Towards Production Deployment of a SDX Control Framework

Mert Cevik*, Michael Stealey*, Cong Wang*, Jeronimo Bezerra†,
Julio Ibarra†, Vasilka Chergarova†, Heidi Morgan‡, and Yufeng Xin*

* RENCI, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA
† CIARA, Florida International University, Miami, FL, USA
‡ ISI, University of Southern California, Marina del Rey, CA, USA

***Abstract*—Deploying and operating a production software defined network (SDN) requires substantial engineering effort to maintain the stringent function, availability and security requirements. One major challenge is the SDN controller software system that needs an efficient development and deployment environment, and a systematic process to combine both network engineering and software engineering best practices to fit in a targeted network environment.**

In this report, we present our recent engineering work to bring a wide-area SDN controller prototype towards production deployment and operation. We primarily focused in three major areas: Enhancement of the software functions and the substrate configurations to fill the data plane gap between the prototype and the physical network substrate to fully support advanced layer-2 connection services; creation of a high-fidelity testing pipeline consisting of the unit test, emulation, and a realistic testbed; and establishment of a continuous integration and continuous deployment (CI/CD) environment and the DevOps process. Our experience proved that the presented environment and process greatly increased the software quality and development efficiency, which would ultimately lead to a more reliable migration to the targeted production network deployment.

I. Introduction

Through centralized control and control plane programmability, software defined network (SDN) provides a new paradigm at different parts of the Internet which could greatly automate the network operation and reduce the operator's burden.

While the SDN controller software takes over some of the major control tasks that were normally performed by operators, deploying and operating a complex SDN software system still remains a great technical challenge, especially when migrating an existing network with legacy non-SDN elements. In this new territory, network engineering and software engineering have to be considered holistically in order to guarantee the successful operation of the network, especially when there are stringent requirements on system availability and security in a production network.

While SDN deployments have become pervasive in data centers, its adoption in the wide area network (WAN) has also been growing rapidly, especially in large cloud operators' networks that are over geographically distributed areas [1]. Applying SDN over a multi-domain network environment has attracted lots of interest in recent years. However,it remains a great challenge to migrate a multi-domain network to become SDN capable to (1) automate the advanced end-to-end network services in other layers, such as layer-1 and layer-2 connections; (2) provide QoS guaranteed service, due to well-known provisioning complexity, domain heterogeneity, and stringent SLA (Service Level Agreement) requirements. The best engineering practice in this area still largely relies on in-person communications and manual configurations of a group of collaborative operators, which often takes a long time and is error-prone.

One particular multi-domain WAN network environment is the so-called Open Exchange Point (OXP) for more efficient interconnection of participating networks. OXP finds its prominent use cases in the global research and educational network community where efficient and high-bandwidth networking services for dissemination of large volume of data is essential. Several EU and US OXPs pioneered the SDN adaptation through experimental testbeds and pilot deployments [2], [3].

The most challenging parts of adapting the SDN technology is the choice or development of the SDN controller software. The common open source SDN controller frameworks are far from sufficient to become operational in the complex OXP landscape. Popular single Openflow (OF) controllers, such as Floodlight and Ryu, are not scalable for the wide area distributed OXP networks. Popular distributed control frameworks, like ONOS, also need significant customization and further development to provide the needed services at different network layers with devices from different vendors [4]. A main challenge is that some key networking service functions like topology discovery, connection computation and management, and QoS do not exist in the existing controller framework. The lack of north bound APIs to end users and applications as well as the authentication and authorization functions also made it difficult to deploy the common controller framework in a production setting. Another challenge is the highly heterogeneous physical devices from different vendors in each domain. They often comply with the OF specification to different degrees, as there exists multiple versions of the specification, and implementation details are always very different. More advanced switches, not like the simple OF white box switches, support network virtualization natively, which requires device specific configurations to work with a specific OF controller

This work is supported by the US National Science Foundation through the NSF awards OAC-1451024.

of choice.

As a result, some OXPs started to develop their own controller software with their specific network domains, vendors, and use cases in mind. AtlanticWave-SDX (AW-SDX) represents such a controller framework that has recently finished a prototype with a distributed system architecture design [5]. While the prototype project started with a functional design in collaboration with the network engineering team and followed good software development practices, there still are some important gaps towards production deployment. After a systematic review process, we identified three major areas for improvement.

Firstly several AW-SDX functions need to be implemented and enhanced in order to fill the data plane gap between the prototype and the physical network substrate to support the main connection services reliably: dynamic point-to-point (P2P) and point-to-multipoint (P2MP) layer-2 services with rate limiting. These include the failure handling capability that would not manifest its importance in the prototype but is extremely important for the operationalization of the SDX system.

Secondly, a high-fidelity testing environment is imperative to provide the complete functional and performance coverage. This environment should consist of a pipeline of unit test, emulation, and an at-scale testbed to guarantee the software development and deployment quality. The last one is to create a continuous integration and continuous deployment (CI/CD) environment to provide software quality guarantee and deployment efficiency. This effort follows the recent DevOps trend that was started from the IT system domain and quickly became the main stream methodology due to its superior capability in combining the software development (Dev), IT (including network) operations (Ops) efforts.

In this report, we present our work in engineering the AmLight SDX control framework towards deployment and operations to a production OXP network environment in the context of the above three thrusts.

The rest of this report is organized as follows. We introduce the AW-SDX controller software and functional enhancement towards the targeted production network in Section II. We present a high-fidelity system emulation and testing pipeline environment in III. We describe the DevOps-based software engineering practice that features a CI/CD (continuous integration/continuous deployment) environment that covers the complete pipeline from the testing environment to the production network. Evaluation results are presented in Section V and the paper is concluded in Section VI with lessons learned and future work.

