [41], a special purpose OpenFlow controller that acts as a semi-transparent proxy between OpenFlow switches and multiple OpenFlow controllers as shown in Figure 4. The FlowVisor isolats the slices based on *flowspace*,

which is a collection of rules that identify the subset of flows managed by a particular slice. With FlowVisor inserted in the middle, all control communication from the switch is classified based on the flowspace, and forwarded to the appropriate controller. In the reverse direction, any message from the controller is validated based on flowspace ownership rules and then forwarded to the switches. With slicing and FlowVisor, we can assign all production traffic to a slice and each experiment to a separate slice. Our deployment in this phase used FlowVisor version 0.3 initially, and then upgraded to version 0.4. In these early versions of FlowVisor, the slice specification and policies governing slice arbitration were specified manually, and remained static through the complete run.

The rest of the SDN software stack included two types of controllers: NOX version 0.4 for experimentation/research, and SNAC [42] version 0.4 for production OpenFlow traffic. SNAC is a special purpose open-source controller based on NOX that Nicira [27] released to make it simple for network administrators to specify access control policies for production OpenFlow traffic (We defer the discussion of production traffic to Section 6). Between Phase 1 and 2, the controllers were upgraded to include support for OpenFlow version 0.8.9, as well as fixes to various bugs.

The rest of the section presents the infrastructure built, key demonstrations enabled, and lessons learned.

### *4.1. Infrastructure built*

We expanded our Stanford deployment from the laboratory to within the basement and third floor wiring closets of Gates building using NEC, HP and NetFPGA switches running the new firmware for OpenFlow version 0.8.9. The infrastructure had 4 NEC, 2 NetFPGA and 1 HP OpenFlow-enabled switches. We supported both production network and experimentation network on this infrastructure. Fig. 5 illustrates the interconnection of the network that we deployed. With NEC and HP switches, we used VLANs to create multiple instances of each OpenFlow switch. That is, with one physical switch, we can create multiple VLANs and associate each VLAN with a controller such that the controller thinks it is connected to multiple distinct switches. This is an easy way to emulate a much larger topology than the physical one. All experiments and production traffic used this logical network.

### *4.2. Key Demonstrations*

The most important capability we wanted to demonstrate was how SDN infrastructure with slicing can support multiple concurrent slices where a slice can support a research experiment, carry production OpenFlow traffic, or carry legacy production traffic. Furthermore, within a slice, SDN retained its core value proposition of making it easy to write innovative applications on top of a controller. At

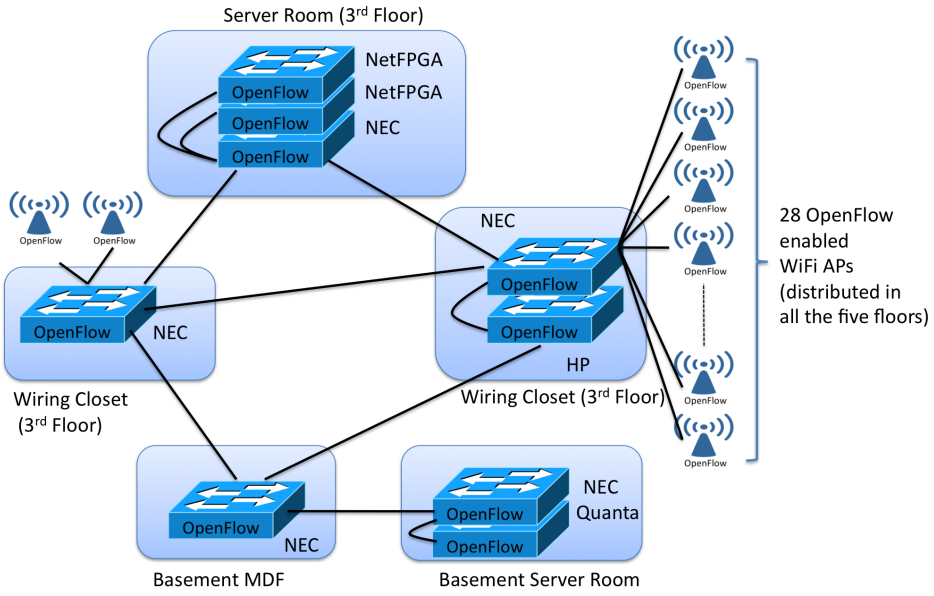



Figure 5: Phase 2 Infrastructure (logical view) in Stanford Gates building

SIGCOMM 2009 conference in Barcelona, as well as, at several other venues, we demonstrated how the infrastructure can be virtualized using FlowVisor to support isolated slices, for example, the following set, each managed by a different NOX-based controller [39]:

- *Plug-N-Serve* research project explored the hypothesis that web load-balancing is nothing but smart routing. That is, web requests can be load-balanced across various servers by a smart routing mechanism within the network by routing each request to an appropriate server taking into account load on each server and congestion on various paths leading to the server. The Plug-N-Serve experiment's slice tested various algorithms for load-balancing web requests in unstructured networks [25]. In this experiment, web queries arriving at a configurable rate are load-balanced by the controller (using its knowledge of the global network state) taking into account both path characteristics and server load. This slice used the flowspace comprised of all HTTP traffic that uses TCP port number 80.

- *OpenRoads* research project explored the hypothesis that we can create a much better user experience on a mobile device if it can utilize multiple cellular and WiFi networks around it under user/application control. The project used a slice to experiment with and demonstrate loss-less handover between the WiMAX and WiFI wireless nodes [44] for a demanding video streaming application. The OpenRoads slice took control of all traffic generated to or from the WiFi access points.

- *Aggregation* experiment showcased OpenFlow's ability to define a flow in a very flexible and dynamic way based on a combination of L2, L3, and L4 headers. Furthermore, a set of flows can be easily aggregated as a single flow by reprogramming a flow definition in a table as sub-flows travel deeper into the network. The specific demonstration shows how hundreds of TCP flows are "bundled" into a single flow by re-programming a flow table entry as component flows traverse the network. The aggregation experiment's slice policy was defined based on the MAC address of the hosts dedicated to this experiment.

- *OpenPipes* demonstrated how hardware designs can be partitions over a physical network. In this experiment, the traffic consisting of video frames were encapsulated in raw ethernet packets and were piped through various video filters running on nodes distributed across the network [11]. Furthermore, the individual functions can be dynamically moved around in the network. The slice policy was defined based on the ethertype specified in the L2 header.

- *Production* slice that carries all non-experimental traffic and provides the default routing for all users' production traffic.

As in the case of our Phase 1 demonstrations, all research experiments in different slices required minimal effort to develop. The SDN part was a simple software application that essentially created the network map in a local data structure, manipulated it for its control logic, and sent flow update events to the NOX. This further validated power of SDN in enabling innovation and making it easy to create new network capabilities. Successful demonstration of OpenFlow/SDN slicing also proved viability of SDN to serve as the networking substrate for GENI.

We built on our Phase one's lesson of creating visually compelling demonstrations to show SDN value proposition and got very good results. Our demonstration at SIGCOMM 2009 won the best demonstration award and more people in the research and industry got interested in exploring SDN — the real value of a good demonstration.

### *4.3. Lessons Learned*

We learned the following lessons through our deployment and experimentation:

- Slicing based on flowspace provides sufficient flexibility to delegate control to different experiments or administrative entities. Our experiments further ratified the need for the flowspace concept.

- FlowVisor version 0.3 initially did not provide sufficient isolation of the control plane workload across the different control applications. Thus, any increase in flow ingress rate for a particular experiment, say Plug-N-Serve, caused poor response to flows in the flowspace of a different experiment, say Aggregation. This was addressed in the next release of FlowVisor – version 0.4 – by adding a software rate limit per slice. Despite this feature, it is tricky to enforce the data plane and control plane isolation across slices, especially when the number of slices exceeds the number of data plane queues available.

- Overlapping flowspace is a tricky concept that is poorly understood. The FlowVisor (both version 0.4 and 0.6) allowed overlaps in flowspace that belong to each slice. This can be a feature in some cases, and a pitfall in others. FlowVisor version 0.6 provided the possibility of associating priority to slices, so as to specify which controller gets precedence in receiving switch events pertaining to a particular flow. This feature, however, cannot prevent two controllers that own the same flowspace from performing conflictive actions. This is a caveat we try to share with all experimenters and still they end up misusing it and thus get into trouble.

- We obtained a deeper understanding of the performance implications of using SDN: The key metric we tracked was the flow setup time. Our experience suggests 5-10ms flow setup time for a LAN environment is typical and acceptable. However our Phase 2 deployments showed flow setup time can be at times greater than 100ms and that is too high to have good user experience. Our in-depth analysis revealed that higher flow setup time is often due to the high load in the CPU subsystem at the switch, when the SDN deployment operates in reactive mode. The true reason for this issue, however, is that most switch embedded CPU do not have enough power to handle increased load due to OpenFlow.

## 5. Phase 3: End-to-end deployment with a national footprint

Our Phase 2 deployment and the demonstrations led to significant interest from other universities especially in US and Europe wanting to deploy and experiment with SDN on their campuses. We identified eight universities in US (Clemson University, Georgia Institute of Technology, Indiana University, Princeton University, Rutger University, Stanford University, University of Washington, University of Wisconsin) that had strong SDN research interest and network administrators keen to deploy and experiment with SDN. Both national research and education backbone networks – Internet2 and National Lambda Rail (NLR) – also agreed to build a small SDN deployment to interconnect SDN islands at these 8 campuses to create a nation-wide SDN to be used as a networking substrate for GENI. NSF/GPO (GENI Project Office) agreed to fund this planned build out, and as part of it also create an OpenFlow deployment at GPO. Soon after ACM SIGCOMM 2009 we also worked with T-Labs (Deutsche Telekom) and a set of universities in Europe to help them organize a similar project called OFELIA with the goal of deploying an end-to-end SDN infrastructure in Europe. These collaborative projects and increasing interest in SDN led us to the following goals for the next phase of SDN deployments.