II. AW-SDX Control Framework Enhancement

SDN assumes a separate and logically centralized control plane. The OF framework standardized the southbound API to control the switches. Various controller frameworks have been developed to implement the OF specification and provide northbound APIs for applications to interact/program with the SDN. As a SDN network scales up, single controller (Ryu, Floodlight, etc.) would suffer from various performance, scalability, and fault tolerance problems. Therefore, distributed multiple controller frameworks have been emerging, with ONOS being the most popular one. However, these multi-controller frameworks primarily focus on scalability and availability challenges of the controller software, in which physically distributed controllers are all part of the same logical controller that still keep the same overall network view. For the multi-controller framework, decisions also have to be made to decide the placement and connectivity of the controllers, i.e., the control plane network. The common practice is to separately provision a static control plane network (normally the same management network) for both inter-controller and controller-switch communications. There have been several efforts to automate the creation of the control plane network, via a so-called bootstrapping process [6].

For the targeted multi-domain OXP network environment, the controller design choices range from the hierarchical resource orchestration oriented and protocol oriented distributed model architecture. A representative orchestration architecture uses a central controller to interact with local domain proxies via a brokering service to make high level end-to-end resource provisioning decisions [7]. Another representative distributed network control framework is the popular NSI [3]. However, these legacy system do not support the SDN standards, especially the OF specifications.

The AW-SDX controller framework is a distributed solution that has native SDN support as part of its design goals. As depicted in Figure 1, the AW-SDX controller is a two-level controller architecture where a central controller (SDX controller) is tasked to handle users' request (APIs) and control a group of local controllers (LC1, LC2, and LC3 in the figure) , each of which controls a part of the SDN network (network1, network2, and network3 in the figure). The local controller embeds a customized Ryu controller to control the OF switches in its own domain. The AW-SDX control plane network that connects the SDX controller and all LCs can be bootstrapped via an in-band data plane network. The current implementation of AW-SDX assumes a pure OF data plane consisting of switches supporting OF1.3. Specifically it targeted a production SDN network built from the advanced Corsa switches. This distributed architecture targeted the typical wide area OXP network landscape where scalability is a top concern. More details on the AW-SDX architecture can be found in [5].

In the following, we describe four common function and quality pitfalls that were really hard to catch in the prototype work. All these pitfalls turn out to be critical to the successful operation of the production network. Not surprisingly they all became our focus in the process of migrating towards the production deployment.

A. Data plane, multi-tenants, and Corsa switches.

While the public Internet and enterprise networks primarily operate on the 'narrow waist' IP layer, the research and educational networks often need to provision high-bandwidth

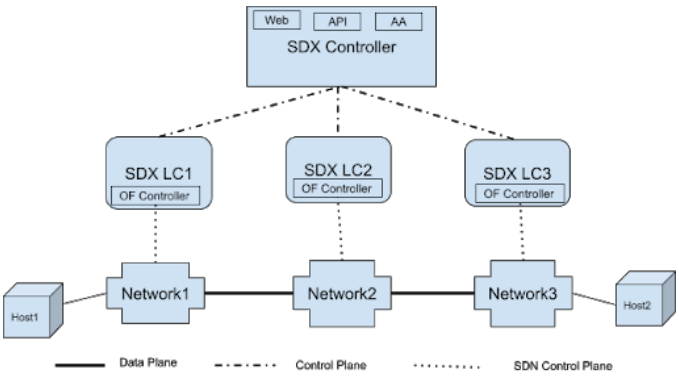


Fig. 1: Distributed SDX Controller Architecture

layer-2 network services to support data-intensive scientific applications. This is particularly true for OXP networks. Therefore, one major assumption that the AW-SDX controller made was that it will operate at layer-2 and have VLAN connections as its primary service. Secondly, OXPs need to support a large number of organizations with a broad spectrum of SLA requirements, and multi-tenant virtual network operational requirements. The low-level OF rules for data plane provisioning are highly device dependent. While the AW-SDX software architecture itself, especially the separated LCs, supports plug-ins of different substrate devices, the current implementation only supports the OVS and Corsa switches. This means the production network substrates need to be configured accordingly to a state that the AW-SDX is able to provision end-to-end services. Because the Corsa switches in the production network over the wide area OXP network are connected via several hops of non-SDN devices and networks, the precursory work to configure the underneath non-SDN network elements to create a SDN/SDX substrate is significant and substantial.

Most OF switches do not support device level virtualization in terms of flow space and performance isolation. A few switches support the hybrid mode, *i.e.*, splitting the physical ports into traditional layer-2 (learning switch and VLAN functions) and OF (controlled by outside OF controller) groups. The de facto virtualization solution is to add a software virtualization layer like Flowvisor; however, its development stopped at only supporting OF1.0.

To alleviate the performance and complexity challenges of existing solutions to support multi-tenant network environments, Corsa introduced the device level virtualization via the Virtual Fowarding Contexts (VFC) constructs and quickly gained many users in the research and education network communities. The VFC provides native layer-2 and layer-3 VPN supports [8]. Therefore, appropriate VFCs need to be designed and configured in order to work with the OF controller to support the designated services. There are two implications from using this advanced switch platform to the SDX development: (1) During the development stage, SDX software development largely relied on the Mininet with Open vSwitch (OVS) for prototyping and testing. However, data plane emulation in Mininet-OVS emulation is far from the actual device platforms. Even the initial implementation and testing against a single Corsa switch is not sufficient, because it lacks the network aspect. (2) On the other hand, to our advantage, VFC allows us to test and trial the SDX controller in an isolated slice in a SDN network without affecting other traffic.

Rate Limting. Adding to the complexity is that the Corsa switch did not fully comply with the OF 1.3 specification, and OF doesn't support rate limiting. Nevertheless, since the main design goal was to support an end-to-end layer-2 VLAN connection service, a layer-2 VFC needed to be configured in every Corsa switch in the network. However, this type of VFC doesn't support a rate limiting function. Our solution was to use an accompanying VFC in every switch in the data path to provide the rate limiting function. We show a configuration of a switch in Figure 2. VFCs are represented inside the green rectangle.

The VFC in the middle acts as the main OF switch. The VFC on the right side is a rate limiting VFC implemented in the prototype. These two VFCs are connected via a fiber connection via two pairs of physical ports. The OF controller will be instructed by the SDX controller to have the rate limiting VFC in the data path if a rate limiting request is served. However, in the real testbed testing with multiple Corsa switches, this configuration doesn't work for the rate limiting in P2MP connections. The root cause is that, unknown ports (multi-point nature) are tracked via OpenFlow Metadata. OpenFlow Metadata is optional and Corsa switches reject some optional metadata in different ways. This forced us to re-implement the rate limiting function with a different type of VFC, the one on the left side in the figure, which is a type of dedicated "Layer2-vpn" VFC. Packets from the main VFC will be sent to this VFC, through tunnels that have a shaped-rate configuration on them. These tunnels will be created and deleted on demand. Accordingly, the related functions in the SDX controller and the OF controller were also re-implemented.

Point-to-MultiPoint Connections Another example where re-implementation was necessary was the P2MP connection feature. Due to the small scale of the original testing environment, initial tests were all simpler P2MP connections that were manifested in two layers of trees where switches only need to forward traffic between two ports. However, when we tested a more complicated P2MP tree structure in a more complext testbed, as shown in the Figure 3, the intermediate node "DukeS1" didn't multicast the packet to the three ports in the tree correctly. We finally validated that the original OF broadcasting rules were not working in the P2MP use case, resulting in a re-implementation of the related OF rule generation functions via the OF "group" mechanism.

Legacy Network Devices and VLAN Continuity. In emulation or in the testing environment, the whole VLAN range was available on all the links. Therefore, the prototype assumed VLAN continuity, in which the same VLAN is