- Help campuses, GPO, NLR, and I2 deploy SDN without requiring heroic effort.
- Create a functioning end-to-end SDN infrastructure with slicing and with GENI interface so multiple concurrent slices can be hosted on the same physical infrastructure for experimentation.
- Help researchers on campuses to do experimentation and demonstration of their ideas on the local and national SDN infrastructure.

- Help mature OpenFlow and SDN components.
- Help grow the SDN ecosystem.

By the time we were ready to do serious deployments for this phase, Stanford team released OpenFlow version 1.0 and that became the default for all deployments. The main feature added was queue-based slicing, which is a simple QoS mechanism to isolate traffic in OpenFlow networks. Other features added include matching IP addresses in ARP packets, flow cookies to identify flows, selective port statistics, and matching on the ToS bits in the IP header.

Fig. 4 shows the complete SDN stack that was deployed in this phase. The stack comprised of OpenFlow-enabled switches running version 1.0 firmware, FlowVisor version 0.6 for slicing individual OpenFlow islands, and NOX version 0.4 for controller. To scale up experiments and to work with GENI control framework, we added policy management framework which was provided by two additional components, Expedient and Opt-in Manager, as follows:

- Expedient [10] is a centralized control framework designed for GENI like facilities with diverse resources – it centralizes the management of slices and experiments, as well as authentication and authorization of users. It is designed to support each resource as a pluggable component of the framework and each resource to have its own API for resource allocation and control. This simplifies addition of new resources into the framework and their management as each resource can have its own API. For our deployments, Expedient supported PlanetLab API, GENI API and Opt-in Manager API for OpenFlow networks. Expedient is implemented as a traditional three-tier web server (with a relational database as the backend).
- Opt-in Manager [36] is a database and web user interface (UI) that holds information about the flowspace that each user owns and the list of experiments that are running in a network along with the flowspaces that they are interested in controlling. The web UI allows users to opt their traffic into individual experiments. When a user opts into an experiment, the Opt-In manager finds the intersection of that user's flowspace and the experiment's flowspace and pushes it down to flowvisor causing the packets matching the intersection to be controlled by the requested experiment's controller. We used version 0.2.4.2.

As we started to support other campuses and groups, we created several recipes for SDN deployments and posted them on our deployment website [9]. These recipes allowed a group to quickly create an SDN network in software and gain

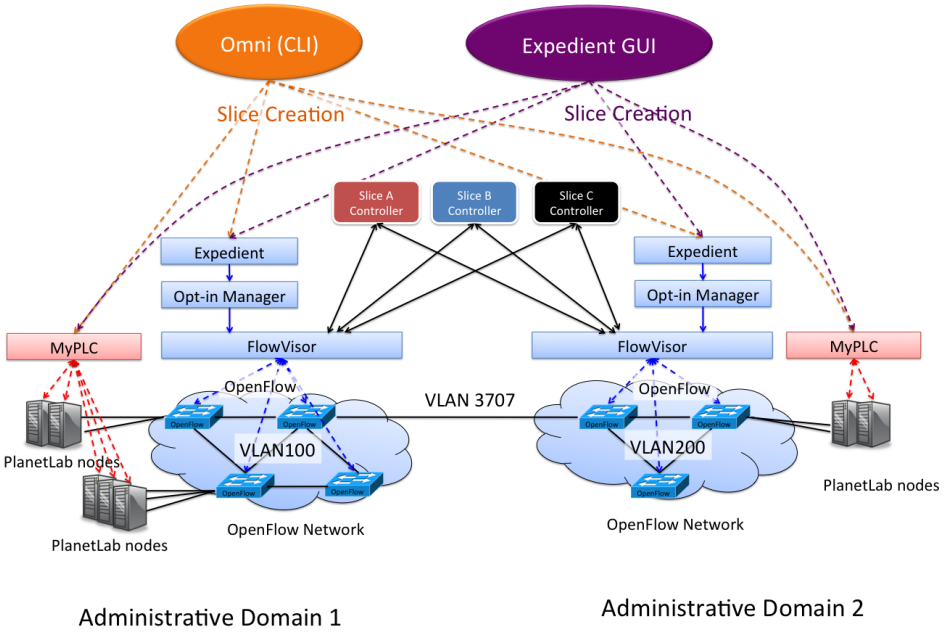



Figure 6: Wide-area SDN network showing the different software components at each island. A slice request from an experiment is processed by individual island components to form a large-scale stitched slice (with networking and computing).

useful experience; deploy a small SDN in a lab; or deploy a network in a building for research and production use with real hardware switches and SDN software stack. We also tried to create processes for the release and support of SDN components such as NOX, SNAC, FlowVisor, and others[7]. These processes needed to be revised often as we were learning and OpenFlow/SDN components and the ecosystem was rapidly evolving. Though we were not able to perfect the processes, they helped us and others keep track of SDN components, receive support and plan for orderly upgrades.

In the rest of the section, we present the end-to-end infrastructure we built with our collaborators, the key demonstrations enabled, and the lessons learned.

### *5.1. Infrastructure built*

Each campus deployed an OpenFlow network as well as a few PlanetLab nodes as compute servers to create a local GENI substrate. Fig. 6 presents an illustration

[7]GPO encouraged us to specify the processes and a release plan and it helped.

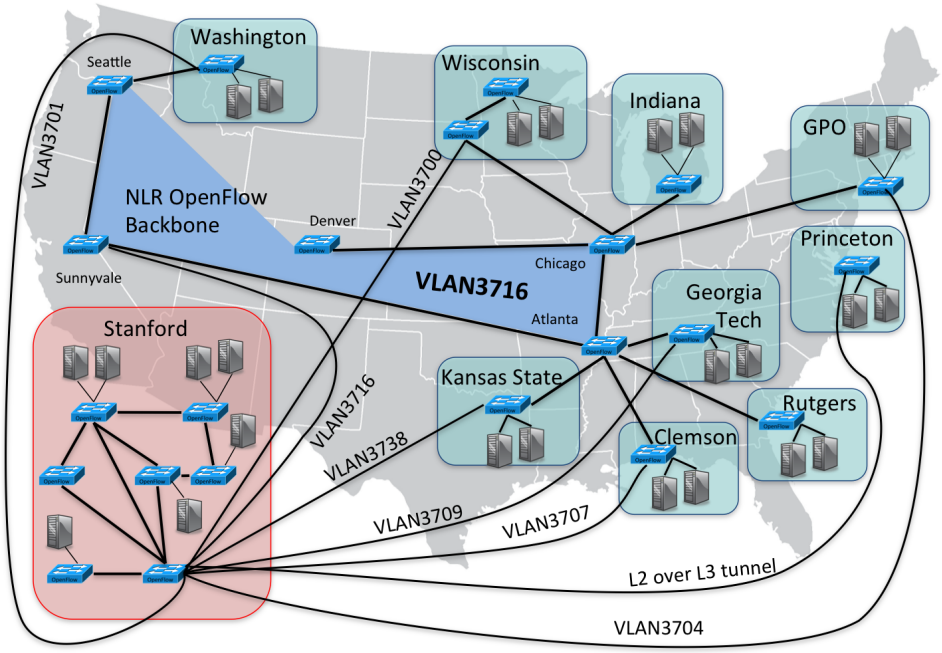



Figure 7: Phase 3 Infrastructure

of how substrates are stitched to form a large-scale slice using SDN stack (FlowVisor, Opt-in Manager, Expedient, Controller), MyPLC for PlanetLab, and GENI control (Expedient, Omni).

Each campus team was responsible for its own SDN deployment, and the Stanford team and GENI project office (GPO) was responsible for the interconnectivity among islands and for much of the testing of the infrastructure. Although each campus deployment had some commonality, it had its own unique aspects as well. For instance, Rutgers deployment included OpenFlow-enabled wired infrastructure for WinLab; Georgia Tech included a deployment that spanned multiple buildings with a plan for deployment in dorms for access control; and Indiana's emphasis was on production deployment and network operation.

The 11 individual islands (i.e., 9 campuses, GPO, and NLR)[8] were interconnected, as shown in Fig. 7, to create a small nation-wide SDN infrastructure in-

[8]Internet2 at the time did not have OpenFlow-enabled switches and provided point-to-point VLANs to interconnect OpenFlow island. Internet2 OpenFlow island was built later [38]

terconnecting virtualizable computing nodes to create a single sliceable and programmable computing and networking facility. Initially we interconnected the different campus SDN deployments, using point-to-point VLANs over the NLR, to the Stanford deployment. For instance, VLAN 3707 was used to interconnect Stanford and Clemson Universities at Layer-2 such that the two islands seem like just one by routing traffic through MAC-level switching. In a similar manner, we created dedicated VLAN links to each deployment and trunked the traffic at our basement NEC switch, thereby creating a star topology for the national OpenFlow network. Once the NLR SDN deployment became ready for use, each campus deployment set up an additional VLAN connection (VLAN 3716) to the NLR deployment. Most experiments started using this NLR network as the backbone (instead of the Stanford deployment), and it continues to be so even today. Since Princeton University does not directly connect to the NLR, we created a software L3-in-L2 tunnel (using the Capsulator) from a server in Stanford network to a server at Princeton University. Note that Rutgers University connected to the NLR island through Internet2.

The interconnection among OpenFlow islands was carefully stitched to ensure that the overall infrastructure looks like a single flat Layer-2 island to the experimenter. In other words, all GENI traffic was carried on a set of VLANs that spanned different segments (viz., campus segment, NLR segment, Internet2 segment) that connected to each other. Although the traffic within each OpenFlow island was processed using flow table entries that include L2-L4 headers as installed by appropriate application controllers (i.e., forwarding within OpenFlow is layer independent and not confined to a single layer such as L2), the interconnects used legacy switches and we had no control over them.

Experimenters using this global OpenFlow island created a slice of their specification (comprised of a list of computing nodes along with the interconnecting network elements) using either a centralized Expedient running at Stanford or through a software tool called *Omni* [32]. Both Expedient and Omni interacted with the Expedient Aggregate Manager at each campus to create the slice. Once the slice is created, the local administrator at each campus inspects the slice requirements within the Opt-in Manager and "opts-in" the local resources to the experiment. Once opted-in, the rules are created in the FlowVisor, thereby mapping the flowspace requested to the experimenter's slice controller. Once a slice is created and flowspace is mapped, the slice (even one that spans all islands around the country) has one controller to manage it.