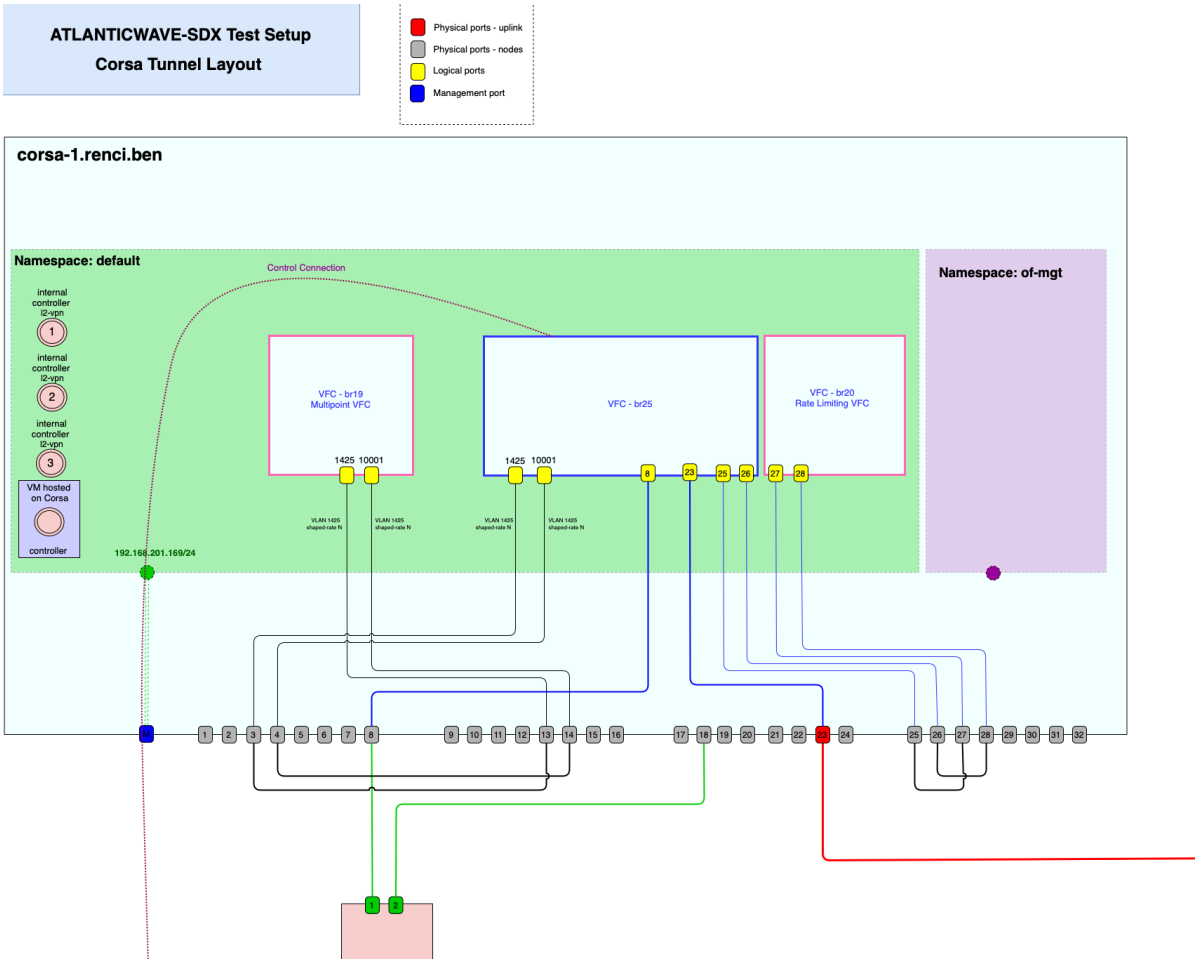


Fig. 2: Corsa VFC Configuration

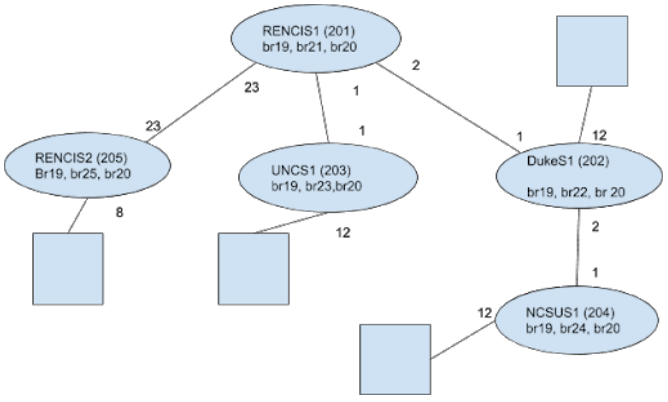


Fig. 3: A P2MP Connection Example

used for a connection, and for P2P or P2MP. However, in a production network over multiple domains, a link between two SDN switches is pre-provisioned through multiple legacy switches in the underneath networks. These networks normally have large chunks of VLAN pools for their local use, and only commit a small range of VLANs for SDN use. These available VLAN ranges on different SDN links are normally not the same, not even overlapping. As a result, we re-implemented the topology description, the path finding algorithm, and the VLAN translation OF rule function to remove the VLAN continuity constraint.

Dynamic connection creation and release. When connections come and go as on-demand or advanced reservation, network resources are reserved and released dynamically. It is a fundamental function to keep the network resource state correctly managed, including connection and resource states in the SDX controller and the LCs, as well as the flow entries in all the switches. This requires thorough at-scale testing using a sufficient number of test cases, and with a large number of connections. As a matter of fact, we were only able to discover and fix a few deeply hidden bugs in the prototype when handling flow entry creation and release, after we tested a large number of mixed on-demand and advance reservation P2P and P2MP connections in the testbed.

B. Failure handling

While the basic connection provisioning functionality and implementation were the focus of the AW-SDX prototype,

high availability is an imperative requirement for a production network. We systematically addressed the following failure scenarios:

Controller hardware and software crashes. The SDX controller and LC software may crash due to server resource (CPU and memory) depletion or unhandled bugs. If the controllers run in a cloud, the host servers or virtual machines may crash due to different reasons. In both cases, we need to be able to restart the SDX software and/or reboot the servers without losing the state of SDX controllers and the existing connections.

Therefore, following failure handling mechanisms need to be in place: (1) resuming the communication between the SDX controller and LC controller, which is accomplished by a socket-based communication implementation. (2) recovering the state information in AW-SDX system that includes the topology, all the existing users and connections that have been created. To save the state information, the AW-SDX system must rely on a persistent database.

The persistent database capabilities and schema were fully tested and tightened such that when a controller (AW-SDX controller or one of the Local Controllers) needs to be rebooted, the topology and all existing connection information and rules are fully recovered.

Management plane failures. The Management plane is an out-band pre-configured secured network for the system admin to access the servers running the SDX controllers and the switches remotely. Our solution follows best practices in system administration: (1) configure all the SDX controllers in one network behind a firewall for the SDX administrator to access; (2) as the switches are behind management networks separately administrated by local networks, we need to have an efficient communication process in place for the administrators and the SDX administrator to work together to fix access failures. At the RENCI testbed, all the controllers were deployed in the form of VMs, inside the RENCI production data center. This is the recommended model to deploy the SDX controller with the benefit of seamless migration of the controller VMs and management plane when the data center environment is changed. The management network interfaces are configured behind a firewall that only allows VPN access. We also planned to deploy a system health monitoring and intrusion detection system.

SDX control plane failures. As a distributed networking infrastructure, it is desired for AW-SDX to promptly detect failures, and to recover the failures anywhere within the federated infrastructure. There are two types of failures in the AW-SDX system: data plane failure on the SDX network path, and control plane failure on SDX controllers. Link failures will disrupt the communication between the SDX controller and one or more of the LCs. The development effort of the Control Plane bootstrapping, recovery and failure handling consists of adding the following functions to the SDX controller:

- Improving the current implementation, which does not handle any link failure and reconfiguration scenario. We implemented a new module in the AW-SDX controller that, when a link failure caused disconnection of the control plane network, the AW-SDX controller will recalculate and reestablish the control plane network using the backup ports and links.
- Thorough testing of the control plane to guarantee rule consistency when the AW-SDX controller and one or more local controllers are rebooted. We developed and validated the capability to recover the control plane network after we broke a link by disabling a switch port on the original control plane.
- Separating the control plane bootstrapping function to a standalone software.We developed a separate software module that only bootstraps the control plane. This would allow easy and clean debugging of the control plane problems.
- Separating the configuration manifest files for the SDX controller and individual local controllers. We restructured the manifest file and developed the necessary parsing code so that the LC manifest only need to contain the local network information.

SDX data plane failures. Link failures will cut off the existing connections and disable the capability to create new connections over the part of the network affected by the failed link(s). The development effort of the Data Plane recovery and failure handing consists of adding the following functions to the SDX controller:

- P2P and P2MP rule persistency (recovery) after switch and/or LC reboot.
- Due to the in-band control plane, a link failure may break both control plane and data plane networks. A complete failure handling function needs to coordinate the recovery of both networks.
- P2P and P2MP connection rerouting after link failures.

Figure 4 shows an example of the SDX failure handling mechanisms. In case of a link failure between the NCSU switch and the UNC switch, or UNC local controller node failure, the SDX system is able to coordinate between the SDX controller and NCSU's adjacent local controller–Duke in this case–to establish the link between San UNC and NCSU switches. Therefore, the control plane and data plane connection will be recovered.

C. Authentication

Due to the nature of multiple administrative domains, extra care needs to be taken to address the AAA (authentication, authorization, and accounting) and incomplete domain information (topology and resource abstraction). Authentication is handled by outsourcing to the Internet2 CILogon federated identity management service. CILogon provides an integrated open source identity and access management platform for research collaborations, combining federated identity management (Shibboleth, InCommon) with collaborative organization management (COmanage). Federated identity management enables researchers to use their home organization identities to access research applications, rather than requiring yet another username and password to log on.

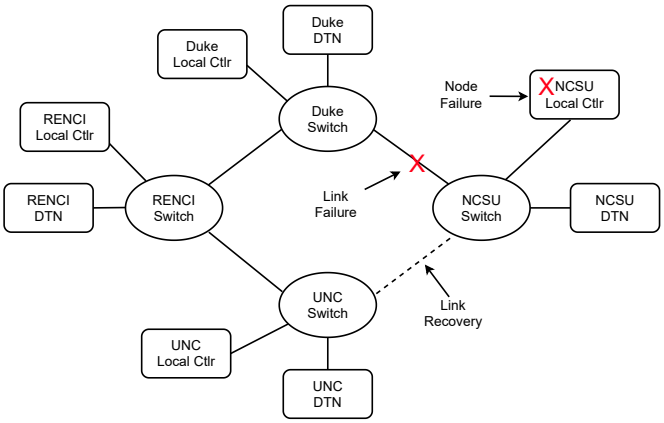


Fig. 4: SDX Failure Handling

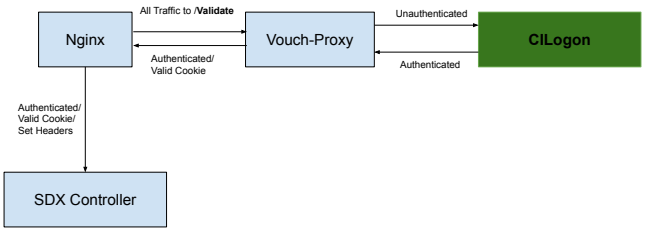


Fig. 5: Authentication CILogon Outsourcing

Vouch Proxy is an SSO solution for Nginx using the auth_request module that relies on the ability to share a cookie between the Vouch Proxy server and the application it protects. Figure 5 depicts our implementation using Vouch-Proxy (with Nginx) to enable authentication with CILogon's OpenID Connect (OIDC) service for gaining access to our AW-SDX web interface.

The implementation is customized to fit into the existing Flask web application framework that the AW-SDX portal and API are based on. We added two containers running a Vouch Proxy server and a Nginx web server to the SDX system to enable authentication using CILogon for users to gain the access to the AW-SDX web applications (portal and APIs). As shown in Figure 5, the authentication subsystem consists of an Nginx web server, a Vouch-Proxy server, and the application (SDX Controller), each of which runs in a separate Docker container. Vouch-proxy is a general OAuth/OIDC (Open Authentication/OpenID Connect)) login solution that supports many IdP (Identity Providers), including the CILogon Service that we chose to use. The login workflow consists of the following steps: (1) User visits the SDX web site from a local browser; (2) the Nginx reverse proxy server proxies the request to the Vouch server. (3) Vouch server maintains the state of this Login session. (3.a) If it's validated already, it returns a validate JWT (JSON Web Token) to allow the user to access the SDX service; (3.b) If not validated, the Vouch server will proxy the user to the CILogon; (4) the user can then login using her home institute's login service. (5) the Vouch service returns the validation back to the Nginx server.

III. Testing Pipeline

Testing is the most important step to guarantee functional behaviors and software quality. Due to its distributed system nature, the AW-SDX software system consists of multiple components interconnected via the SDX control plane network that need to maintain communication reliability and persistent internal states. Therefore, an emulation testing environment is important for sufficient testing of the basic AW-SDX control plane and data plane functions. As we discussed in Section II, data plane functions are highly device and network configuration dependent, requiring testbed-based testing and validation.

Unit tests Initial SDX software development has unit tests as an integral part of the development process, which covers the majority of the major functions responsible for the inter-controller communication over the control plane, the path computation and splitting between the LCs, and the APIs.

Containers and Mininet based emulation The technologies of Docker, OVS software switches, and Mininet software emulator have made it convenient to build a virtual network emulation environment for the basic system functional test of the multi-component AW-SDX controller. Furthermore, the entire emulation can be packaged and executed inside a virtual machine (VM); and within multiple containers running the individual controllers (SDX controller and LCs) and a Mininet network. In order to conduct repeated deployment and testing during the active development and testing, we opted to implement this example as a Jenkins job that is built and deployed to a VM running in the RENCI production data center.

Testbed As we discussed in previous section, emulation testing is far from sufficient. Therefore we built a Corsa testbed on top of an existing metro-scale optical network facility that RENCI maintains. This testbed spans four locations in the RTP NC area: RENCI, UNC, Duke, and NCSU, which are connected via a fiber ring network. As shown in Figure 6, we deployed 5 CORSA switches, two in RENCI, and one in each of the other three sites. Each switch was connected to a high-end server that hosts the SDX Local Controller (LC). The server also played the role of end hosts of the data plane connections. A dedicated VM was provisioned in the RENCI data center to host the AW-SDX controller (SDX-CTRL). This testbed mimics the current physical infrastructure of the targeted AmLight network. We note the physical cable connection between a pair of ports on the CORSA switch. Its purpose is for the AW-SDX software to provide a bandwidth rate limiting function.

All entities, including the switches and controllers, are first provisioned with a dedicated management network that allows the operator to remotely access them. Two VFCs are preconfigured in each Corsa switch. A dedicated VLAN 1411 is reserved for the in-band AW-SDX control plane. The in-band control plane topology (tree) is specified in the manifest file.

The AW-SDX controller and LC run inside Docker containers, whose image build, deployment, and start up are automated through our customized scripts and CI/CD tasks in the Jenkins server, which we describe in next section.