### *5.2. Key Demonstrations*

We continued our emphasis on demonstrations to showcase SDN potential and to demonstrate new scale and capabilities of the infrastructure. The following

demonstrations came together for the GEC-9 plenary session that highlighted the nation-wide infrastructure and SDN capabilities.

- *Aster*x*: A wide-area experiment, based on the earlier plug-n-serve system [25], to investigate different algorithms for load-aware server and path selection. The infrastructure used spanned servers and clients located at all SDN/OpenFlow islands (except Rutgers University).

- *Pathlet*: This experiment showcased a highly flexible routing architecture that enables policy-compliant, scalable, source-controlled routing [13]. This feature is implemented over the OpenFlow controller. Specifically, the experiment demonstrated how edge devices can effectively use multi-path routing flexibility in the face of network dynamics via end-to-end performance observations.

- *OpenRoads*: This experiment built on the earlier OpenRoads system and involved a live video stream from a golf cart driving around at 5-10 miles an hour. The experiment showed great user experience for live video if the device is allowed to use multiple wireless networks around it and SDN made it easy to implement handoff between APs and WiMAX base station and tri-casting video stream over both WiFi and WiMAX networks [44].

- *SmartRE*: This demonstration showed how the SmartRE framework allows both the service and the network capacities to scale by removing redundant content in transmission [2]. The traffic was generated by a popular on-demand video service and the system used a network-wide coordinated approach to eliminate redundancy in network transfers. The OpenFlow controller and the application made it easy to manage routers with the redundancy elimination feature.

All these demonstrations yet again confirmed that building new applications or capabilities on SDN is relatively easy even if the network spans multiple OpenFlow islands around the country. All the demonstrations worked flawlessly and GEC-9 was a grand success for demonstrating an operational GENI. However, the underlying SDN infrastructure was not as stable and robust as we would have liked. To achieve more stability and predictability, we proposed a long term experiment that was aimed at creating lots of OpenFlow slices and running some for days and weeks. The GPO team took the lead and defined the Plastic Slices experiments conducted between GEC-9 and GEC-10 [38]. The experiments lasted for several months and helped debug a few problems and make the infrastructure more stable.

At the end of Plastic Slices project, GPO ran 10 GENI slices continuously for multiple months and within each slice, the hypothetical experimenter performed file transfer between pairs of virtual computers in a varying schedule.

*5.3. Lessons Learned*

This was the time when interest in OpenFlow and SDN was growing fast and many groups around the globe wanted to create their own SDN infrastructure for research and experimentation. Our informal non-scientific estimate based on downloads and email traces suggests there were close to 200 SDN deployments by mid 2011 and we supported many in a variety of ways including help and advice on the design of the network; selection of SDN components; debugging of problems with configuration and deployments; and overall hand holding. Working with these groups around the globe provided invaluable experience and lessons that we summarize in the following paragraphs.

**GEC-9 Deployment and Demonstrations.**

- Coordinating and supporting deployments: Creating a single large-scale network by stitching together individual islands is a challenging task for technical and non-technical reasons. Most OpenFlow islands are part of CS and EE departments and they needed to be connected to the campus backbone and then to I2 and NLR backbones and this is nontrivial. It involved creating a complete inventory of available network resources, creating figures with network topologies, identifying VLANs available end to end, allocating the VLAN numbers, verifying the connectivity, and coordination with tens of people. GPO did most of this ground work and without that the nation-wide infrastructure would not have come together.

  Our group at Stanford mostly coordinated with the network researchers and administrators directly involved in SDN deployments. Still the coordination was an balancing act in that we provided SDN tools, support, a blue print for deployment and guidance to each campus, and at the same time, we wanted each campus to have enough flexibility to be creative and serve its own research and operation goals. We had regular bi-weekly phone meetings with each campus for close to two years to coordinate the deployments and that worked well for everyone involved.

  When it came to updating and maintaining latest software on all campuses, each campus selected a different approach. In some cases, the campuses trusted us (i.e., the deployment team at Stanford University) to configure components such as the MyPLC and the Expedient/Opt-in Manager and upgrade them as necessary. In other cases each campus assigned a local person. The flexibility and trust on part of individuals were key to success.

- Path to maturity: We were either too naive or too bold to sign up for several demonstrations at the GEC-9 plenary session that used the GENI national infrastructure with 11 independent OpenFlow/GENI islands with prototype OpenFlow-enabled switches from different vendors and several prototype software systems that were not tested together. It was shear hard work and some luck that made it all work but the underlying infrastructure was not as stable and predictable. In retrospect, we should have done the Plastic Slices project first to debug the issues and increase the stability. A more stable infrastructure would have made lives of the demonstrators easier and less stressful.

- SDN Tools Development, Deployment, and Support: Our software release process was complicated because we wanted to ensure that all islands ran the same version of the software stack and three key software components (FlowVisor, Expedient, and Opt-in manager) were being released for the first time. These components were undergoing active development, debugging and deployment at the same time – not recommended. This required too much of coordination among developers, deployment teams, and campus folks. It worked out for GEC-9. However our experience suggests the university based developers (students and post docs) and deployment engineers were not well prepared for the challenge and we cannot recommend doing it again. In fact after GEC-9 we decided that we cannot continue development, deployment, and support of SDN tools and platforms from the university. Instead we established a separate non-profit organization called Open Networking Laboratory (ON.Lab) that would be responsible for open source SDN tools and platforms and their deployment and support [33].

**Network Operation.**

- Controller placement: A single server hosting a controller for a nation-wide slice spanning ten islands turned out to be sufficient with regards to the load – the server never became a bottleneck. However, for latency-sensitive experiments (like Aster*x load-balancing), reactive setup of flows in Princeton by a controller in Stanford is sub-optimal. The placement of the controller is, thus, an important consideration for both performance and reliability [18].

- IP-subnet-based flowspace: We learned that we cannot let each experimenter specify her own flowspace as it leads to flowspace overlap and to flowspace explosion. We learned that it was best to associate experiments with a particular internal IP subnet that spanned the wide-area substrate and allow the experimenter to only control traffic within that subnet (e.g., the subnet

10.42.111.0/24 was dedicated to the Aster*x experiment that stretched to all islands) and all compute servers used by Aster*x sent traffic within that subnet, which was managed by Aster*x's controller. This avoids overlap in flowspace across experiments and has been the standard practice since after GEC-9.

- Scaling: This phase of the deployment was the largest at the time in terms of the number of OpenFlow islands, switches deployed, slices supported, flow table entries needed, and users. As expected running over 10 experiments, each having of the order of 1000 rules stressed the software stack and early versions of OpenFlow switches. For example, the FlowVisor would crash often and flow setup time was too high in many cases. It took several iterations for the FlowVisor software to mature and become stable. Similarly the switches were seriously stressed by certain experiments and provided further evidence that their CPU subsystem did not have enough performance for OpenFlow use case. Until switch vendors can provide higher performance CPU subsystem, we used a number of tricks (e.g., enforcing a message rate limit) to keep the the load of switch CPU subsystem below an acceptable threshold.

- Interconnection: Since the world does not have ubiquitous OpenFlow support, we had to cross legacy switches and networks. This can be tricky because of 1) MAC learning in the legacy network where traffic is overlay routed by OpenFlow switches or where a network uses Q-in-Q can cause perpetual learning and CPU overload, 2) topology discovery errors that can potential arise in the controller when an OpenFlow network is connected at two locations to a legacy network. The first issue is also discussed in the lessons of Phase 1. The second issue is a problem because it causes the controller MAC learning to oscillate and be inconsistent with choosing uplinks for the flows.

- Network Loops: Most experimenters wish to have a network topology with loops so as to provide multiple paths to handle their traffic. Having a loop within the OpenFlow network that interconnects with the legacy network is risky because a broadcast storm caused within the OpenFlow network will cause meltdown of even the legacy network. Since the OpenFlow network does not support STP, the legacy network will be unable to identify and block any disruptive loops. In the early GENI network, broadcast storm occurred two times and caused outage in the legacy network: 1) When a loop at BBN was enabled by the experimenter starting their controller before the administrators added safeguards, 2) When a loop across backbones was caused

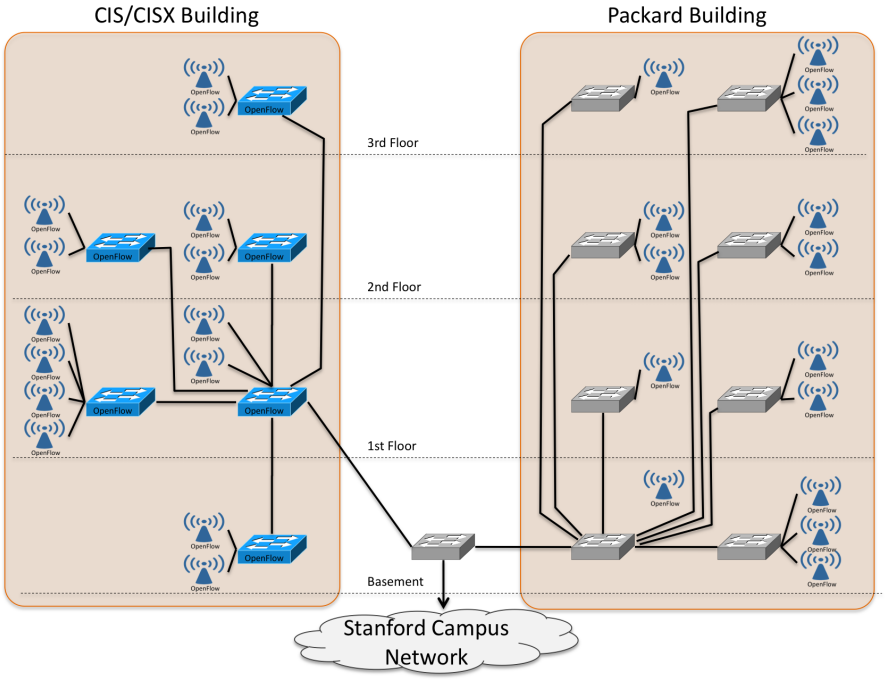



Figure 8: Phase 4 Infrastructure

by connecting a campus over both NLR and Internet2 to Stanford. GENI's *Emergency Stop* is required to recover from such a failure when an experiment becomes disruptive.