IV. Software Engineering and CI/CD Pipeline

A typical software development process generally falls into broad groups which can be defined as a Software Developement Life Cycle (SDLC) as: (1) Planning/Requirements (2) Analysis/Design (3) Development and Implementation (4) Testing (5) Maintenance

In addition, the emerging DevOps paradigm in system software development and deployment proved extremely valuable in improving the efficiency and availability in an operational product network environment [9], [10]. In this work, we closely followed best practices in software engineering and DevOps.

Regardless of the development team size, there are tools that can greatly aid in the management and curation of code development, testing and maintenance throughout the lifecycle of a project. These tools fall into two broad categories, source code version control and continuous integration / continuous delivery (CI/CD). In many cases these tools are free to use as publicly hosted web services, like the Github, or available as open source products, which can be built/hosted by the development team.

We created a CI/CD environment using the popular Jenkins platform hosted in the RENCI data center. Using this platform, along with the project Github repo, the processes of software integration, including automatic build and unit testing, and deployment into the testing pipeline (see Section III) were fully automated.

Porting to Python 3. The AW-SDX prototype is based on Python 2. Steps were taken to port the codebase to be Python 3 compliant, such that it maintains proper package support beyond the Python 2 end-of-life date on January 1, 2020. It is desirable to maintain the compatibility with Python 2. The porting process took substantial work, especially for a system software like AW-SDX that has dependencies on other complex software and many Python features that are not compatible in Python 3.

To ease the porting complexity, the Python organization provides the *Futurize* package to update the code. We used the 2-step *Futurize* process to solve the compatibility issues more efficiently. After this process, some deeper compatibility issues still remained.

Python 3 introduced several incompatibility issues in some basic types and data structures. The most notable one is that Python 3 distinguishes two string types, *str* and *bytes* that are equivalent in Python 2. Since the SDX LC functions involves the *bytes* type in generating, saving, and socket serializing the OF flow related rules, it required a manual fix in many places in the code.

Many data structures, including the dictionary, are no longer orderable in Python 3. so manual adjustments in many related data structures that AW-SDX implemented were needed.

Version control and Continuous integration (CI) The code is hosted in a Github repository and we re-organized it to the standard structure of three branch levels: Master, Development, and Features. Accordingly, we formally established our software development process: (1) individual developers create feature branches off the Development branch for new feature addition and bug fixes; (2) after unit tests, emulation tests, and peer review, feature branches can be merged into the Development branch for testbed deployment and tests; (3) after a certain milestone was met, the Development branch can be merged into the Master branch for a new release.

Good test coverage is key to minimizing bugs and issues from creeping into the code as changes are being made, which is frequent. Code coverage is a measurement of how many lines/blocks/arcs of code are executed while the automated tests are running. Python has a very strong notion of style and code format. Python based projects make use of multiple packages from official vendors, as well as third party contributors. It is often painstaking for a developer to keep track of the state of all packages being used within a project. Additional code analysis tooling exists that can be used to automatically inspect and generate interactive reporting for developers to use to improve their overall code efficiency, readability and maintainability. These are proven open source software tools to conduct the above tasks and they can all be automated as part of the Jenkins tasks. To help us improve software quality in a more efficient way, we implemented several of the tasks using the following open source tools: *PyIint* source code quality checker, *requiresio* package monitoring, *SWAMP* for continuous software assurance, and *CodeClimate* for code quality analytics.

Continuous Deployment For the testbed deployment, a Jenkins CD task was created to deploy the software from a chosen branch in the Github repo. This task was optimized by parallelizing the deployment of the SDX controller and LC into the hosting servers distributed over the test environment. As shown in Figure 7, the deployment process finished within four minutes. By comparison, an earlier non-optimized version took about ten minutes to finish the deployment in a series of actions.

A similar task was also created for the production network deployment.

The end result was the implementation of a complete CI/CD chain that automated the testing and deployment in the testing pipeline and production network.

V. Evaluation

We conducted extensive testing in the SDX testbed. The main performance evaluation results on both control plane and data plane are presented in this section.

Figure 8 shows the time-stamps read from the log file in the AW-SDX controller in milliseconds. It shows when the AW-SDX controller was ready and the connections to the four local controllers (LC) were ready in a rather short period of time.

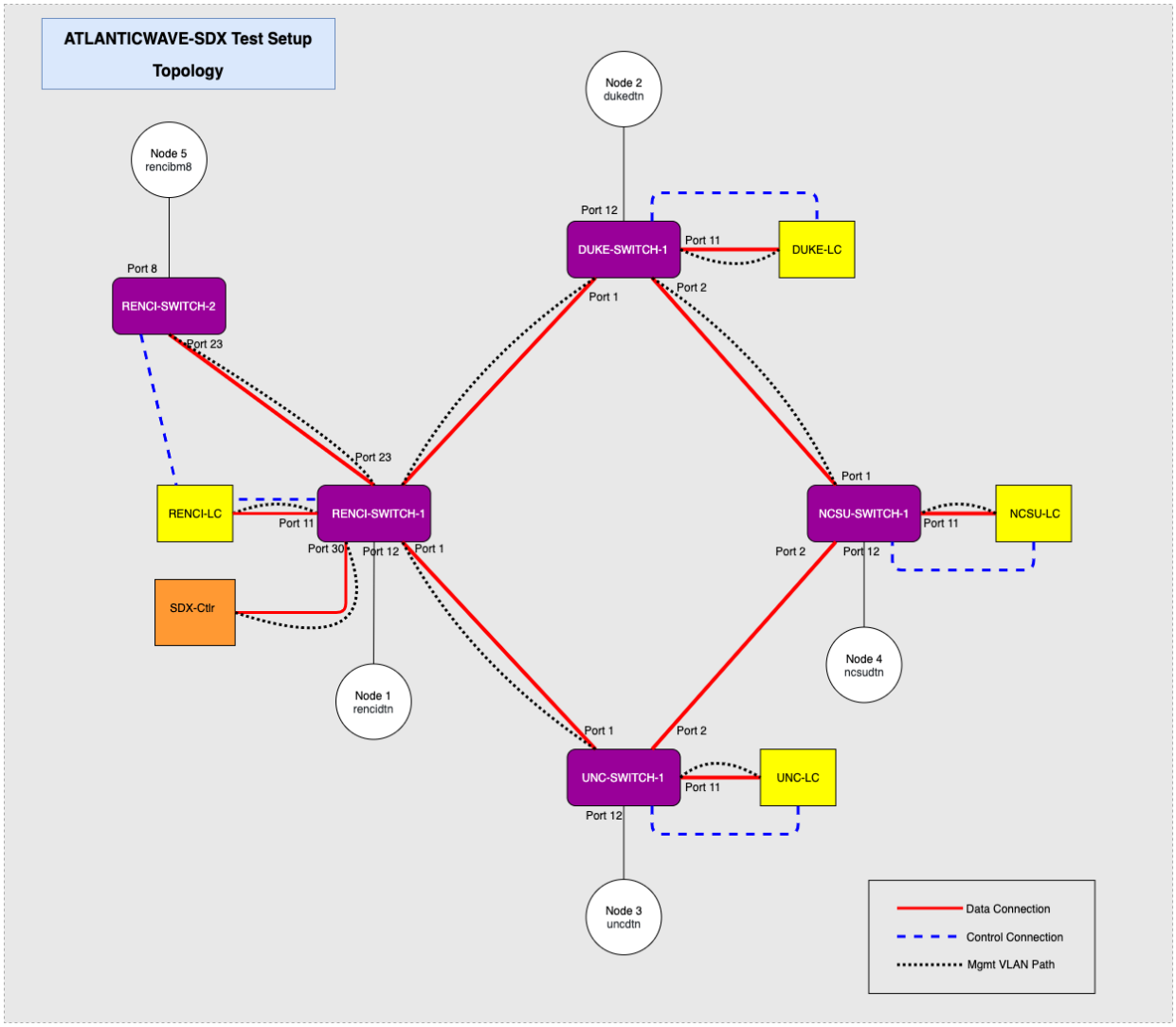


Fig. 6: BEN Corsa Testbed

In order to validate the correctness of dynamic connection provisioning and scalability performance, a large number of requests was submitted to the AW-SDX controller to validate that they were created, deleted and re-created, and that OF flows were pushed to the switches by Local Controllers correctly. L2Multipoint requests were created for connections with 3 and 4 endpoints respectively.

Figure 9 shows when different numbers of requests were submitted, the provisioning time scales changed in a linear manner. Figure 10 shows the number of OF entries in the OF switches also scaled linearly.

To validate the rate limiting capability, we used 'Iperf' between the end hosts of a connection to measure the throughput. Figure 11 shows one such test when a 2Gbps request was created and the effect of the rate limiting was clearly observable.

Finally, we show the result of running a scientific workflow application on top of the SDX testbed. The Pegasus Workflow Management System (Pegasus WMS) [11] bridges the scientific domain and the execution environment, such as compute, networking and storage infrastructures, by automatically mapping high-level workflow descriptions onto distributed resources. AW-SDX leveraged Pegasus' capabilities to orchestrate the science application execution on the underlying SDX infrastructure. In the demonstration, both 1000Genome and CASA Nowcast workflows were able to seamlessly run on AW-SDX's real-time provisioned networking and compute resources. AW-SDX was able to provision a high-bandwidth request, and automatically recover the synthetic link failure in the demonstration. The real-time AW-SDX performance data transfer was reflected on the Pegasus online monitoring system, shown in Figure 12.

VI. Conclusion and Deployment to the Production Network

Figure 13 shows a representation of the testbed. ATLANTA SWITCH, MIAMI SWITCH, and SANTIAGO SWITCH are