## 6. Phase 4: Production Deployment

We had two complementary goals with SDN deployments for production use: bring OpenFlow and SDN as close to production ready (with network vendors and university network administrators) as a university based research group can; and demonstrate that SDN with slicing can support both research and production use. The production phase actually overlapped with Phase 2 and 3 and we demonstrated feasibility of using SDN with slicing for research and production use (for our group) during these phases as we briefly explained in the previous sections.

As much as our network administrators believed in SDN, they did not want to deploy it for production use until: 1) they could see a production SDN network in operation for 3-6 months with consistent results; and 2) they could get SDN

components such as OpenFlow-enabled switches, controller, and a few core applications with support from trusted vendors. To meet these expectations we decided on the following plan;

- Create an OpenFlow WiFi network that anyone registered to Stanford official WiFi can use as an alternative WiFi for "almost production use".
- Create a production SDN infrastructure for our group of approximately 20 people to use.
- Create a comprehensive measurement and debugging infrastructure so we can understand and demonstrate performance and stability of our group's production infrastructure on a continuous basis.
- Work with hardware and software vendors to achieve the goals listed above.

We had an understanding with our network administrators that if we were able to achieve the goals above, they would deploy SDN in their buildings. In the rest of this section we present our experiences with the tasks outlined above and how this led our EE department network administrators to do a limited production deployment in two buildings. Though much of the EE production deployment is still ongoing, we present the current snapshot.

### *6.1. OpenFlow WiFi Network*

We first built an OpenFlow-based WiFi network (called *ofwifi*) in the Gates building and made it available for any registered user of the Stanford official WiFi network. The OpenFlow WiFi network allowed us to offer "a semi-production service" to guests who do not demand the same level of service as regular users.

As part of Phase 2 deployment, we had deployed 31 OpenFlow-enabled WiFi access points (APs) throughout the six floors (including basement) of Gates building [45]. Because we have OpenFlow switches in limited wiring closets (one in the basement and two on the 3rd floor), only four APs had a direct connection to the OpenFlow switches and the rest were connected over a software tunnel to one of our OpenFlow-enabled switches.

We opened up the *ofwifi* network for guests and it quickly attracted many users (approximately 20 users during the busy hour) for their public Internet access. Since users always had the option to switch back to the Stanford WiFi by changing the SSID, extremely high availability was not a critical requirement for the *ofwifi* network. Therefore we used this network as a place to first deploy new releases of SDN components after testing them in the laboratory. As expected the network was flaky at the beginning and would suffer many outages. However by the time

we were done with Phase 3 deployment with more mature SDN components and considerable amount of debugging, the *ofwifi* network became quite stable except during the frequent planned outages to upgrade software. Guests used it without realizing that the network was not meant to be production quality.

### *6.2. Group Production Network*

We, as a group of approximately 20 people spanning 7 rooms in Gates building, decided to use wired SDN for our day-to-day production use with the intent to live the SDN experience ourselves and also use the production network to study and demonstrate performance and stability to our network administrators.

Our production SDN network used NEC, HP, NetFPGA and Pronto (with Indigo firmware and Pica8 firmware) switches with support for OpenFlow version 1.0.0. We initially used SNAC version 0.4 as the controller, and later replaced it with the BigSwitch commercial controller. Two switches in two different wiring closets on the 3rd floor had direct connections to our 7 office rooms and both switches were connected to the aggregation switch in the basement. This infrastructure was built as part of Phase 2 deployment shown in Fig. 5. We deployed various SDN components in this wired production SDN network only after testing them in the lab and in the *ofwifi* network. Thus the wired production network had relatively higher availability.

Both *ofwifi* and this wired group production networks were instrumented and measured on a continuous basis since 2009 as explained in Section 9. We shared highlights of weekly measurement and analysis with our network administrators. On one hand the network administrators helped us debug problems, and on the other, they got to see first hand how the production network has been maturing. The section on measurement and debugging elaborates on example issues or problems we had to deal with before the network became stable and met all performance metrics.

### *6.3. Limited Production Deployment in EE*

After two years of close monitoring of SDN evolution in our group's production network, the network administrators of CIS building started enabling OpenFlow capability in their production network with a firmware upgrade from HP for already deployed switches. This was an important milestone for SDN deployment, both at Stanford and in general.

Fig. 8 illustrates the production deployment in CIS and Packard buildings. We placed 30 OpenFlow capable WiFi access points in each building. At the time of writing this paper, all 6 production switches in CIS building are OpenFlow-enabled, and they are controlled by a single controller together with 30 OpenFlow capable APs. The switches in Packard building are not yet running OpenFlow

and thus the OpenFlow APs in Packard building are not directly connected to the OpenFlow switches, but the commercial controller we use now supports multiple OpenFlow islands and thus we do not need to create tunnels as we did in the *ofwifi* network.

### *6.4. Lessons Learned*

Even for a small production deployment, switch hardware limitations can be an issue. Today's OpenFlow-enabled switches are limited by:

- Flow table size that typically supports a maximum of $\approx$ 1,500 flow table entries, not large enough for a core switch of a production VLAN within a single building. To mitigate this, controller vendors (e.g., BigSwitch) are using tricks such as aggregation of flows through wildcarding of flow-match header fields (e.g., wildcarding the TCP and UDP port numbers). Also some switch vendors have started to augment the TCAM to expand the flow table size by a factor of 100. This proved sufficient for our deployment.

- Low flow ingress rate capability that prevents an OpenFlow-enabled switch from being deployed in the core part of the network. This is circumvented in production deployments by breaking a larger network into multiple OpenFlow islands that are physically connected. Controller vendors are offering controllers that support multiple-island design as a default feature for this reason.

## 7. Input to OpenFlow Specification

Our deployment experience provided important input to the OpenFlow specification process especially early on[9]. As expected deployments and demonstrations exposed a number of ambiguous points and some bugs in the specification and made the case for a number of new features. We summarize highlights of how deployments and demonstrations in part helped shape the OpenFlow specification.

- Flow modification: Allowing quicker (replacement and) modification of existing flow table rules without having to delete and add. This was particularly needed for the handoff of mobile devices from one AP to another and included in OpenFlow version 0.8.9.

[9]The specification process has been receiving input from many sources especially as more vendors and providers got involved in SDN. All of this input shapes OpenFlow specifications and not just our input.

- ICMP code matching: Allowing match of ICMP packets generated by `ping`. This allows the control application to treat the ICMP request differently than the ICMP reply. This was included in OpenFlow version 0.8.9.

- Controller failover: Allowing an OpenFlow network to switch the controller if the primary one were to fail or become unreachable. This has been an important requirement for SDN since beginning and became more evident as we got more experience with deployments. This was included in OpenFlow version 0.9 and added to the reference implementation and picked up by the vendors.

- Emergency flow cache: Our deployment showed the need for managing the dataplane in the event the switches lose connection to the controller(s) altogether. OpenFlow version 0.9 included the possibility of inserting rules that are to be used, instead of the normal rules, when such a control connectivity outage occurs. Some switch vendors implemented this by having two flow tables: normal and emergency.

- Queue-based slicing: Our early deployments showed a broadcast storm in a slice caused disruption to other slices. To provide better isolation in the dataplane, we asked for queue-based slicing of the network, wherein each slice can be mapped to a different queue in the network. This feature was added in OpenFlow version 1.0.

- Experimentation based feedback: We, along with the experimenters who created interesting SDN applications, provided feedback that also helped shape OpenFlow specifications. Following are some of the main features requested and added to OpenFlow version 0.9: 1) *Barrier* command to notify the controller when an OpenFlow message has finished executing on the switch and is useful to sync the state between controller and the switches; 2) Capability to match on the VLAN PCP priority bit field; 3) Sending flow expiration (flow_exp) message even for controller-requested flow deletions; 4) Rewriting IP ToS field with custom value to change prioritization of packets in legacy network; 5) Signaling a switch to check for overlaps when inserting new rules that may possibly conflict with a pre-cached rule. Following are other features requested and added to OpenFlow version 1.0: 1) Ability to specify a *flow cookie* identifier for each flow to enable controller to easily track per-flow state; 2) Ability to match IP fields in the ARP packets; 3) Switch owner specifiable description of an OpenFlow switch thus making it easier for the programmer to use it.

Following are 2 issues, experienced during our deployment, that are yet to be resolved in the OpenFlow specification:

- Inband control: In several deployment scenarios, it is not possible to create out-of-band network for the control plane communication. This was especially the case for the early OpenFlow WiFi deployment, where some early commercial products did not allow OpenFlow VLAN and non OpenFlow VLAN on the same physical port using VLAN tagging. When the only network connectivity available is already part of the dataplane, then the switch needs to provide *inband control*, wherein the switch has to make special classification on whether a packet is part of control or general data. Although we requested this feature, the OpenFlow specification team was not able to achieve a consensus on the actual mechanism and deferred it for later versions.

- Spanning-tree protocol: Most Layer-2 switches in a legacy deployment run the Spanning Tree Protocol (STP) to avoid packet broadcast storms due to loops in the network. Since most OpenFlow-enabled switches (e.g., NEC and HP switches) also have an inbuilt legacy stack, it is tempting for some production deployments to use both STP and OpenFlow, wherein STP will prevent any downsides of the loop and OpenFlow will provide control of all traffic within the safe environment. However enabling STP on an OpenFlow-enabled VLAN can lead to unpredictable outcomes because the OpenFlow stack has no awareness that a port is "blocked" by STP (port status is "up"). Although this issue is mentioned in OpenFlow specification version 1.1, it is yet to be fully addressed by the hybrid working group of the Open Networking Foundation (ONF). Our deployment does not use STP at this point for this and other reasons.

## 8. Input to SDN Architecture and Components

Our deployments and demonstrations also provided useful input to the evolution of the SDN architecture and its components. For example, each demonstration or application on NOX typically first builds a network map of its own and then writes an application logic to work on this map. This insight (among others) led to the realization that it is better for the network operating system to offer the network map as the abstraction as opposed to network events and that is how the SDN architecture is evolving now.

One of the goals of the deployment has been to support multiple concurrent experiments and production traffic on the same physical infrastructure. This led to

the concept of slicing and the design of FlowVisor. There are other business motivation for network virtualization and FlowVisor helps with them but its original design and how it fits within the SDN stack were influenced more by the short term deployment goals. For example, FlowVisor serves as a semi-transparent proxy between the controller and the switches, so as to implement slicing. Any control message sent from a slice controller to the switch is intercepted, and tweaked to ensure that the slice manages only the flowspace it controls. This tweaking process may involve replacing the output ports (outports), rewriting match header fields, and altering the actions performed. In the reverse direction, any control message sent from the switch may be routed to the appropriate controller(s), based on the match headers and the input port (inport). This allows each experimenter to own a slice of the flowspace and slice of the network resources and an end user can opt-in to the flowspace.

FlowVisor was conceived, designed, and prototyped in our group at Stanford and it became a critical component of our deployments. As expected deployments and experimentation helped quickly identify bugs including memory leaks and design limitations of FlowVisor leading to new versions as summarized in the following paragraphs.

- Bugs: We uncovered cases where the FlowVisor translated a FLOOD action from the slice controller into an OUTPUT action among an incorrect list of switch ports [43]; this bug caused the control message to be dropped by the switch. Secondly, we noticed cases when the FlowVisor performed an incorrect demultiplexing of the LLDP packet_ins being received, thereby causing the slice controllers to not see the right topology. Lastly, the FlowVisor did not pay attention to topology or port changes, thereby keeping an incorrect list of active ports; this bug caused incorrect port translation and caused control messages to be dropped by the switch. All these issues were fixed in subsequent release of FlowVisor version 0.6.

- New Features: Based on experimentation, we requested two features: 1) Need for a per-slice monitoring system that tracks the rate and number of control plane messages per slice; this will allow us to verify the slice isolation at real-time; and 2) Need for building safeguards in the FlowVisor to push a default drop rule (for the flowspace) to the switch when the slice's controller is unavailable. They were included in later release of FlowVisor 0.4.

  To make FlowVisor easier to use, we asked for two other features: 1) dynamic rule creation, and 2) better user interface. These issues were addressed in a Java-based FlowVisor version 0.6, thereby making it ready for the GENI

deployment. Secondly, this version featured a new set of *fvctl* commands that operated on the state stored and provided good visibility into the rule set; examples of the commands include *createSlice*, *addFlowSpace*, *deleteFlowSpace*, *deleteSlice*, *getSliceInfo*, *listSlices*. Thirdly, the FlowVisor was built with a new abstraction that made it easier to allow multiple ways to input policies to the FlowVisor and manage its rule set (e.g., A slice can be created both by the FlowVisor *fvctl* commands as well as the Opt-in Manager).

- FlowVisor rule explosion: The flowspace requested by an experimenter is often expanded into a set of individual internal rules based on the ranges of all the fields as well as the list of switches and ports. This cross-product can lead to the number of rules of the order 1,000 for many experiments[10]. The matter is further complicated when two experiments request an overlapping flowspace (e.g., If slice *A* requests *IP=any, TCP port=80* and slice *B* requests *IP=1.1.1.1, TCP port=any*, then the switch will have to programmed even before the packets arrive to prevent conflicts, thereby causing an increase in the switch rule set). In some cases the rule explosion caused the FlowVisor commands to be less responsive because it takes too long to install (or delete) all the expanded rules in (or from) FlowVisor memory. During operation we observed cases where the opt-in step takes as much as few minutes, which is too long for interactive slice creation. Although this does not affect the management of the flows in the data plane, it proved to be a serious bottleneck for creating slices. This issue continues to be an open issue as there is no real fix though there are short term deployment specific optimizations that we and others have been using.

During the course of the nation-wide GENI deployment, we discovered several issues that had to deal with the consistent state across the full vertical stack, i.e., consistent state of the slice among Expedient, Opt-in Manager and FlowVisor. As each software tool/system was developed by a separate team, inconsistencies should not have been a surprise. Examples include 1) a slice was instantiated in the Opt-in Manager, but not in the FlowVisor, 2) a slice was deleted at Expedient, but still available in the FlowVisor, and 3) a slice was opted-in at the Opt-in Manager, but the rules not yet pushed to the FlowVisor. The issues were gradually resolved with deployment and testing and finally verified by the Plastic Slices project after GEC-9.

[10]Note that negation rule was discontinued in FlowVisor version 0.6 because expanding to match all other values consumes too much rule table space, e.g., Matching all traffic with *TCP dst port != 80* may expand to 65534 rules.

Lastly, our deployment helped provide constant feedback to the controller developers. For instance, we observed that the HP switch in our deployment could not be virtualized based on VLAN to create multiple OpenFlow instances from the same physical switch. We discovered the problem to be with the NOX controller version 0.4, which inaccurately was implemented to only retain the most significant 48 bits of the datapathid (i.e., the switch instance identifier). This caused 2 logical OpenFlow instances within a single HP switch to seem like 1 instance. Similarly, we encountered a bug where the controller (both SNAC[42] and NOX) was not accurately detecting links in the presence of a HP switch. After extensive debugging, we uncovered the issue to be that the LLDP packets generated by the controller's discovery module had a multicast address in the source MAC field. The HP switch, however, is pre-programmed to drop such packets, thereby causing some links to not be discovered. Such issues often manifested as switch problems and took significant effort to identify the root case. Based on our feedback, the controller developers resolved the issues in subsequent releases.

## 9. Monitoring and Debugging

As we scaled our SDN deployments, especially production deployments, we had to develop a comprehensive measurement infrastructure to collect, display and chronologically archive the important network operation metrics. We, further, developed approaches to using this data to monitor and debug the network, as well as to share the performance and stability details with others. In this section, we first describe the measurement infrastructure, and then we present approaches we adopted for performance analysis and debugging of SDN deployments.

### *9.1. Measurement Infrastructure and Performance Metrics*

Our objective for the measurement infrastructure was to obtain dataplane measurements to profile the performance as seen by an end-user as well as to identify issues that are not necessarily visible from the SDN control plane. We deployed several measurement probes in the network that actively measure different metrics to help understand the user experience. Fig. 9 illustrates the measurement infrastructure that we built. As shown, we collected three different data sets and stored them in a central repository for analysis: 1) Control channel dump, passively collected using `tcpdump` of the communication between the switch and the controller, 2) End-to-end performance actively measured by the probes, 3) Switch status metrics actively collected from the controller and switches. The following is a list of end-to-end performance metrics that we use for measuring the end user experience.

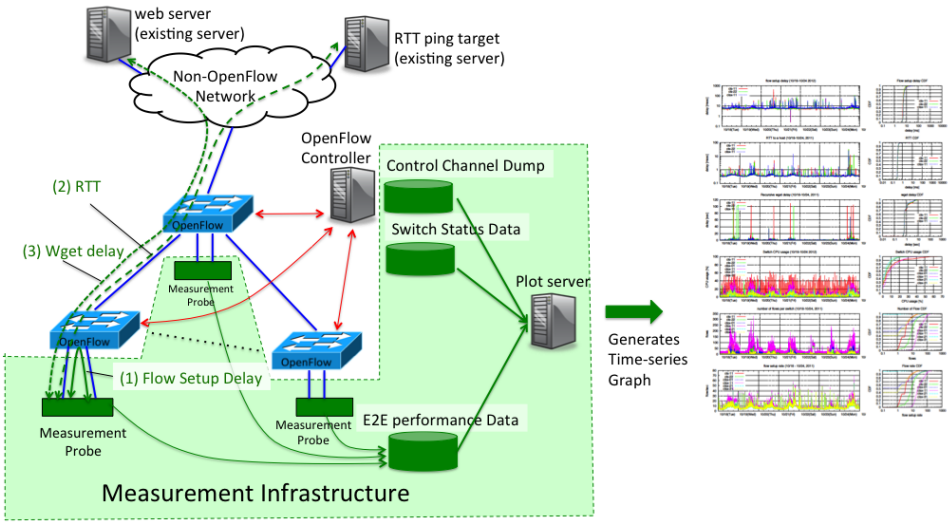



Figure 9: Measurement Infrastructure

**Flow-setup time** We measure the flow setup time using simple 10-second interval `pings` between a pair of probes attached to a switch; we use a 10-second interval (which is greater than the typical idle timeout of rules (5 second) inserted by NOX version 0.4) to allow the flow table entry to expire, causing each new ICMP packet to generate a packet_in message at the OpenFlow controller. The flow setup delay plot shows the bi-direction flow setup time experienced by the `ping` traffic.

**Round-trip time** We measure the round trip `ping` delay from the measurement probe to the target host located outside of our OpenFlow network.

**Recursive `wget` delay** We measure the delay to download a pre-determined single web page including its embedded images and links using recursive option of `wget` command. This mimics the web browsing delay that the end user perceives; the target web page was `http://www.stanford.edu/`.

The following is a list of switch performance metrics that we collect to understand the overall network behavior.

**Switch CPU usage** As we found in the beginning of OpenFlow deployment in Phase 1, when the switch CPU shows a high load the network is not functional. Since most OpenFlow-enabled switches today use weak CPU that can be overloaded, we monitor the usage carefully.

**Flow table entries used** Today's OpenFlow-enabled switches are often limited by the size of their flow table, making it important to monitor the usage of the flow table. This information also gives an idea of the network usage in the production setup.

**Flow arrival rate** Today's OpenFlow-enabled switches, owing to the weak CPU, are not capable of processing a large burst of new flows (In reactive mode, each new flow arrival corresponds to 3 control plane messages). Thus, we track the arrival rate of new flows (i.e., packet_in).