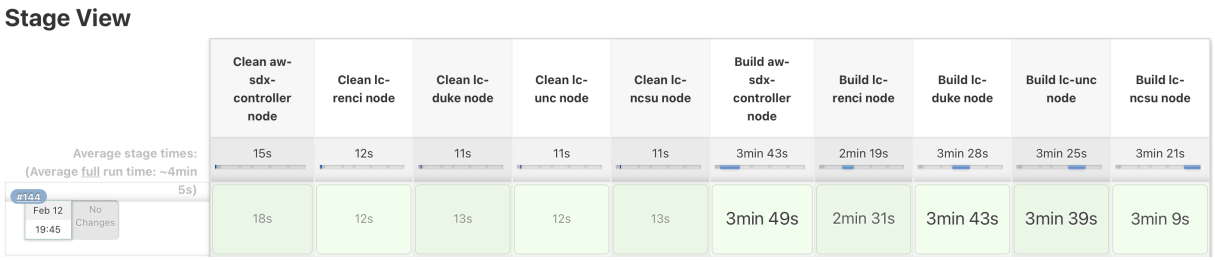


Fig. 7: CI/CD pipeline for the testbed deployment

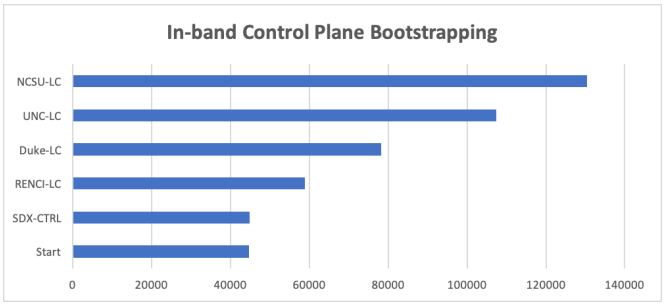


Fig. 8: SDX Controller Bootstrapping Time

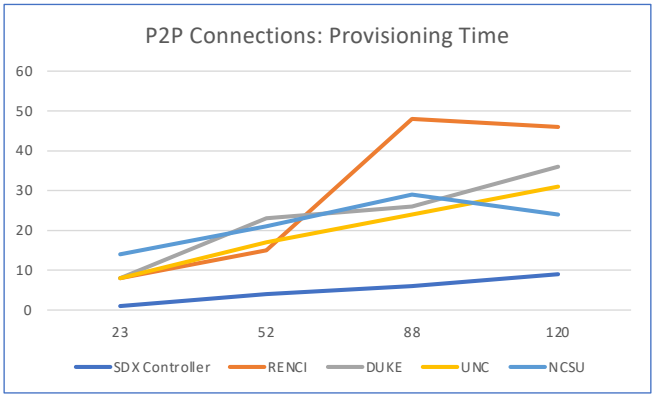


Fig. 9: P2P Connection Provisioning Time

Corsa switches deployed at SoX, AMPATH, and AndesLight respectively. Internet2 AL2S and Florida LambdaRail (not shown) are the transit networks between AMPATH and SoX, and AmLight is the transit network to AndesLight in Santiago, Chile. VLANs for Data, Control and Management planes have been deployed between SoX and AMPATH, and are in process to AndesLight. The AtlanticWave-SDX Production Setup Node Layout documents the port assignments for each Corsa switch (data plane) to Local Controllers running at each exchange point. The Production Setup Tunnel Layout documents the physical ports for uplinks and nodes, logical ports and management ports. The AtlanticWave-SDX Production Testbed Setup documents the configurations for the testbed, such as the Out-of-band management and In-band management configurations, Port tunnel-modes, Corsa Virtual

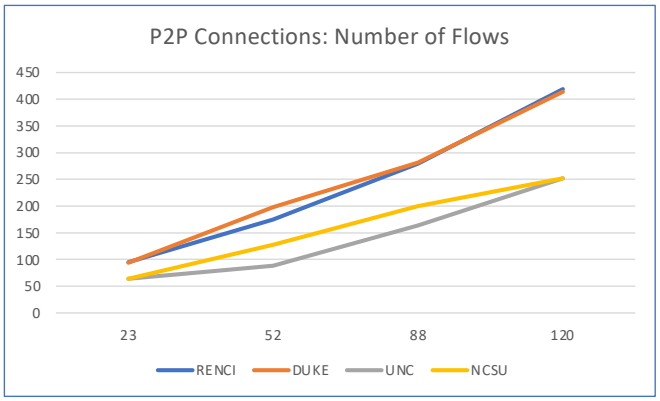
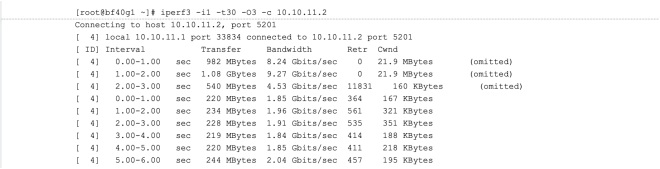


Fig. 10: P2P Connection Flow Entries



```
[root@bf40g1 ~]# iperf3 -i1 -t30 -O3 -c 10.10.11.2
Connecting to host 10.10.11.2, port 5201
[  4] local 10.10.11.1 port 33834 connected to 10.10.11.2 port 5201
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec   982 MBytes  8.24 Gbits/sec    0   21.9 MBytes       (omitted)
[  4]   1.00-2.00   sec  1.08 GBytes  9.27 Gbits/sec    0   21.9 MBytes       (omitted)
[  4]   2.00-3.00   sec   540 MBytes  4.53 Gbits/sec  11831    160 KBytes       (omitted)
[  4]   0.00-1.00   sec   220 MBytes  1.85 Gbits/sec  364    167 KBytes
[  4]   1.00-2.00   sec   234 MBytes  1.96 Gbits/sec  561    321 KBytes
[  4]   2.00-3.00   sec   228 MBytes  1.91 Gbits/sec  535    351 KBytes
[  4]   3.00-4.00   sec   219 MBytes  1.84 Gbits/sec  414    188 KBytes
[  4]   4.00-5.00   sec   220 MBytes  1.85 Gbits/sec  411    218 KBytes
[  4]   5.00-6.00   sec   244 MBytes  2.04 Gbits/sec  457    195 KBytes
```

Fig. 11: P2P Connection Rate Limiting

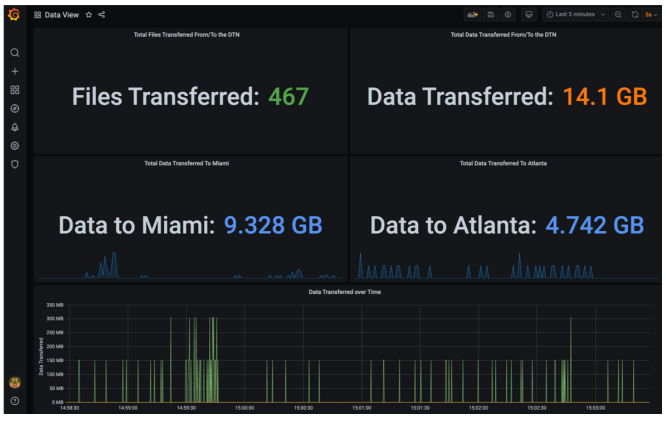


Fig. 12: Run Workflow Applications on AW-SDX Testbed

Forwarding Contexts (VFCs), Rate Limiting VFCs, and the scripts to start the SDX Controller and Local Controllers. Documents are all accessible in GitHub.

Yellow rectangles represent SDX Local Controllers (LC).