### *9.2. Example Performance Snapshot*

Barring any network problems, we observe that the SDN deployment behaves the same as a legacy one except for flow setup time because once the flow actions are cached, packet forwarding works the same with OpenFlow and legacy operation. In Fig. 10 we present a week's snapshot of the performance metrics for the deployment in CIS building. We observe that the metrics are stable and are within expected limits.

1. *Less than 100-millisecond flow setup time* We observe that flow setup time takes approximately 10ms in our deployment and most users do not perceive this delay in their network use.

2. *Sub-millisecond round-trip time* as observed from the RTT progression plot.

3. *0% packet loss in network*, as observed from the fact that the CDF of the RTT values reaches 1.

4. *Less than 1 sec total delay for 80% of* `wget` *requests to* `www.stanford.edu` *webpage with over 50 objects*, as observed from the `wget` delay plot.

5. *No CPU overload* as observed from the SNMP-based CPU usage statistics, where the CPU utilization is bound within 70%.

Besides our active monitoring using probes, we also measure and present in Fig. 10 the number of flows and flow setup rate seen by the controller by querying it. We observe an upper-bound of 320 flows that arrive at a maximum rate of 60 flows/sec. This is well within the limits of what the HP switch can handle. Lastly, we measure the number of WiFi users by contacting the `hostapd` daemon on the WiFi APs; the count of users exhibits a diurnal pattern and reaches a maximum of 18 during the busy hours.

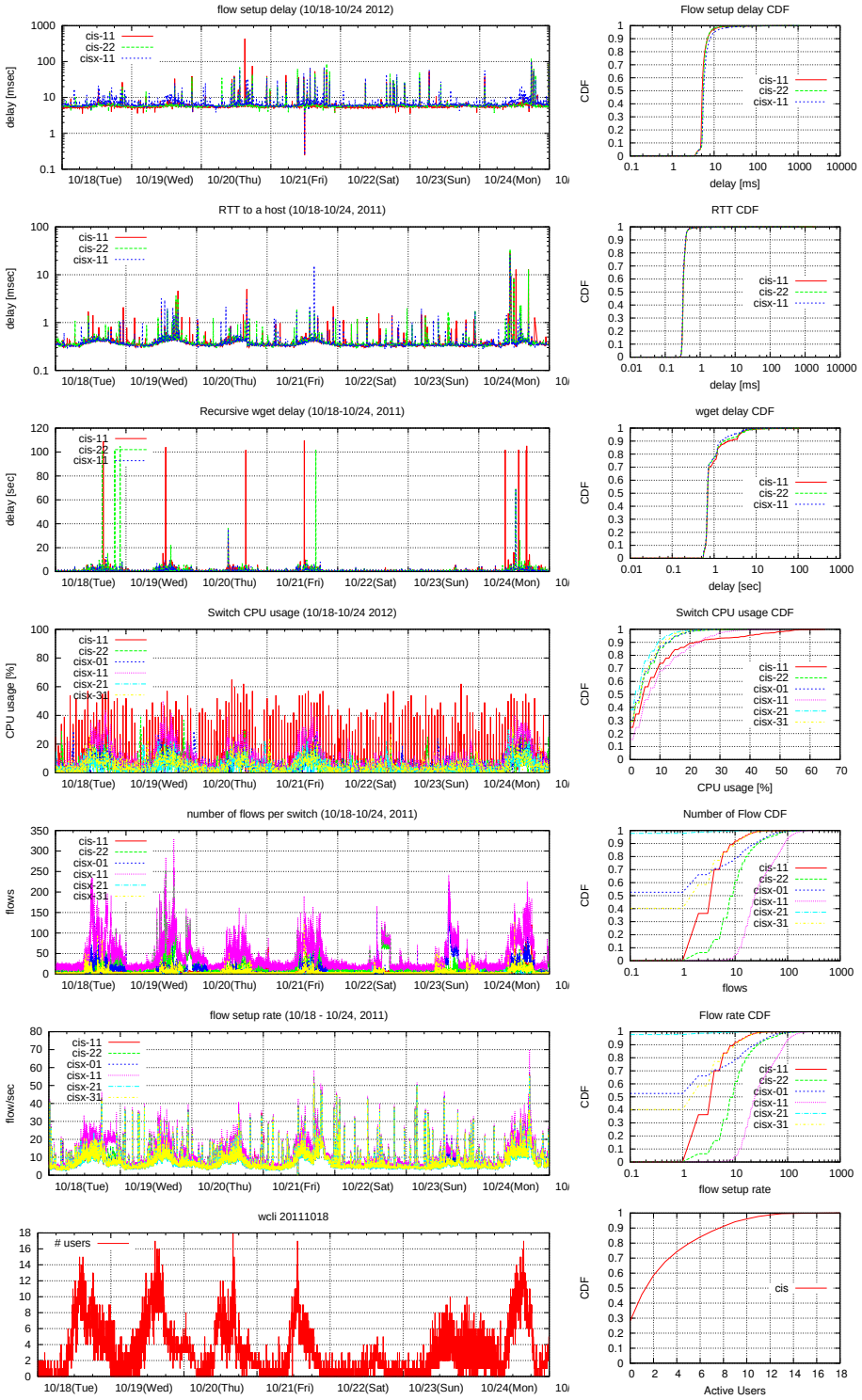
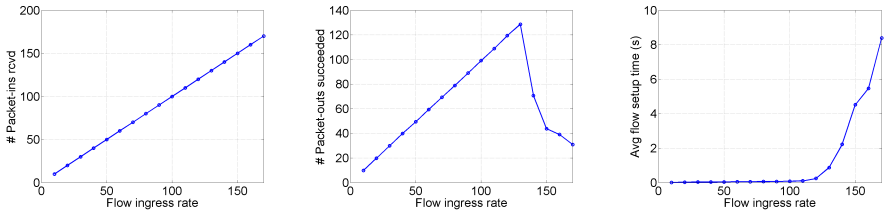





Figure 10: Measurement data plot for a week (10/18-10/24, 2011). Flow setup time, RTT delay, `wget` delay, CPU usage, Number of active flows, Flow arrival rate and Number of users are shown (in this order). The left column shows time series and the right coloun shows the CDF




(a) # of packet_in messages received by controller (b) # of packets that were transmitted from buffer (c) Average flow setup time

Figure 11: Summary of the behavior of a Broadcom-based OpenFlow switch for different ingress flow rates. Comparing the different characteristics of the data plane and the control plane shows where the switch becomes too much loaded to handle reactive flow handling. In the above example, the flow setup time grows greater than 100 ms when the flow ingress rate exceeds 110 flows/sec. Thus, we recommend using this switch only within that flow ingress rate.

Besides the network performance, we also conducted active tests to understand the behavior of the switches[11]. As described in the SDN reference design of Section 2, the flow setup process in SDN deployments is limited by the performance of switch CPU subsystem and suffers from flow setup time overhead. To better understand this aspect, we used a tool similar to oflops [30] to evaluate the limitations of switch hardware. The general idea of the tool is to generate new short-lived mice flows (with flow size of just 1 packet) and observing the controller communications as well as the output of the switch. Our evaluation showed that there are several limitations in today's hardware. To illustrate this, we present the behavior of a Broadcom-based OpenFlow switch in Fig. 11. The figure shows that the switch starts dropping new flows after the flow ingress rate crosses 110 flows/sec. Once the switch load becomes too high, the flow setup time, packet transmission and flow expiry message generation becomes unpredictable and poor.

### *9.3. Network Performance Diagnosis Process*

In addition to revealing user experience, the time-series plots can be correlated to infer the root cause of the issues for some quick diagnosis of network problems. Here we present some insights or rules-of-thumb to aid future SDN network administrators. Table 1 summarizes how we narrow down the root cause.

*If we observe large flow setup times* for a prolonged period, we first check the RTT plot for the non-OpenFlow network to see if this is really an issue with the

[11]Previous analyses showed that most controllers (viz., NOX [28], Maestro [23], Beacon [3]) are capable of reactively handling over 40,000 new flows ingressing per sec [1, 7]. Thus, for most enterprise deployments, the switch bottlenecks overshadow the controller limitations.

OpenFlow network. If RTT of non-OpenFlow network part is normal. If yes then the root cause must be in the OpenFlow part, i.e., processing delay in switch, or the controller processing delay[12]. To identify the exact issues requires looking at the other metrics. For example, as shown in Table 1, if both the switch CPU usage and flow setup rate is low while the flow setup time is too high, it is likely that the root cause is the slow processing time in the controller (Case # 1 in the table).

*If the switch CPU is high* in spite of the low flow setup rate (Case # 2), the switch CPU is overloaded for some processing other than packet_in. The root cause can be that the controller is sending too many message to the switch or the switch is busy doing some other job. We need to look into the control channel dump to distinguish these two, but at least we can narrow down the cause and warrant further debugging.

*If both the switch CPU and the flow setup rate are high* (Case # 3), the root cause is likely the high flow setup rate. To see the packet that caused this issue, we look for the packet_in message in the control channel dump to see what packets caused the burst of packet_ins.

Table 1: Typical symptom and possible root cause

| Case | Symptom | | | | | Possible Root Cause |
|---|---|---|---|---|---|---|
| | FST | RTT | CPU | #Flow | Flow Rate | |
| 1 | High | * | Low | * | Low | Controller |
| 2 | High | Low | High | * | Low | Switch is overloaded |
| 3 | High | Low | High | * | High | Burst flow arrival |
| 4 | High | Low | Low | * | * | Switch software |
| 5 | High | High | * | High | High | Flow Table overflow |
| 6 | Low | High | * | * | * | Switch software or the controller |

We sometimes observed periods of large round-trip delay while all other metrics being good (Case #6). This indicates that the flow rule is not installed in the switch flow table or the switch is forwarding packet in software. To distinguish this, we further look into the control channel dump to see whether flow_mod (flow table installation) is properly done or not.

Another possible scenario (which is not shown in the table) is when the `wget` delay is too high, while all other metrics are normal. After eliminating the possibility of web server issues (by inspecting the `wget` delay on the non-OpenFlow network), we guess the reason as the switch being unable to handle the high flow

[12]If the legacy network is behaving normal, then the communication delay between the switch and the controller cannot be the root cause.

arrival rate, despite processing pre-cached entries fine.

In all examples above, though we need to inspect the control channel dump to accurately identify the root cause, correlating the time-series plots help in shortlisting the possible root causes and reducing the amount of work in network performance debugging.

### *9.4. Network Debugging*

The SDN architecture, with its centralized vantage points, allows for easier debugging and troubleshooting of network problems. The main information that SDN provides beyond what legacy networks provide are: 1) the flow-table routing information, 2) the flow-level traffic statistics using stats_request command, 3) the sequence of events that occurred in the network based on the control message exchange, and 4) the complete network topology. To make the most of this information, we developed several tools to assist in network debugging [31], such as the OpenFlow plugin for `wireshark`. The tools either collected data through the API that controllers provided or processed packet dump trace of the communication between the controller and the switches.

The most effective approach for debugging network problems was inspecting the control channel traffic between the controller and the switch using `wireshark` and correlating with the observations from the data plane servers or monitors. This debugging approach is well-suited to identify bugs in the control applications, e.g., incorrect rule that never gets matched, as well as non-OpenFlow issues in the end-device, e.g., IP address conflict caused by an Apple laptop with state caching issues.

Since all OpenFlow-enabled commercial switches used in Phase 1 deployment were based on Stanford reference code, interoperability among them did not turn out to be a big issue. Still there were bugs or vendor-dependent behavior, such as the logic behind inserting flow-table entries, that were exposed in a multi-vendor network after extensive use. Our deployment played a significant role in finding these bugs or interoperability issues and we worked closely with vendors to fix them. We present 2 issues (1 for controller and 1 for switch) that was uncovered by our deployment:

- *Incomplete handover*: When a WiFi terminal *B* moved from one WiFi AP to another AP while running `ping` to terminal *A*, the ICMP reply stopped being received by the WiFi terminal *B*. To debug this issue, we inspected the flow-table of all the involved switches, as well as the control communication between the involved switches and the controller. Fig. 12 illustrates the root cause we found: After handover to the new AP, the flow table entry in switch *X* and *Y* were not updated for the ICMP reply flow, causing the ICMP reply

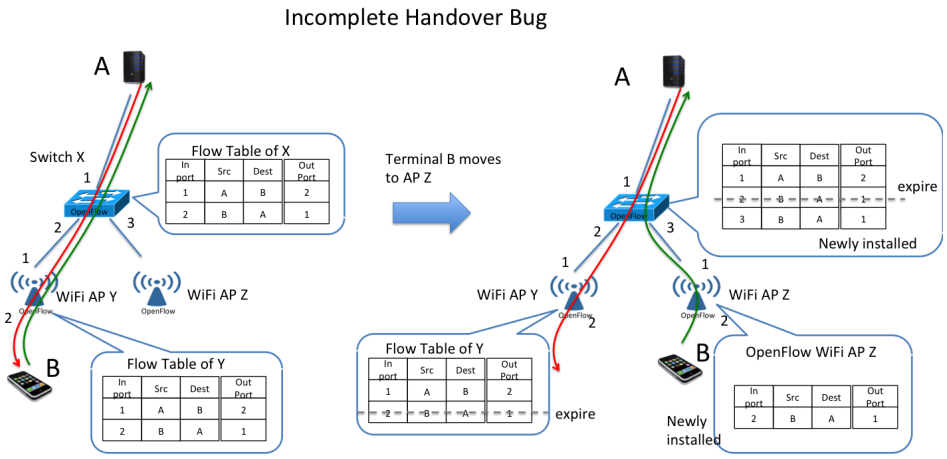



Figure 12: Debugging packet loss after handover. The left figure illustrates the flow tables before handover, and the right figure shows after handover. Although all switches updated the entries for the ICMP request from terminal *B*, they did not have the right entry for the ICMP reply from terminal *A*. The first entry cached in Switch *X*'s flow-table is, thus, the root cause of the loss.

packet to be sent to the old location of the WiFi terminal *B*. We fixed this bug by changing the controller logic such that when the controller detected the change in location of a terminal (through packet_in from a new WiFi AP *Z*) it flushes the flow entries related to the reverse direction traffic and reprograms the switches en route.

- *Race condition bug in a switch*: We observed poor web browsing experience in our production network. We debugged that by dumping the TCP SYN and SYN-ACK packets of the HTTP session, at the client-side and server-side, using `tcpdump` and correlated them with the control plane events observed using `wireshark`. We observe an increased drop rate for the TCP session handshake packets. The correlation, further, revealed that the control plane events were as expected, but the packets were not properly forwarded by the data plane, even though the flow-table of each switch in the network was accurate – this is very intriguing of course. This loss of the TCP handshake packet explained poor user experience for browsing, because the TCP session waits for a prolonged TCP timeout (in the order of seconds). However the handshake packet loss behavior was not consistent with everything else. So we hypothesized that there was a race condition as shown in Fig. 13. With this race condition, a packet (the TCP SYN packet) arriving at the second hop switch, while the switch is in the middle of processing a flow_mod operation

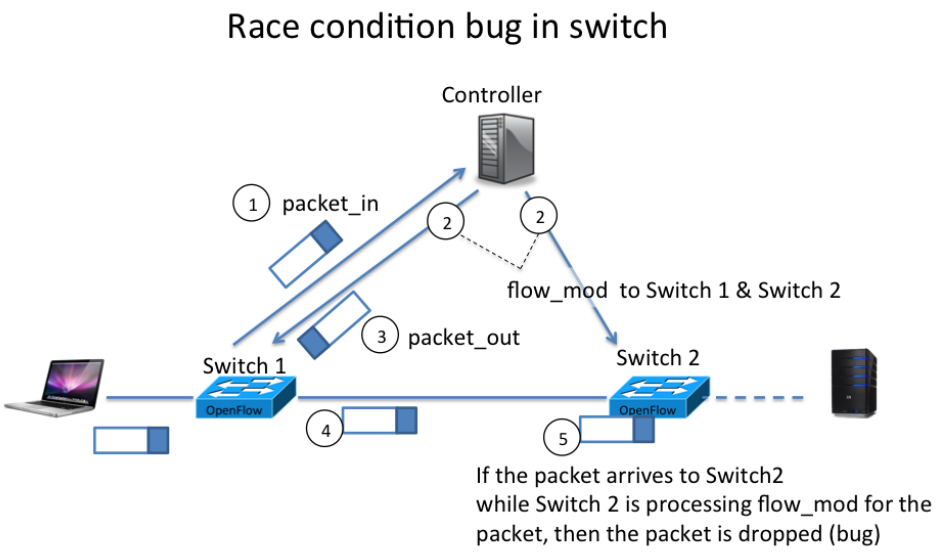



Figure 13: Debugging non-deterministic loss of TCP SYN/SYN-ACK packets.

(to insert a new rule), is sometimes dropped. To confirm this, we placed a delay box between the 1st and the 2nd switch and found that the packet is not dropped. That is, if we delay the packet enough (so as to give time to the second switch to finish processing the flow_mod message), everything works just fine.

These examples confirm that SDN enables debugging of network problems by inspecting the control channel traffic between the controller and the switch and correlating with the observations from the data plane servers or monitors. This approach shows large potential for future automated troubleshooting.

## 10. Concluding Remarks

Software Defined Networking (SDN) has captured the imagination of the network industry and research community. Leading network owners and operators such as Google, Verizon, NTT Communications, Deutsche Telekom, Microsoft, Yahoo!, and others argue that SDN will help them build networks that would be simpler, easy to customize and program, and also easier to manage. They expect these attributes of SDN will lead to networks with reduced capital and operational expenses and increased revenues due to new services enabled. They also believe SDN with virtualization would allow the network to be a programmable pluggable

component of the emerging cloud infrastructure for multi-tenancy operation. Most network equipment and software vendors are embracing SDN to create products and solutions to meet the expectations of the network owners and operators and bring new innovations to the market place. The network research community considers SDN as an opportunity to help shape of future of networking and bring new innovations to the market place. In short SDN has emerged as the new paradigm of networking.

For this rapid maturing of SDN, four years of deployments combined with the applications and demonstrations played an important role. They demonstrated the key value proposition of SDN; proved that the physical infrastructure can support multiple concurrent virtual networks for research and production use; revealed a number of performance tradeoffs; provided valuable input to the OpenFlow specification process; helped vendors and help grow the larger ecosystem. Our deployments achieved the original goals and taught us a number of important lessons. The following is a list of our high level takeaways from this experience:

- Visually compelling demonstrations were critical for showing power of SDN, and that is, enable innovation by writing a relatively simple software program to create new network control and management capabilities. These demonstrations helped to win over people including skeptics. Although this lesson is well known, it is not easy to practice and many ignore it. Fortunately we emphasized demonstrations with very good results.

- The deployments required us to develop measurement and debugging tools and processes for SDN. On the other hand we realized SDN architecture helps in network measurement and debugging – the flow table as an abstraction of a switch and the central vantage point of the controller. We took advantage of the SDN architecture to debug a few important and difficult network issues and this shows another important value proposition of SDN for troubleshooting.

- One of our goals has been to show how SDN infrastructure, with slicing and virtualization, can support both research and production traffic on the same physical infrastructure. Though we demonstrated feasibility and supported both research and production on our group's operational infrastructure, it is not easy to do in the short term. SDN is evolving very rapidly and lot of energy is required just to create a production deployment. If it has to also support research and experimentation, then getting the whole infrastructure to be stable and useable represents a big challenge. The next round of commercial SDN products, with support for network virtualization, will make it

much easier for most network administrators to achieve this important capability.

- Experimentation on a real deployment with users has lot of benefits. However, it has limitations. A researcher cannot create arbitrary topologies; cannot scale the network or the number of end users; and cannot experiment with arbitrary traffic mix. Furthermore, a real deployment is time and resource intensive. This led some of our colleagues to develop *Mininet* to help researchers and developers to quickly emulate SDN of arbitrary topology, scale, and traffic mix. On this emulated network, one can develop and experiment with real SDN control stack that can subsequently be deployed on a physical infrastructure without any changes [22]. We expect the mix of Mininet and real deployments to cover the entire spectrum for research and development.

- Deployments and good demonstrations require significant effort for a few years to achieve the impact that is possible.

- These deployments helped create a collaborative and productive relationship between the university community and the network vendor community. All through four years of deployments, we and others in our partner universities worked closely with many vendors. In fact we worked with almost all vendors that have been developing SDN technologies, platforms, or products. We provided OpenFlow software reference implementations, offered to test, profile, use SDN switches and controllers, provided bug reports, and suggested new features to make the products better. For example we worked with switch vendors such as Cisco, HP, Juniper, NEC and Pronto, and controller vendors such as Nicira, NEC and BigSwitch. We worked extensively with NEC, HP, Nicira, and BigSwitch. HP has been a vendor of choice for the legacy network at Stanford and it provided OpenFlow-enabled switches very early to be used in our production setting. NEC committed ample resources to OpenFlow and SDN development, and offered very interesting OpenFlow-enabled switches and controller for our research and experimentation. Pronto provided a very low cost OpenFlow switch that we and many other universities found very attractive for research and experimentation. Nicira participated in all aspects of SDN and provided open-source SDN controllers, NOX and SNAC, that played a very important role in early deployments and early success of SDN. Subsequently, BigSwitch provided a controller for the potential production use. This kind of relationship between universities and networking vendors has been missing for many years and has been a drag for the field. We hope both the vendor and university

community will build on the success we have had with SDN deployments and together help move networking and SDN forward.

Despite all the successes of SDN so far, it is still in an early stage of development. We as a community have completed the first stage of SDN development to demonstrate its potential and articulate an exciting research agenda to realize its full potential for various domains of use: including data center, service provider, enterprise, and home. As we and others pursue this research agenda for the next several years, we will require many more deployments and demonstrations. This 4-year four phase deployment effort is just the beginning.

**Acknowledgements**

We were supported by several individuals part of vendor organizations, university IT departments, GENI support staff, and researcher and experimenters. We would especially like to thank the following people, categorized based on their primary contribution:

**OpenFlow specification process and reference implementations.** Martin Casado, Ben Pfaff, Justin Petit, Guido Appenzeller, Dan Talayco, Glen Gibb, Brandon Heller, Yiannis Yiakoumis, David Erickson, Rob Sherwood, Mikio Hara, Kyriakos Zarifis, James Murphy McCauley, Nick Bastin

**Commercial implementations.** Jean Tourrilhes, Sujata Banerjee, Praveen Yalagandula, Charles Clark, Rick McGeer, Atsushi Iwata, Masanori Takashima, Yasunobu Chiba, Tatsuya Yabe, Sailesh Kumar, Valentina Alaria, Pere Monclus, Flavio Bonomi, Lin Du, James Liao, Oscar Ham, Adam Covington

**Stanford deployment.** Charles M. Orgish, Joseph Little, Miles Davis

**Demonstrations.** Larry Peterson, Sapan Bhatia, Brighten Godfrey, Aaron Rosen, Vjeko Brajkovic, Mario Flajslik, Nikhil Handigol, Te-Yuan Huang, Peyman Kazemian, Jad Naous, David Underhill, Kok-Kiong Yap, Yiannis Yiakoumis, Hongyi Zeng, Michael Chan

**GENI project office, NLR, Internet2.** Joshua Smift, Heidi Picher Dempsey, Chaos Golubitsky, Aaron Helsinger, Niky Riga, Chris Small, Matthew Davy, Ron Milford

**GENI campus deployment.** Aditya Akella, Nick Feamster, Russ Clark, Ivan Seskar, Dipankar Raychaudhuri, Arvind Krishnamurthy, Tom Anderson, Kuang-Ching Wang, Dan Schmiedt, Christopher Tengi, Scott Karlin, Jennifer Rexford, Mike Freedman, Glenn Evans, Sajindra Pradhananga, Bradley Collins, Shridatt Sugrim, Ayaka Koshibe, Theophilus A. Benson, Aaron Gember, Hyojoon Kim, Tim Upthegrove, Ali Sydney

## References


[1] Amin Tootoonchian, Sergey Gorbunov, Yashar Ganjali, Martin Casado, Rob Sherwood, On Controller Performance in Software-Dened Networks, in: 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE), 2012.

[2] A. Anand, V. Sekar, A. Akella, Smartre: an architecture for coordinated network-wide redundancy elimination, in: ACM Sigcomm, 2009.

[3] Beacon Controller, http://www.openflowhub.org/display/Beacon.

[4] Tunneling capsulator, www.openflow.org/wk/index.php/Tunneling_-_Capsulator.

[5] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, S. Shenker, Ethane: taking control of the enterprise, in: ACM Sigcomm, 2007.

[6] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, S. Shenker, Sane: a protection architecture for enterprise networks, in: Proceedings of the 15th conference on USENIX Security Symposium - Volume 15, 2006.

[7] Controller Performance Comparisons, http://www.openflow.org/wk/index.php/Controller_Performance_Comparisons.

[8] David Erickson et al., A demonstration of virtual machine mobility in an OpenFlow network, in: Proc. of ACM SIGCOMM (Demo), 2008.

[9] Deploying OpenFlow, http://www.openflow.org/wp/deploy-openflow.

[10] Expedient Clearinghouse, http://www.openflow.org/wk/index.php/Expedient.

[11] G. Gibb, D. Underhill, A. Covington, T. Yabe, and N. McKeown, OpenPipes: Prototyping high-speed networking systems, in: Proc. of ACM SIGCOMM (Demo), 2009.

[12] GENI: Global Environment for Network Innovations, http://www.geni.net.

[13] P. B. Godfrey, I. Ganichev, S. Shenker, I. Stoica, Pathlet routing, in: ACM Sigcomm, 2009.

[14] Going With the Flow: Googles Secret Switch to the Next Wave of Networking, http://www.wired.com/wiredenterprise/2012/04/going-with-the-flow-google (April 2012).

[15] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, H. Zhang, A clean slate 4d approach to network control and management, SIGCOMM Comput. Commun. Rev. 35 (5).

[16] G. P. Group, Geni design principles, Computer 39 (2006) 102–105.

[17] N. Handigol, B. Heller, V. Jeyakumar, D. Mazires, N. McKeown, Where is the debugger for my software-defined network?, in: Proceedings of Hot Topics in Software Defined Networking (HotSDN), 2012.

[18] B. Heller, R. Sherwood, N. McKeown, The controller placement problem, in: Proceedings of Hot Topics in Software Defined Networking (HotSDN), 2012.

[19] Internet2 network, http://www.internet2.edu.

[20] New generation network testbed jgn-x, http://www.jgn.nict.go.jp/jgn2plus_archive/english/index.html.

[21] P. Kazemian, G. Varghese, N. McKeown, Header Space Analysis: Static Checking for Networks, in: Proceedings of 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2012.

[22] B. Lantz, B. Heller, N. McKeown, A network in a laptop: rapid prototyping for software-defined networks, in: Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks, 2010.

[23] Maestro Platform, http://code.google.com/p/maestro-platform.

[24] N. McKeown, Making sdns work, Keynote talk at Open Networking Summit 2012, http://opennetsummit.org/talks/ONS2012/mckeown-wed-keynote.ppt.

[25] N. Handigol, S. Seetharaman, M. Flajslik, N. McKeown, and R. Johari, Plug-n-Serve: Load-balancing web traffic using OpenFlow, in: Proc. of ACM SIGCOMM (Demo), 2009.

[26] Netfpga platform. http://netfpga.org.

[27] Nicira, inc., www.nicira.com.

[28] NOX - An OpenFlow Controller, http://www.noxrepo.org.

[29] NTT COM Announces First Carrier-based OpenFlow Service, http://www.nttcom.tv/2012/06/13/ntt-com-announces-first-carrier-based-openflow-service (June 2012).

[30] OFlops and CBench for OpenFlow benchmarking, http://www.openflowswitch.org/wk/index.php/Oflops.

[31] Monitoring and Debugging Tools for OpenFlow-enabled networks, http://www.openflow.org/foswiki/bin/view/OpenFlow/Deployment/HOWTO/ProductionSetup/Debugging.

[32] Omni Client, http://trac.gpolab.bbn.com/gcf/wiki/Omni.

[33] ON.Lab, http://www.onlab.us.

[34] Open Networking Summit, http://www.opennetsummit.org.

[35] OpenFlow Tutorial, http://www.openflow.org/wk/index.php/OpenFlow_Tutorial.

[36] Opt-in Manager, http://www.openflow.org/wk/index.php/OptIn_Manager.

[37] L. Peterson, A. Bavier, M. E. Fiuczynski, S. Muir, Experiences building planetlab, in: Proceedings of the 7th symposium on Operating systems design and implementation (OSDI), 2006.

[38] Plastic Slices Project Plan, http://groups.geni.net/geni/wiki/GEC11PlasticSlices.

[39] R. Sherwood et al., Carving research slices out of your production networks with openflow, SIGCOMM Comput. Commun. Rev. 40 (1).

[40] The Future of Networking, and the Past of Protocols, http://www.opennetsummit.org/talks/shenker-tue.pdf.

[41] R. Sherwood, G. Gibb, K. kiong Yap, M. Casado, N. Mckeown, G. Parulkar, Can the production network be the testbed, in: USENIX OSDI, 2010.

[42] Simple Network Access Control (SNAC), http://www.openflow.org/wp/snac.

[43] A. Wundsam, D. Levin, S. Seetharaman, A. Feldmann, OFRewind: Enabling Record and Replay Troubleshooting for Networks, in: Proc. USENIX ATC, 2011.

[44] K.-K. Yap, S. Katti, G. Parulkar, N. McKeown, Delivering capacity for the mobile internet by stitching together networks, in: Proceedings of the 2010 ACM workshop on Wireless of the students, by the students, for the students (S3), 2010.

[45] K.-K. Yap, M. Kobayashi, D. Underhill, S. Seetharaman, P. Kazemian, N. McKeown, The Stanford OpenRoads deployment, in: Proceedings of the 4th ACM International workshop on Experimental evaluation and characterization (WINTECH), 2009.